MSc in Computer Science 2017-18

Project Dissertation

Project Dissertation title: Implementing and experimenting with variants of Weisfeiler-Leman graph stabilisation

Term and year of submission: Trinity Term 2018

Candidate Name: Dario Asprone

Title of Degree the dissertation is being submitted under: MSc in Computer Science

Word count: 20421

Implementing and experimenting with variants of Weisfeiler-Leman graph stabilisation

Dario Asprone

Jesus College

University of Oxford

Submitted in partial completion of the

MSc in Computer Science

Trinity term 2018

To my father
and late mother
I hope you're both proud

Abstract

The Weisfeiler Lehman (WL) graph stabilisation is an algorithm presented in 1968 and later expanded to a hierarchy of algorithms that allows the user to obtain an approximation of the orbits/orbitals of a graph, and eventually the orbits/orbitals themselves.
At the moment there's no efficient, open-source implementation of the entire hierarchy of algorithms, much less in any integrated capacity with a general mathematics software library.
Purpose of this dissertation is to detail, test and evaluate a project aimed at developing a single parametric method that is able to run the entire WL family of algorithms, implemented in a possibly open-source maths library.
We provide a description of two such methods and a total of three different implementations for them, as an extension to the Sage Mathematical Software library, and then proceed to run a series of tests and benchmarks to check that they are both correct and efficient.
Results will show that I produced a general purpose, standard implementation of a commonly described k-WL algorithm and two very fast and memory light implementations of a novel algorithm for k-WL that is only slightly weaker than the first one, when the two are provided with the same value of k, or at least as powerful when to the latter is given a value of k increased by 1 w.r.t. the former.
In conclusion, the initial criterias for the desired implementation of k-WL were met, in the meantime probably also producing an equivalent but memory lighter version of the deep stabilisation algorithm by, again, Weisfeiler and Lehman.

Contents

List of Figures

1 Introduction

The Weisfeiler Lehman method is an algorithm initially presented in [1] in 1968, exactly 50 years ago, with the aim of tackling the graph isomorphism problem through the computation of what was called a minimal coherent cellular algebra's regular representation.
Testing graphs for isomorphism was and still is a highly evasive problem, known to not be NP-complete but also with no polynomial complete algorithm yet found able to solve it. Such is the nature of the problem it was hyperbolically defined a disease in [2], and an entire complexity class has been defined, the **GI** class, for any problem that can be polynomially reduced to testing graph isomorphism.

Initially, Weisfeiler and Lehman conjectured that all graphs (bar a handful of exceptions) shared a property that allowed their algorithm to obtain the orbitals of any graph's automorphism group in polynomial time, thus solving the isomorphism problem [3].
That assertion proved untrue, but the idea gave birth to a hierarchy of algorithms named after the authors of the paper.
The idea behind the hierarchy, named *k-dim WL* by Babai, k-WL for short, is to initially colour k-tuples from V^k based on the structure of the graph itself (colouring two k-tuples the same if the direct mapping between their corresponding elements is an isomorphism in the original graph) so as to divide them in equivalence classes, then building for each k-tuple a multi-set of adjacent k-tuples' colourings, for some definition of adjacency, and again create a finer partition based on multi-set equality. The commonly used adjacency is the 1-hamming distance [4]
Problem with 1-WL and 2-WL was that they could only be trusted when the result

was negative: given two graphs, if the Weisfeiler Lehman method says they are not isomorphic they surely are not, while a positive result might just indicate that WL couldn't distinguish them, and doesn't imply actual isomorphism. Nonetheless, Babai proved in two papers that a normal form can be produced by 1-WL for almost all graphs, precisely for all but a $n^{-\frac{1}{7}}$ fraction of the graphs of order n in [5] and for all but a $c^{-\frac{n \log n}{\log \log n}}$ fraction, for some constant c, in [6], making this algorithm very promising for isomorphism solvers.
Later on, another conjecture was made that there was some value of k for which k-WL could produce a normal form for all graphs of bounded valence, or that at least there was a value k_d for each valence d that could allow k_d-WL to produce normal forms for the relative graphs, but both claims were drastically proven wrong in [7], where a family of pairs of cubic 4-colourable graphs was presented that had at least one member that could not be recognised by k-WL for any value of k.

While this shattered the dream of using k-WL to single-handedly solve the graph isomorphism problem, the algorithm succeeded in staying relevant for modern applications too.
As we will see in the following chapter, 1-WL is currently used in the best practical isomorphism solvers, such as *nauty* [8], while k-WL has seen applications in linear programming, machine learning, linear algebra and the counting of graph invariants. Its usage and effects on graphs are still studied, as papers published just a year ago proved that all planar graphs can be recognised by 3-WL [12], or showed which invariants the algorithm makes use of to distinguish graphs during its execution [24].

While k-WL has seen a huge number of papers published that study its theoretical properties, at least for the lower orders of the WL hierarchy [9], many of them never rely on any real implementation of the algorithm or make use of a single-use quick one developed by the paper's author, so there's no real common, efficient version of k-WL that could be used for standardised research and experimenting purposes,or also to test and verify results found in literature.
If an implementation of a generic k-WL algorithm even exists, from a literature search it seems to be either integrated in some other tool without any public interface and/or in a closed source, obscure implementation, since I was not able to find any mention of such a program.
Other than being efficient, generic and possibly open source so that other researchers can analyse its code, an ideal implementation of k-WL would also have to be specific, that is be developed with the precise objective of providing a k-WL algorithm for general use instead of as a subroutine for some other application, and it being integrated with other mathematics research tools would make it perfect.

Here is where my project comes into play, aiming to provide an open source, efficient implementation of k-WL integrated in a fully fledged mathematics software library to allow easy interaction with it.
A very important part of the project was choosing a suitable platform that could support both the development and the later use of my implementation in a proper manner. The choice fell onto *SageMath* [10], a free, open source mathematics software system that provides libraries for a huge range of maths branches, such as algebra, calculus, number theory, cryptography, group theory and, most importantly for my project, graph theory. The library was initially developed as an aggregator of several open source packages that were used through a common Python-derivative interface, but then expanded through the addition of its own scripts in the Python or Cython language.
All the algorithms, the test case generators and interfaces were developed natively as an integrated part of the SageMath library.
A secondary aim of my project was also to develop a variant of the k-WL algorithm described, for example, in [11] or [4], one that could improve with respect to the standard version on either speed or memory usage; since results were published that the currently best known implementation of 1-WL has an optimal asymptotic time complexity given a series of reasonable assumptions on the type of algorithm used, and since the best current algorithm for general k-WL is based on the same principle as the one for 1-WL, I set on producing a variant that improved the space efficiency of the algorithm without affecting too much its time complexity.

This report consists in a full description, analysis and testing of the produced algorithms, and is divided in chapters as follows:

1) This first introductory chapter, explaining the background, previous research, importance and aims of the project.

2) A chapter detailing in full the possible applications of a k-WL algorithm, with references to relevant papers and explanations for the main theorems and results connected to k-WL in each field.

3) Full description of 1-WL and k-WL as presented in the main relative papers, followed by an in-depth analysis of the two algorithms that I implemented for my final project and their description through pseudocode

4) Closer look at how each algorithm has been implemented, describing the chosen data structures, the trade-offs and algorithmic choices made. After each in-depth explanation, a section is provided with the full space/time complexity analysis of the algorithms *as actually implemented*.

5) In this chapter the main test cases for the k-WL implementations are presented, accompanied by the results and snippets of SageMath's scripts that allow the reader to replicate the experiments.

6) Final chapter that wraps up the project and comments on the results obtained, with a section of possible future developments of the work carried out.

Note

Regarding the word count on the cover sheet, it was calculated including every word in every header, caption or general text section, including the references and the abstract, but excluding the snippets in chapter 5, the algorithms in chapter 3 and the mathematical formulas in general. Also, as stated again later on in the document, the appendices were not included since they consist of listings, which by regulations are not to be included in the word count for up to 30 pages of them.

2
Applications

Contents

2.1 Overview

As mentioned in the previous chapter, the *k-dimensional Weisfeiler Lehman* (k-WL for short) method is an algorithm (or family of algorithms, depending on how one considers the k parameter) that splits the vertices and ordered pairs of vertices of a graph $G(V, E)$ into classes, depending on the similarity of k-tuples in V^n that include them; this process of splitting graph elements is called coloring, and the classes in which they are split are called color classes.
It needs to be said that k-WL actually colors the k-tuples themselves, but that can be used, as we'll se later on, to accordingly color the vertices and edges of the graph they are generated from.

A note must also be made on how we will address pairs of vertices from now on, in the scope of k-WL coloring: while it's true that we use k-WL on a graph $G(V, E)$ to color either its vertices or pairs of them, an ordered pair of vertices can be seen as either an edge in G or in $\overline{G}$ (plus pairs of the kind (v, v), which are self-loops and need special consideration on a case by case basis), so we will refer to pairs of vertices colored by k-WL as edges during the rest of this text, meaning directed edges of the complete graph $\hat{G}(V, E \cup \overline{E})$.

While k-WL's usage and effect is pretty simple and straightforward, understanding how to use the obtained color classes, what can be done with them, but also their limitations is of primary importance to gauge the relevance of the project subject of this dissertation.

Also listed in the introductory chapter, the main applications for k-WL are in Graph Isomorphism Testing, Equivalence Testing for the logic $\mathcal{C}_{k+1}$, Linear Programming Pre-Processing, and Machine Learning.
We will go through them one by one in the following sections, and also comment at the end on a couple of interesting connections that WL has with algebra and graph invariant analysis.

2.2 Graph isomorphism

This is the first and oldest application for k-WL, the one for which the algorithm was designed and presented in [1].
Since in the original paper the algorithm was presented as a procedure to be applied on a graph's adjacency matrix, without any particular name or k-dimensional version, we will refer to it as *Original-WL* (or O-WL for short), keeping in mind that O-WL is equivalent to the 2-dimensional version of k-WL.
O-WL was introduced as an algorithm capable of computing a canonical form for a graph (specifically, one with a lexicographically minimal adjacency matrix) and thus also capable of deciding if two given graphs are isomorphic or not. It was found out in later research that the classes returned by O-WL when run on a graph G did not actually correspond to the orbits/orbitals of its automorphism group, but were instead a partition of G's vertices/edges that could be either equal to the orbits/orbitals (which are also a partition of the vertices/edges) or coarser than them; while it's not the counter example given originally against O-WL, which was presented only a year after the paper's publication, a simple graph that shows the issue is the Shrikhande graph, shown in fig. 2.1: while the orbits are immediately recognised, the orbitals of the graph, that is the orbits on its edges, are not correctly

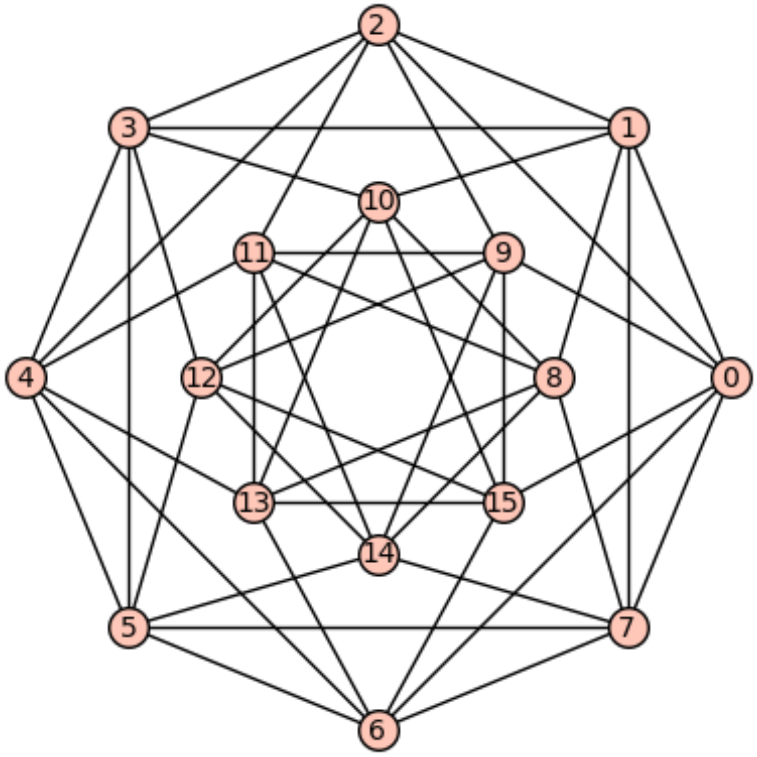

Figure 2.1: Depiction of a strongly connected graph with degree 6, that cannot be recognised by 2-WL

returned by O-WL.

The same problem occurs with any strongly connected, regular graph whose orbitals are not the trivial partition, as it is obvious from the way O-WL works, but we will return to this in chapter 3, where we will analyse the Weisfeiler Lehman algorithm more in detail. In general, when O-WL (or k-WL in general), run on a graph G, returns a partition that is strictly coarser than the orbits of G, we will say that O-WL (resp. k-WL) does not *recognise* the graph.

It doesn't matter how many graphs are not recognised by O-WL, heart of the matter is the fact that just because unrecognised graphs exist, O-WL can't be used as a complete algorithm to determine if two graphs are or are not isomorphic.

At this point, one could be tempted to say that the algorithm is useless, but far from it, there are a lot of things that can still be done. For starters, 1-WL recognises almost all graphs and all trees and forests, as seen in [5] and [4], though it recognises none of the regular graphs (those that don't have trivial orbits at least); furthermore, each order of the Weisfeiler Lehman procedure recognises strictly more graphs than its lower dimensional counterparts [7] and 3-WL already recognises every planar graph [12]. All of this clearly indicates that the case in which the returned result is not equal to the orbits of the graph given as a parameter is rare for 1-WL already,

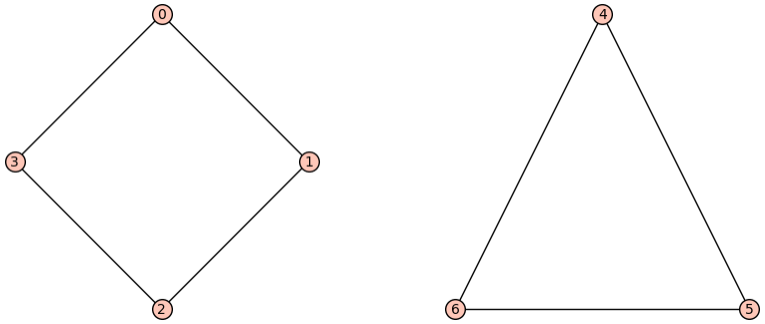

Figure 2.2: Depiction of a graph resulting from the disjoint union of a cycle graph of length 4 (on the left) and one of length 3 (on the right)

and gets increasingly rarer the higher up we go with the k parameter; also, for higher values of k, the result is often times just slightly coarser than the orbits, so the work needed to recognise the graph completely is much less than if we started with no partial solution at all.
The only real problem left, then, is that we can't blindly trust the result coming out of k-WL. If used to compute the orbits, one must remember that there is a possibility the result is just a partition that can be refined into the orbits, but not necessarily equal to the orbits themselves; if used to test two graphs G and H for isomorphism, what can be done in lack of the possibility of computing a real canonical form is to run k-WL on the disjoint union $G \dot\cup H$ and check the returned classes: if the vertex labelling induced by the result on the graphs makes them different, we can be sure they are not isomorphic and will say they are *distinguished* by k-WL, otherwise we will have to check each colour class to be sure that the newfound graph equality is not changed by eventual refinements of the result.

In summary, k-WL can be safely used as a negative isomorphism test between two graphs, but requires a second algorithm to be employed in order to check a positive result is not fake, and is thus incomplete; still, k-WL can be used as a subroutine, an heuristic, in a proper isomorphism testing algorithm.

2.2.1 Particular graphs

A particular mention is needed for graphs (or better, families of graphs) that behave peculiarly when k-WL is used on them. We have already discussed how regular graphs are problematic for 1-WL, since all vertices have the same degree and this poses some issues if the orbits are not equal to the trivial partition, as can be seen for example in fig. 2.2. We also saw how it was determined that all planar graphs can be recognised by 3-WL, but there are still a few families of graphs that are interesting to analyse:

Strongly regular graphs

Strongly regular graphs are regular graphs of order n and degree k whose any pair of adjacent vertices has l common neighbours and any pair of non-adjacent vertices has p common neighbours. Such graphs are determined by the listed parameters, and referred to as SRG(n, k, l, p). Note that two graphs might have the same parameters without being isomorphic.

This kind of graphs are notoriously not recognised by k-WL for $k < 3$, are sometimes recognised for $k = 3$ without any currently known pattern, and there are some strongly regular graphs that need even higher values of k to be recognised.

Egawa graphs

Egawa graphs are graphs introduced by Yoshimi Egawa in [13] for the purpose of classifying distance regular graphs. In particular, given a distance regular graph $G(n, q)$, if $q = 4$ then G is isomorphic to some Egawa graph $I(p, s)$, with $n = 2p + s$. These graphs are cospectral mates with Hamming graphs, but their peculiarity is that, while their orbits are correctly returned by k-WL for k as low 1, their orbitals are not correctly returned for any k-WL with $k \leq 2$: 3-WL is needed if $s = 0$, or an even higher order than that otherwise . Among their properties, we can also find that I(0,s) is isomorphic to H(s,4) where H is the letter standing for the Hamming graphs, and the fact that no two Egawa graphs with different parameters are isomorphic.

One eminent representative of this family of graphs is the Shrikhande graph, equivalent to $I(0, 1)$.

Paley digraphs

Paley digraphs are a special case of strongly regular digraphs, whose number of vertices N is equal to $N = 4n - 1$, where n is a prime power. They are the directed version of Paley graphs and are used to produce tournaments in which every small subset of vertices is dominated.

The peculiarity of this family of tournaments is that both their orbit and orbitals are not correctly returned by k-WL for $k = 2$ [14].

Cai-Furer-Immerman graphs

Presented in [7], Cai-Furer-Immerman graphs are not a family of graphs in the common meaning of the expression, it would be in fact more correct to talk about a Cai-Furer-Immerman construction (CFIC for short), a process that, starting from a base graph, produces a new graph with interesting properties;

to precisely state what these properties are and why they are interesting to us, we first need to introduce the concept of Furer gadgets, a family of graphs introduced in the same paper mentioned above and used as a support in the Cai-Furer-Immerman construction.

Furer gadgets are a family of graphs with a single parameter, referred to as F_f, and whose components have a pretty simple structure: they're made up by a middle layer of 2^{f-1} vertices, f "upper" vertices labelled from a_0 to a_{f-1} and f "lower" vertices labelled from b_0 to b_{k-1}. Each middle vertex is labelled with an element of even cardinality from the power set of $\{0, \dots, f-1\}$, and connected to exactly one of either a_i or $b_i, \forall i \in [0, k[$; specifically, a middle vertex labelled with the set M will be connected with a_i if $i \in M$, with b_i otherwise. The important aspect of Furer gadgets, which is what makes them useful for the construction in question, is that any mapping that only swaps pairs of vertices a_i and b_i is an automorphism if and only if the number of such swapped pairs is even.

Now, with Furer gadgets covered, the Cai-Furer-Immerman construction on a graph $G(V, E)$ is pretty straightforward, as it consists in first swapping each vertex $v \in V$ with $F_{d(v)}$, where $d(v)$ is the degree of the vertex v, being careful to name each vertex in the Furer gadget so that it has a unique name in V, and then substitute every edge as follows: given an edge $(u, v) \in E$ and calling the Furer gadgets they were swapped out for F_u and F_v with a little abuse of notation, in place of (u, v) two edges will be added that go from (u, a_i) to (v, a_j) and from (u, b_i) to (v, b_j), where (u, a_i) and (u, b_i) are a pair of outer vertices in F_u not yet used for any edge substitution, and (v, a_j) and (v, b_j) are the same for F_v.

One last but very important addendum is that there exists a second, slightly different, version of the Cai-Furer-Immerman construction defined *twisted*, where for one and only one edge $(u, v) \in E$ the new two edges to be introduced will have their ends *twisted*, that is they will go from (u, a_i) to (v, b_j) and from (u, b_i) to (v, a_j).

Note that it's possible to adapt the construction to digraphs, but such version won't be covered here since it doesn't interact in any interesting way with k-WL.

What's important about this family of graphs is obviously not their construction, but how some particularly constructed graphs respond to k-WL: the real breakthrough in [7] was that for any graph H that has no balanced vertex separator smaller than s, no k-WL with $k < s$ can distinguish the graphs

resulting from applying CFIC in its normal and twisted versions on H.
Seeing why this is true is certainly not easy, and I'd suggest reading the original paper for further details, but a good summary would be that Cai, Furer and Immerman first prove that the normal and twisted versions of the CFI construction on H (let's call them $\hat{H}$ and $\tilde{H}$ respectively) are not in fact isomorphic, then that k-WL has the same graph distinguishing power of a quantified first-order logic whose formulas can use k+1 variables (called $\mathcal{C}_{k+1}$), and finally through a game theoretical approach they prove that at least $\mathcal{C}_{s+1}$ is needed to distinguish $\hat{H}$ and $\tilde{H}$ if H has no vertex separator of size s or smaller; specifically, they prove that each Furer gadget can "hide" the twist introduced in $\tilde{H}$ from $\mathcal{C}$ once per game, at the cost of reducing the size of the balanced vertex separator by one, so if the number of turns (which is equivalent to number of variables used by $\mathcal{C}$) is at most s the twist will never be found and $\mathcal{C}$ (or WL for that matter) will not be able to distinguish the two graphs.

2.3 Equivalence testing for $\mathcal{C}_{k+1}$

As mentioned in the paragraph of section 2.2.1 about Cai-Furer-Immerman graphs, k-WL has been proved to be a suitable equivalence test for assignments in the fragment of first order logic that uses at most $k+1$ variables, when extended with counting quantifiers (that is $\mathcal{C}_{k+1}$).
The connection was first proved in [4] and then re-proposed by Cai, Furer and Immerman in [7] and used to prove that k-WL is not a complete isomorphism testing algorithm, even for bounded degree graphs or k-regular graphs (since it's possible to have a family of graphs with increasing size of vertex separator but fixed degree or k-regularity, i.e. expander graphs).

The proof itself is pretty simple, and consists in proving by induction on the number of refining rounds of k-WL that, given two graphs $G_0(V_0, E_0)$ and $G_1(V_1, E_1)$, two k-tuples of vertices from V_0^k and V_1^k are in the same colour class arising from k-WL if and only if they agree on every formula in $\mathcal{C}_{k+1}$ when taken as k-configurations.
A note is needed on the fact that the agreement on the formulas is tested through a game theoretical approach using a two player game with an upper limit m on the number of moves: the m-moves game is winnable (by player 2, which is our goal) if and only if the graphs with the given configurations agree on all formulas with quantifiers whose level of nesting is at most m. It obviously follows that the

game is always winnable by player 2 *iff* it's winnable for every value of m, and similar considerations hold for the agreement on formulas

The base case is straightforward, since for 0 refining rounds k-WL returns colour classes induced by the initial vertex colouring, and so two vertices from G_0 and G_1 are in the same colour class if and only they belong to the same isomorphism type (by construction in the paper), which is the definition of a 0-move win in the game set up in the paper and thus is true *iff* the graphs agree on all formulas without quantifiers.

The inductive step is a lot more complex, and since the complete proof works on a three-way equivalence predicate it uses a circular proof as follows:

1. If two k-tuples were not in the same colour class after r refinement rounds ($\neg a$) then a formula with r nested quantifiers would not be agreed on by the two given graphs ($\neg b$)

2. If there's a not-agreed-upon formula with r nested quantifiers ($\neg b$), then there is a game with at most r moves where player 2 cannot win ($\neg c$)

3. If the colour pattern induced by k-WL after r refinement rounds is the same on both graphs (a) then there is a winning strategy for player 2 in the game with at most r moves (c) that consists in using the similarities highlighted by k-WL.

From the three points above follows that $a \Rightarrow c$, $c \Rightarrow b$ and $b \Rightarrow a$, thus proving the three-way theorem and the fact that k-WL is a perfect test for $\mathcal{C}_{k+1}$'s equivalence.

2.4 Linear programming pre-processing

Linear programming is a branch of Operations Research concerning algorithms for solving linear optimisation problems, that is problems with a linear objective function to be optimised (either in positive or in negative) and linear inequality and equality constraints on the variables used in the problem definition.
Without delving too much into details, it is known that solving linear optimisation problems is in P, as they are solvable with the ellipsoid method in polynomial time on the size of their binary representation, but in practice the method is not much feasible and even the more common simplex method starts getting slow when too many constraints are added to the problem. One of the ways to speed up the solving of linear problems is reformulating them, that is trying to solve an equivalent but smaller problem, removing redundant or similar constraints.

First in [15] and then in [11], a method is shown that allows to reduce the size of most linear programs (where such a thing is possible, obviously) in polynomial time and also to linearly map the feasible (resp. optimal) solutions of the reformulated problem to feasible (resp. optimal) solutions of the original one, using a link between 1-WL and linear algebra through the idea of fractional isomorphisms, presented in [16] and then in [17]; to explain what the method is, we first have to go through a few concepts.

2.4.1 k-WL and fractional isomorphisms

In [16] Tinhofer finds a connection between colour refinement (1-WL) and the isomorphism problem when it is expressed as a linear program.
Given two graphs (without labelled edges, but adapting the construction is relatively easy) $G(V, E)$ and $H(W, F)$ with disjoint vertex sets, we call A and B their adjacency matrices, we say that they are isomorphic if there is a matrix P s.t. it has one entry with value 1 per row and column, all the other entries are 0 (that is, P is a permutation matrix), and $P^T BP = A$.
The linear program obtained by using as constraints the equation $BP = PA$ (which is true because P is a permutation matrix, and thus orthogonal) and all the equalities and inequalities needed to force P to have the properties stated above is called **DSI** in the original paper (standing for *Doubly Stochastic Isomorphism*).
It's easy to see that any feasible *integer* solution to DSI is an isomorphism between G and H, also because there's really no way to "rank" isomorphisms and thus no way of establishing which isomorphism is optimal; this is why we will only consider feasible solutions for DSI from now on.
Since finding an integer solution for DSI is hard (as it corresponds to testing isomorphism, and we're still not really sure how hard *that* is), Tinhofer focused on its real (or rational) solutions and called them fractional isomorphisms; from the fact that the resulting real-valued matrix is doubly stochastic, in that its rows and columns sum to 1, comes the name DSI for the linear problem. We'll say that there is a fractional isomorphism between G and H if there is any real solution for DSI that is not also an integer solution.

The important result of Tinhofer's paper we're interested in is that there is a fractional isomorphism between G and H if and only if 1-WL cannot distinguish them. The proof is not exceedingly difficult, while rather tricky:

- The forward statement is proven by taking the fractional isomorphism matrix P, result of DSI, and creating a graph Z that has the union of G's and H's

vertex sets as vertices and en edge between every two vertices $v, u \in V \cup W$ s.t. $P_{v,u} > 0$.
A colouring for the disjoint union of G and H is then obtained by taking the vertex sets of the connected components of Z and making it so that two vertices have the same colour iff they belong to the same component; finally, it is proven that such a colouring is stable using the constraints of DSI, and as such it is returnable by 1-WL

- The backward statement is proven in a much simpler way, by simply taking the colour classes $C_1, \dots, C_r$ returned by 1-WL for $G \dot\cup H$ and constructing a solution P that has $\frac{1}{|c_i|}$ as a value for any entry $P_{v,u}$ s.t. $u, v \in C_i$ for some i, and 0 everywhere else, with c_i being the colour class C_i restricted to the vertices of G (this is possible without losing generality because by the hypothesis G and H were not distinguished); through some straightforward calculations it is then checked that such constructed P satisfies all the constraints of DSI, thus being a real feasible solution, and thus a fractional isomorphism.

We are still limiting ourselves to 1-WL though, which is interesting, but not enough. In [11], Tinhofer's theorem is extended to k-WL as follows.
When trying to solve a integer linear problem, one approach is exactly the one we presented in the previous paragraph, that is dropping the integrality constraints and solving a normal linear problem, trying then to approximate the obtained solution to an integer one. The problem when doing this is that the solution space of the relaxed problem, while including the entire search space of the integer problem, might also include feasible solutions that are not close enough to the integer ones; a way to mitigate this issue is to add linear constraints to the linear problem so that the set of integer feasible solutions stays the same, while the continuous solution space gets shrunk; to do so systematically, hierarchies of constraints have been designed, one being the *Sherali-Adams hierarchy* [18].
When adding the constraints from the k^{th} and $(k+1)^{\text{th}}$ levels of the Sherali-Adams hierarchy to DSI, we obtain a new linear program, called $\text{DSI}^{(k)}$ in Grohe's paper, that has a rational solution if and only if k-WL cannot distinguish the two graphs on which $\text{DSI}^{(k)}$ is built.

Another very important result for the application of 1-WL we are describing is the extension of the theorem above (the one regarding 1-WL, not its multi-dimensional extension) to real weighted (and thus edge labelled) graphs by making 1-WL refine not on vertex degree but on the sum of the weights of a vertex's edges. The extension of the result is the naive one for graphs with symmetric

adjacency matrices (undirected graphs or directed graphs that are isomorphic to an undirected one), while it gets slightly more complicated for arbitrary matrices that can be not symmetric (or even not square); in summary, given an arbitrary matrix A indexed by two sets V_1, V_2, it's done by defining the concept of a stable bi-colouring as a stable colouring of the matrix

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

split into two colourings, one for each set, and of a fractional bi-isomorphism between two matrices B and C with the same shape of A defined above, that is two doubly stochastic matrices X_1, X_2 s.t. $BX_2 = X_1C$, and then adapting the proof we used previously to this case. Furthermore, a definition is given of a fractional automorphism of an arbitrary matrix A as two doubly stochastic matrices X_1, X_2 s.t. $AX_2 = X_1A$, and it is explained how to construct two such matrices for any A we have a stable bi-colouring for.

2.4.2 Using 1-WL to pre-process

Finally we see how 1-WL can be used to reduce the size of a linear optimisation problem.

After defining all of the concepts in the previous two subsections, explaining the method is not very difficult.

Starting with a simple linear system $A\mathbf{x} = 1$, where A is a $m \times n$ real-valued matrix and the other symbols have the obvious definition, the idea presented in the paper consists of first constructing 4 matrices P, P^*, Q and Q^* from a stable bi-colouring (C, D) of A, such that the matrices $X = PP^*$ and $Y = QQ^*$ have the same entries as the ones used in the construction of a bi-automorphism as explained above; such matrices are then used to obtain a new matrix $A' = P^*AQ$ with dimension $|C| \times |D|$, and finally it's proven that if $\mathbf{x} = \mathbf{y}$ is a solution to $A\mathbf{x} = 1$, then $\mathbf{x}' = Q^*\mathbf{y}$ is a solution to $A'\mathbf{x}' = 1$ and conversely that if $\mathbf{x}' = \mathbf{y}$ is a solution to $A'\mathbf{x}' = 1$ then $\mathbf{x} = Q\mathbf{y}$ is a solution to $A\mathbf{x} = 1$.

Finally, the proof is extended to a linear program of the form

$$\begin{aligned} \max_{\mathbf{x}} \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

by finding a bi-colouring for A (a matrix indexed by the vertex sets V and W) through 1-WL, using an initial colouring that assigns the colour b_i to each vertex

$i \in V$ and the colour c_j to each vertex $j \in W$.
We can then apply the same process stated above to compute a feasible (resp. optimal) solution for the transformed linear program and then map it to a feasible (resp. optimal) solution for the original linear program.

Of course, all of this is at the moment applicable only to 1-WL, since there is no published research on using the extension to k-WL of the Tinhofer theorem for this purpose, but given the similarities between the two cases, it's safe to assume that there is a way to exploit k-WL superior recognising power to speed up linear programming optimisation.

2.5 Machine learning

This last application of k-WL is fairly recent, but is also one of the most promising applications of k-WL for widespread use, outside of graph theory and linear programming. First, a disclaimer is needed on the fact that the Weisfeiler Lehman method is mostly used in machine learning up to the second order, with the first order being wildly more used than the second.
This is due to a variety of reasons, that can be however summarised in two main points:

- For the usage required by machine learning, space and, mainly, time efficiency are essential, since WL will be called repeatedly, in the order of millions of times or even more, and using any algorithm that is more than quadratic (as 2-WL is) in such a way is a stretch already.

- While for mathematical applications 1-WL might not be enough, since it's easy to encounter highly regular graphs and structures, in practical applications 1-WL recognizes almost all graphs as the number of vertices increases, and two random non-isomorphic graphs are almost certainly distinguished by it, as proved in [5]

To further the second point, in a machine learning setting, as we will see, it's better if k-WL doesn't succeed in distinguishing two given graphs as the objective is to cluster them roughly by general structure similarity; if we let individual graph details complete separate each graph from all the others, we won't obtain any useful result.

One of the applications of machine learning is pattern analysis on a set of objects, that is separating or classifying elements based on the similarity of their features. To do so, the naive way is to create large vectors of such features and comparing them with some measuring function, so that it can be determined how similar or

different the original objects are from each other.
A way to speed up this process is to use a function that measures an element directly, called a kernel, and compare the distance of the measurements themselves, skipping the process of building and comparing a high-dimensional vector, in order to, finally, classify the inputs through functions called *kernel methods*.
Important as to why k-WL might be relevant in this field is that there exists a branch of machine learning that works with graph structured data, that is pieces of data presented in the form of a vertex-coloured, edge-labelled graph. Several kernels for graphs have been developed, with the most successful ones being those that compute graph invariants and use them as features (i.e. number of triangles, k-regularity, etc.), see [19] for a comprehensive review; however most if not all of these kernels share the same drawback, that is being particularly computationally expensive, especially when faced with larger graphs.

Here's where k-WL (or at least its lower-dimensional versions) comes in: we can run k-WL on the disjoint union of two graphs for a certain number r of refining rounds, and then take as a measure of similarity the sum over each round of the number of pairs of vertices in the same colour class. With the same reasoning used above of trying to avoid using too many details in our measure, it's best to keep r low, with a series of experiments in [20] establishing an empiric best value of 5 for it.
We can see that, using 1-WL, it's possible to compare two graphs in $O((m_1 + m_2)r)$ where m_1 and m_2 are the number of edges of each graphs, but this would give us a total running time of $O(N^2 \cdot m_{MAX} \cdot r)$, where m_{MAX} is the maximum sum of number of edges of different graphs and N is the total number of graphs to compare, so it's sometimes preferred to compute a vector of the number of pairs of vertices with the same colour (one entry per refinement round) for each graph, and then compute the scalar product of each pair of vectors, reducing the running time to $O(N \cdot r \cdot m + N^2 \cdot r)$, where m is the maximum number of edges in any considered graph.

The Weisfeiler Lehman graph kernel, such is the name given to it by its creator, has found successful application in source code comparison, medical analysis and even in fields where the data is continuous and not discrete, such that it cannot be represented with a graph, through a combination of randomised techniques and hashing, as seen in [21].

2.6 Interesting usages

Similar to the connections to linear algebra and machine learning, there are another couple of interesting use cases for k-WL that have, however, a far less extensive field of application: cellular algebras and graph invariants recognition.

2.6.1 Cellular algebra

Cellular algebras are a different but completely equivalent way of interpreting how k-WL works, studied primarily in Eastern Europe in the years going from Weisfeiler and Lehman's first paper (who first made the connection between 2-WL and this kind of algebra in 1968) to 1987, when Higman's paper [22] reconciled cellular (coherent) algebras with the parallel branch of research on coherent configurations developed in the west.
In general one could say that cellular algebras of order n are a sub-algebra of the matrix algebra $\mathrm{C}^{n\times n}$ that satisfies certain properties, but the subject is better explained in a more structured way.
A cellular algebra as defined in [23] is the embedding of a *cellular ring* W with basis $\mathcal{A}$ into the complex matrix sub-algebra generated by $\mathcal{A}$ itself; a cellular ring W is a ring over $n \times n$ integer matrices (where n is called the ring's degree) that satisfies the following properties (of which only the first 3 are independent):

1. W has a basis $\mathcal{A}$ of matrices whose only entries are 0 or 1
2. Given an element $A_i \in \mathcal{A}$, it's true that $A_i^T \in \mathcal{A}$
3. The sum of all the elements of $\mathcal{A}$ is $\mathbf{1}$, the matrix with all entries equal to 1.
4. For each $A_i \in \mathcal{A}$ there exists k_i s.t. every non-zero row in A_i has exactly k_i 1-entries.

The rank of W is given by the cardinality of $\mathcal{A}$. It's important to note that the 4 properties above define a unique basis up to permutation, so W is well defined.
From this definition, a coherent cellular algebra is one that is created from a coherent cellular ring, that is a cellular ring that has I_n in its basis; this is the kind of cellular algebra that has been extensively studied, and the one that interests us w.r.t. k-WL.
The reason why such an algebra was introduced and studied is because of its interesting relationship with graph isomorphism: given a set of matrices S with the same order n, it's possible to compute the smallest cellular algebra (the one with the smallest basis) that contains S; when talking about graphs, we can see S comprising only the adjacency matrix of some graph G, and this is where things get interesting: while the smallest cellular algebra (which is what 2-WL returns) does not always coincide with the centralizer algebra of G's automorphism group, two isomorphic graphs have the same smallest cellular algebras.
Let's see what this means. As explained in [3], a cellular algebra W obviously has

an automorphism group, but it's also possible to define the cellular super-algebra of such an automorphism over W's basis, that is the centralizer algebra of W's automorphism group, also called W's *Schurian closure*; we will refer to this super-algebra as S_W from now on. Now, it's possible that W coincides with S_W, and when this happens we say that W is a Schurian cellular algebra.
A cellular algebra's Schurian closure is relevant to us because it's relatively easy to derive from it the orbits of the automorphism group of a graph G, specifically from the Schurian closure of the smallest cellular algebra that contains the G's adjacency matrix.

Weisfeiler and Lehman in their paper conjectured that every cellular algebra was also Schurian, a claim later on proven wrong, so they believed that 2-WL was enough to compute not only the smallest cellular algebra containing a graph's adjacency matrix, but also its Schurian closure since they believed them to be equal, and subsequently the graph's automorphism group orbits.
We now know, as proved in [3], that each higher order of k-WL produces a super-algebra of the previous order in a sequence that ends when, for some value t of k, t-WL produces a Schurian cellular algebra and thus a result that allows us to directly deduce the orbits of the original graph's automorphism group.

The reason why one should interpret the results of k-WL as a cellular algebra instead of a simple partition that approximates the graph's orbits is that we can produce a regular representation of such an algebra, given by a $m \times m$ matrix (where m is the number of equivalence classes found by k-WL); a lot of interesting information is encoded in this regular form, such as the eigenvalues of the adjacency matrices in S and other interesting graph invariants, that are very useful to spectral graph theory research and graphs analysis.

2.6.2 Other invariants

As we just discovered, k-WL can give us eigenvalues of the adjacency matrix of graphs, and also compute and act based on other invariants when distinguishing graphs. What are these invariants is a question that was partially analysed in [24] for 2-WL, and it was discovered that 2-WL can recognise and use the number of 3-cliques (triangles) in graphs to distinguish them, but can't make use of 4-cliques of above; furthermore, it's shown that 2-WL exploits the number of *c-cycles* for c up to 6, that it cannot count 8-cycles, but it's left as an open question if it can always distinguish by number of 7-cycles.

Since this study was restricted to 2-WL, it would be interesting to analyse the combinatorial power of higher orders of WL, if the *s-cliques* recognised are constantly

only the trivial ones (e.g. k-WL using the number of at most $(k+1)$-cliques), and how the size of the exploited cycles grows with increasing values of k.

3
Algorithm description

Contents

3.1 Overview

Now that I've described all the prominent usages and applications for k-WL in current (and past) research, it's time to see how this very useful algorithm actually works.

I'll divide this chapter into three parts: in the first, I will explain briefly how the best currently known implementation of 1-WL works, to introduce to the workings of k-WL on its simplest version, then, in the second and third part, I will explain how two different implementations of the general k-WL work, both of which I have implemented during my final project.

A few additional notes on each of these parts are due:

1. I didn't actually implement this algorithm, even though it has better space and time efficiency than both the ones I actually implemented. The reason is simple: since this is an highly optimised version of the first algorithm presented by Weisfeiler and Lehman, it only implements 1-WL and can make use of some improvements a fully general version can't utilise, but due to the stringent time constraints implementing special cases for my general algorithm wasn't viable, so I focused on the version with a wider range of applications. This also means that this version will not be present in chapter 4, about implementation details, or in chapter 5, about tests and benchmarks.
   Nonetheless, make sure to read this part, since it also contains definitions and explanation of common properties that all the possible algorithms for k-WL must share.

2. This version is roughly described in [11] and one of its variants is called *the best currently known implementation of k-WL*. While this might be true for asymptotic time complexity (which was the context of the sentence in the paper, to be honest), this version has the major drawback of consuming a huge quantity of space to produce its result, and so ended up being rejected after the first working version and substituted with the next implementation in this list.

3. This second version utilises a different approach to compute the data needed to distinguish edges and refine colour classes during k-WL's rounds, based on coloured (k+1)-cliques induced by the current graph and its temporary colouring. This different approach makes it so that data can be computed on the fly and that the space usage, while it can theoretically get pretty high, grows with the number of colour classes in the output instead of with the number of nodes and edges in the graphs, reducing the memory needed for most of the applications that make use of the higher dimensional WL method. Conversely, it also introduces a limitation in that it can only directly return colourings of vertices and edges, though it is possible, with an additional computational effort, to compute the colourings of any k-tuple starting from the returned ones; still, there are not many applications that would require colourings of k-tuples, so the trade-off seemed acceptable to make.

Let's now get into how these algorithms work.

3.2 1-WL best algorithm

3.2.1 Overview and basic definitions

This algorithm is a simplified and optimised version of the first WL algorithm introduced by Weisfeiler and Lehman, and it's due to McKay [8], Cardon and Crochemore [25].
We will first go through the basic version introduced by McKay, and then present the optimisations that allow this algorithm to be run in quasi-linear time.
The whole idea at the base of the algorithm is to distinguish vertices based on their neighbours: while we cannot distinguish two vertices by their label, for sure if they have a different number of neighbours, then they are different, and there can be no automorphism between them. Continuing with this line of thought, suppose we have divided vertices in t different classes, simply based on the number of their neighbours, and so we know **for sure** that they are not in the same orbit: if we take two vertices that currently are in the same class (so we temporarily think they might end up being in the same orbit) and, checking their neighbours, we discover that the first has, for example, 2 neighbours from class 1 and 3 neighbours from class 2, while the second has 3 neighbours from class 2 and 2 from class 1, then we are sure that there can be no automorphism between this vertices, as there's no way we can swap them and keep the graph being the same.
This is exactly what the main loop of the algorithm does, it has a current partition of the vertices in colour classes and it tries to break each of them apart, one by one, separating the "non-automorphic" vertices in new, different colour classes by checking the number of neighbours they have from each currently known colour class (not the eventually newly computed ones, but the old ones it started with).
All of what was written until now is mostly true for any version of k-WL (minus the part in which it only works with vertices), and another thing all of the algorithms have in common is that each refinement round only depends on the topography of the graph and on the colour classes at the start of the round itself: if during a round the algorithm failed to split any colour class into new ones, there's no way another refinement round can be successful, since both the colour classes and the topography of the graph haven't changed. When this happens, the algorithm terminates and returns what is called a *stable colouring*, that is a colouring whose colour classes cannot be split any further by WL.
Another interesting property is the monotonicity of a refinement. Let's first define a way to compare colourings: given two colourings, one is said *coarser* than the other if the partition induced induced by the former is refined by the partition induced

by the latter. Monotonicity of the refinement is a simple property that is also fairly obvious to prove, that is take two colourings of the same graph C and D, where C is coarser than D and both of them are coarser than the actual orbits, and run a refinement round using first one and then the other, obtaining colourings C' and D'. Since the partition induced from D was a refinement of the one induced by C, it goes without saying that C' will be coarser than D' too, otherwise there would need to be some class in D' that is not a subset of a class in C': it cannot be that that class is a superset of some classes in C' because there was not enough information in C to split the classes better than a refinement on D could do, and it cannot be that the class is neither a superset nor a subset because both C' and D' still need to be refined by the colouring, and this means that refinement is monotonic. We say then that 1-WL returns the coarsest *1-stable* colouring, that is the coarsest colouring returnable when using only edges to distinguish vertices.

3.2.2 Algorithm details

To explain the algorithm, we will make use in fig. 3.1 of the pseudocode provided in [11], as it is a very clear and well thought description of it. Notice that this pseudocode implements the algorithm in $O(n^2 \log n)$, but improvements will be presented that allow it to run in quasilinear time. The general flow of the algorithm is pretty easy, we have a queue of colours on which to refine the graph and the range of colour classes to be refined at each round (represented by c_{min} and c_{max}), furthermore colours go from 1 and increase until we don't have to give any new colour out.

Instead of saving a "feature vector" for each vertex that stores the number of neighbours of each colour, this particular algorithm takes each colour q from the queue one at a time, and creates a partition of all the vertices in the graph, based on the equality of the pair $(C(v), D(v))$, where the first member is the current colour of the vertex and the second member is the number of its neighbours that have colour q; the current colour is included because vertices that are in different colour classes *must* remain in different colour classes after a refinement, even if they have the same number of q coloured vertices for some colour q.

Next, the algorithm takes each colour i currently in the graph (those between c_{min} and c_{max} that need to be refined) and takes the parts of the partition he just created whose elements have $C(v) = i$. This sub-partition for colour i will then induce new colouring on all its elements (every element in this sub-partition gets a new colour, and two vertices have the same colour iff they belong to the same part) and all the newly spawned colours will be added to the queue Q, *except for the one belonging*

COLOR-REFINEMENT(G)
Input: Graph $G = (V, E)$
Output: Coarsest stable coloring C of G

1. $C(v) \leftarrow 1$ for all $v \in V$ // initial coloring
2. $P(1) \leftarrow V$ // P associates with each color the vertices of this color
3. $c_{\min} \leftarrow 1$; $c_{\max} \leftarrow 1$ // color names are always between $c_{\min}$ and $c_{\max}$
4. initialise queue $Q \leftarrow \{1\}$ // Q contains colors that will be used for refinement
5. **while** $Q \neq \emptyset$ **do**
6. $q \leftarrow$ DEQUEUE(Q)
7. $D(v) \leftarrow |N_G(v) \cap P(q)|$ for all $v \in V$ // number of neighbors of v of color q
8. $(B_1, \ldots, B_k)$ ordered partition of vertices v sorted lexicographically by $(C(v), D(v))$
9. **for all** $i = c_{\min}$ to $c_{\max}$ **do**
10. let $k_1 \leq k_2$ such that $P(c) = B_{k_1} \cup \ldots \cup B_{k_2}$ // color class of c will be split // into classes $B_{k_1}, \ldots, B_{k_2}$
11. $i^* \leftarrow \arg\max_{k_1 \leq i \leq k_2} |B_i|$
12. $Q \leftarrow Q \cup \{c_{\max} + i \mid k_1 \leq i \leq k_2,\ i \neq i^*\}$ // add all colors except the one // with the largest class to queue Q
13. **end for**
14. $c_{\min} \leftarrow c_{\max} + 1$; $c_{\max} \leftarrow c_{\max} + k$ // new color range
15. **for** $b = c_{\min}$ to $c_{\max}$ **do**
16. $P(b) \leftarrow B_{b+1-c_{\min}}$ // new coloring
17. $C(v) \leftarrow b$ for all $v \in P(b)$
18. **end for**
19. **end while**
20. **return** C

Figure 3.1: Pseudocode from [11] describing an algorithm for 1-WL that runs in $O(n^2 \log n)$ time

to the largest colour class, we will later see why.
Finally, after all the colour classes have been refined on q, the range of colour classes is updated and the colours of the single vertices are updated.

One big improvement over a naive implementation of 1-WL (other than the switch from feature vectors to colour based refining) is the peculiar skip over the largest colour class of every refinement: this allows us to do one less refinement round over each partial colour class during the entire algorithm's run, reducing the running time considerably, even if the choice of skipping the largest colour class is completely arbitrary, since the only direct advantage in choosing the largest class is reducing the values of $D(v)$; of course, this also means that there are less possible equivalence classes and so smaller partitions, but its worth is questionable.
Let's take a colour class d that is split into the colour classes $d_1, \ldots, d_r$, and let's say w.l.g. d_1 is the largest. If we take another colour c that was already refined by d, we know that all the vertices coloured c have the same amount C of neighbours with colour d. If we now refine c by d_2, then d_3 and so on up to d_r, we will obtain the new colour classes $c_1, \ldots, c_m$, and the following will hold:

$$\forall i \in [1, m], \forall v \in c_i, \forall j \in [2, r] \quad |N(v) \cap P(d_j)| = k_{i,j}$$

where $N(v)$ is the set of neighbours of v, $P(x)$ is the set of vertices with colour x and $k_{i,j}$ is a constant determined by the colour class c_i and the colour class d_j. This means that two vertices in the same colour class c_i will always have the same number of neighbours with colour d_j for any j, but since they also had the same number of neighbours with colour d, then they all have the same number $N_d - \sum_{j=2}^{r} N_{d_j}$ of neighbours coloured d_1, which makes a refinement by d_1 useless.

The next improvement, allowing the algorithm to run in $O(m \log n)$, is simply a matter of removing useless iterations from the loop: given that we use a bucket sort to create the partition for each loop, computing the function $D(v)$ is our most expensive step, since it requires going through every vertex in the colour class q we are refining on and then visit each of its neighbours to increase a counter, for a total complexity of $O(n \cdot |P(q)|)$.
Since each colour class added to Q must be at most half the size of the colour class that generated it (otherwise it would be the largest one and would have never been added to Q), if a vertex appears in subsequently refined colour classes, it can appear only $\log n$ times before its colour class has cardinality 1, so each vertex can appear at most *logn* times and for each vertex appearance the algorithm spends $O(n)$ times, for a total of $O(n^2 \log n)$. What can be done to improve this complexity is noticing that if a vertex of the graph has 0 neighbours in $P(q)$, then its colour can simply stay unchanged. This means that there's no need consider at all the vertices not adjacent to a vertex in $P(q)$, reducing the time to compute $D(v)$ to $\sum_{u \in P(q)} d(u)$, where $d(u)$ is the degree of u. The bucket sort will also take the same amount of time. When we sum this over the number of appearances for each vertex in some $P(q)$, we get a running time of $O(n + m \log n)$, as we wanted.

3.3 k-WL First Algorithm

This algorithm, roughly described in [11], is an natural extension on the algorithm introduced by Weisfeiler and Lehman in their first paper, in that it takes the same general idea of compiling a feature vector for each vertex and then creating a finer partition of the currently computed partition of the vertex set (possibly the trivial one). We will now see exactly how it works, what are the differences from the original 1-WL algorithm and show some pseudocode that exemplifies it.

3.3.1 Pseudocode

Algorithm 1 k-WL First algorithm pseudocode

1: **function** K-WL(G, k) ▷ $G(V, E)$

Initialisation section

2: $AdjMat \leftarrow \textsc{Adj}(G)$
3: **for all** $t \in V^k$ **do**
4: $ATP(t) \leftarrow \textsc{computeATP}(t, AdjMat)$
5: **end for**
6: $E_1(x, y) \iff ATP(x) = ATP(y)$ ▷ Equivalence relation
7: $(C_1, \ldots, C_r) \leftarrow \textsc{OrdPartition}_{E_1}(V^k)$ ▷ Ordered partition induced by E_1
8: **for all** $t \in V^k$ **do**
9: $C(t) = i,$ s.t. $t \in C_i$
10: **end for**

Main loop

11: **loop**
12: **for all** $t \in V^k$ **do**
13: $S \leftarrow \textsc{computeFeatureSet}(t, C, V)$
14: **end for**
15: $E_2(x, y) \iff S(x) = S(y)$
16: $(D_1, \ldots, D_m) \leftarrow \textsc{OrdPartition}_{E_2}(V^k)$ ▷ Ordered partition induced by E_2
17: **if** $r \neq m \vee \exists i \in [1, r], C_i \neq D_i$ **then**
18: $C_i \leftarrow D_i, \forall i \in [1, m]$
19: **for all** $t \in V^k$ **do**
20: $C(t) = i,$ s.t. $t \in D_i$
21: **end for**
22: **else**
23: **break**
24: **end if**
25: **end loop**
26: **return** $(C_1, \ldots, C_r)$
27: **end function**

3.3.2 Explanation

The pseudocode in itself is quite lean and simple, since all the heavy-lifting has been moved into functions that are called from the main body. Some of these functions are straightforward, while others require a little explanation to fully understand how they work and why they are needed.

OrdPartition

This particular function is the conceptually easiest of the three that are present in the pseudocode above, as it just takes as parameters a set S and an equivalence relation E defined on it, and returns a partition of S whose elements are ordered. Now, what we mean by ordered is not immediately evident, and technically there's no need for the partition to be ordered for

the algorithm to work, but it's a convenient simplification. Since both our equivalence relations are based on equality of the images of the element for a particular function, the order we define is the natural order on the images, that is, given a partition $(C_1, \ldots, C_r)$ and an equivalence relation E based on equality of the images of a function f:

$$\forall i, j \in [1, m] : i < j, \forall a \in C_i, b \in C_j, f(a) < f(b)$$

Since sorting the set on the values of the images of its elements is actually a viable way to compute a partition, such a thing as requiring an ordered partition is acceptable.

computeATP

Since there's no natural way of defining an initial colouring for k-tuples of vertices in V, even having an initial colouring for V itself already, we colour each tuple based on its isomorphism type: given two tuples $v(v_1, \ldots, v_k)$ and $w(w_1, \ldots, w_k)$, v and w are said to have the same isomorphism type iff the mapping

$$M(v_i) = w_i, \forall i \in [1, k]$$

is an isomorphism. To do so, we compute for each tuple a matrix called the tuple's *atomic type* (ATP for short).

For a completely unlabelled graph, each entry $ATP_{i,j}$ is 2 if v_i and v_j are the same node, if that's not the case then it's 1 if there is an edge between v_i and v_j and 0 otherwise. If the graph is not unlabelled, we need to have vertex labels for entries indexed by the same vertex, and edge labels in the other cases, specifically $ATP_{i,j}$ will be equal to the negative label of v_i if v_i and v_j are the same vertex, to the label of (v_i, v_j) is such an edge exists in the graph, and 0 otherwise. Note that we required the entries for vertex labels to be negative, and this is possible because we can assume w.l.g. that all labels are positive integers starting from 1; since we're making this assumption, we also need to negate the vertex labels since otherwise we risk two different tuples resulting equal because a vertex label equals an edge label.

computeFeatureSet

Finally, this one is the core of the algorithm, and is the one that creates a vector of "features" that allows us to distinguish tuples. In 1-WL we created a vector with the number of vertex's neighbours of each colour, excluding the number of vertices of each colour that are not adjacent to the vertex from the

vector because they would be redundant, even if a complete feature vector should have included the latter too.
When dealing with k-tuples, we don't really have a natural definition of adjacency or non-adjacency, so we instead compile the vector with the colourings of all the tuples that have a Hamming distance of 1 w.r.t. the tuple we're building the feature set of. Specifically, for each possible k-tuple t, the feature vector will be composed by looping over all the vertices $v \in V$, and constructing a multi-set of colourings of the k-tuples obtained by substituting v for each element in t, then using the sorted multi-set of these multi-sets as t's fingerprint.

An note is needed on the fact that special consideration must be taken for the case $k = 1$ with this algorithm: as can clearly be seen trying out any example for this case, the multi-sets in the way we have described them can't encode the information about adjacencies of the vertices, so it's necessary for each vertex $u \in V$ to include in each of its inner multi-sets for a vertex $v \in V$ the ATP for the tuple (u, v).

Of course, the end result returned by the algorithm is peculiar, in that it's not a colouring of either the vertices or the edges of the original graph, but of k-tuples constructed on its vertex set, which is not very useful for high values of k.
There are several ways of dealing with this issue, but two are the easiest and most viable:

1. We will say that a k-tuple contains an edge if it contains its extremes at least once. For each possible edge in the original graph G, we can create a multi-set of the colourings of the k-tuples that contain it, and use that as a fingerprint to distinguish edges

2. Choose a subset S of the k-tuples such that there exists a bijection between S and V^2, and assign to each edge from V^2 the colour of its corresponding k-tuple based such a bijection (one example could be assigning to each edge $(u, v) \in V^2$ the k-tuple $(u, v, v, \dots)$, and such a method would work for any $k \geq 2$)

3.4 k-WL second algorithm

3.4.1 Overview

This second algorithm aims to reduce the average amount of memory needed to compute the stable colouring for a large graph, by avoiding the generation and

storage of n^k tuples, keeping track instead of only the colourings for each pair in V^2 and the fingerprints for each element of a colour class being refined, computing on the fly the fingerprints.
Question is, how could we compute on the fly the colourings for k-tuples we know nothing of? The answer is, we don't. We go backwards w.r.t. how the first algorithm worked, and start from the idea that an edge can be uniquely identified up to automorphism by the multi-set of the colourings of all the k-tuples that contain it. We then take this a step further, since we know that a k-tuple colouring can be identified by its ATP, we use those instead of the colouring, since we don't have any information about a single colour to assign them. Important note: it's obvious when analysing how the algorithm works, but it needs saying, the cardinality of the tuples needed to compute k-WL for a given k is $k+1$, as can be inferred by the fact that in 1-WL vertices are told apart based on the edges of a graph, in 2-WL based on the interaction each 2-tuple has with a third vertex, and so on and so forth.
However, we now have too much data to store: for each edge in a colour class, even using views of the original adjacency matrix, we at least need to store $O(n^{k-1})$ tuples of cardinality $k+1$; if we consider that at least one colour class (the trivial one of an unlabelled complete graph) is of size $n \cdot (n-1)$, we have a memory usage of $O(n^{k+1} \cdot (k+1))$, which is asymptotically higher than even the first implementation's.
The solution is avoiding storing useless k-tuples, avoiding repetitions, compressing the fingerprint: instead of storing each ATP, we can check which of them represent isomorphic graphs and only store representative ATPs with a counter for each isomorphism type, by computing a canonical form for each ATP and using a hashmap.
The optimisations do not end here, as the space needed can be further reduced, but the specifics will be left for chapter 4, where we will also analyse the resulting, real, time and space complexity. Now that we have explained the general idea, let's delve into the pseudocode describing the process.

3.4.2 Pseudocode

Algorithm 2 k-WL Second algorithm pseudocode

```
1: function K-WL(G, k)                                   ▷ G(V, E)
2:     AdjMat ← ADJ(G)
3:     ClassQueue ← INITIALISECOLORCLASSES(AdjMat)
4:     repeat
5:         numberOfClasses ← |ClassQueue|
6:         newClassQueue ← ∅
7:         repeat
```

```
 8:            ColorClass ← ClassQueue.pop()
 9:            for all (u, v) ∈ ColorClass do
10:                F((u, v)) ← createFingerprint(u, v, V, AdjMat, k + 1)
11:            end for
12:            E(x, y) ⟺ F(x) = F(y)          ▷ Equivalence relation on the fingerprint
13:            (C_1, ..., C_r) ← Partition_E(ColorClass)          ▷ Partition induced by E
14:            for all i ∈ [1, r] do
15:                newClassQueue.push(C_i)
16:                for all edge ∈ C_i do
17:                    C(edge) ← i
18:                end for
19:            end for
20:        until ClassQueue ≠ ∅
21:        ClassQueue ← newClassQueue
22:        updateMatrix(AdjMat, C)                                  ▷ Update the colours
23:    until numberOfClasses ≠ |ClassQueue|
24:    return ClassQueue.elements()
25: end function

26: function createFingerprint(u, v, V, AdjMat, m)
27:    for all (t_1, ..., t_{m-2} ∈ V^{m-1}) do
28:        ATP ← computeATP((u, v, t_1, ..., t_{m-2}), AdjMat)
29:        ATP ← individualiseEdge(ATP, u, v)
30:        vertSet ← canonicalFormVertexSet(ATP)
31:        Counter(vertSet) ← Counter(vertSet + 1)
32:    end for
33:    return sort({(s, c)|Counter(s) = c})
34: end function
```

3.4.3 Explanation

Of course, after explaining the idea behind it, the code is quite simple. The only lines that need some clarification are probably the lines 13, 25 and 26.
On line 13, we build a partition from the equivalence relation on the fingerprints, but this time without require it being ordered in any way, since there's no need to neither for checking termination nor equality of the partitions.
On line 25, we individualise the edge we are working on, that is we identify it by assigning to its entry in the ATP a unique label, since we want ATPs to be isomorphic up to the edge we are refining.
On line 26, we compute the canonical form of the ATP and at the same time compute its ordered vertex set. We don't have any constraints on how to accomplish this in the algorithm itself, and that's why the function is not listed anywhere, but left to be implementation defined.

We have discussed how the amount of memory used by the version of this second algorithm shown in pseudocode is lower than the amount used by the naive

implementation first described in this section, and since the latter was basically equal to the first algorithm's memory usage up to a factor $k+1$ which is going to be so small it could be considered a constant, we know that we have improved on space complexity; how much that is, and whether we traded off some time complexity for it is left to a more detailed analysis in the next chapter.

The only thing left to do now is prove that this approach actually works, that is that we are going to get a stable edge colouring whose induced partition is refined by the orbitals of the graph G, and that for some sufficiently large k we are going to actually recover the orbitals.

The former is fairly straightforward, as we will prove that if two edges get separated in two different colour classes, then they are not in the same orbit, or conversely that two edges in the same orbit are never separated in two different colour classes. This will suffice as a proof since by algorithm construction two edges in different classes can never end up in the same one in a later round.
The proof consists in a reductio ad absurdum, assuming that two edges e_1 and e_2 get separated but they are in the same orbit. If the latter is true, then they must have equal rows and columns in the adjacency matrix up to permutation, as that is an easily provable consequence of being in the same orbit. If they have equal rows and columns though, and the entry relative to the specific edge e_1 or e_2 in the $(k+1)$-cliques is individualised with an independent but equal label, then they must produce the same cliques, and thus could never be separated into two different classes.

The latter can be shown by observing that for any $k = m$, the second algorithm computes at least the same invariants as the first algorithm does for $k = m - 1$, since processing coloured $(m+1)$-cliques on x, y includes, and refines, counting the different coloured m-subgraphs on x and on y using their $(m-1)$-tuples. Since it's proven in [4] that a clearly equivalent version of the first algorithm eventually returns the correct orbitals of a graph for some large enough value of k, then for the reason just stated so does the second algorithm.
Keep in mind that this doesn't tell us anything about the comparative power of the two algorithms when called with the same value of k, just that they must be at least equally powerful when the second is called with a k value one higher than the first algorithm's.

4

Implementation details and complexity analysis

Contents

Note

In this section details and complexity of the actual implementations will be provided. Please note that no source code is provided either in this chapter or in appendix, since it will be provided as a compressed file through the online submission system. If the source code is by any means unavailable, please refer to the web page describing my project, `https://healty.github.io` and to the relative tickets on `https://trac.sagemath.org`, specifically numbers 25506, 25802, and 25891.

4.1 General implementation details

Starting from the basics, both algorithms have been implemented almost fully in *C++*, but since one of the aims of my final project was to produce an efficient algorithm that was well integrated in a full-fledged maths library, the algorithms had to be equipped with a caller method, written in *Cython*, that serves the purpose of translating *SageMath*'s graph representation in one suitable for use with the implemented algorithm, and also of formatting the latter's output in a way that the rest of the library could understand.
The wrapper between SageMath and my implementation is mostly the same for both algorithms, so it will be described only once here. Given the graph $G(V, E)$, a value for k, and an eventual partition p of V as parameters, it creates a copy G' of G with vertex set $V' = \{0, \dots, |V| - 1\}$ and an edge set $E' = \{(i, j) | i, j \in V' \wedge (v_i, v_j) \in E\}$, assigning to each edge in E' an integer label starting from 1 such that

$$label(i, j) = label(w, z) \iff label(v_i, v_j) = label(v_w, v_z), \quad \forall (i, j), (w, z) \in E'$$

Furthermore, the partition p is translated in an equivalent partition p' for V' in the obvious way.
After this normalisation process, the wrapper passes an adjacency list representation of G' to the main algorithm, together with a colouring of V' induced by p' and the desired value of k.
When a stable colouring is returned by k-WL is where the wrapper deviates for the two algorithms, since one must use the techniques explained in section 3.3.2 for the first algorithm to obtain orbits or orbitals of G', while the other naturally returns orbitals and, from those, orbits are easily deduced.
The wrapper then returns to being equal for both, as the only step is using an inverted mapping to translate the orbits/orbitals for G' into the corresponding result for G, and then return it.

4.2 k-WL first algorithm

4.2.1 Implementation details

This section will describe a few general implementation details for the corresponding algorithm presented in chapter 3, like data structure choices or output formatting, and will then go into the implementation of the three methods that were left unspecified in the pseudocode, with a particular focus on the ordPartition function.

In this particular implementation of the algorithm, to save on the space needed to index the feature multi-sets, the colourings and the ATPs with k-tuples, a mapping f is defined that assigns to each k-tuple an unique integer identifier which will be used to index the several needed data structures.
The whole implementation revolves around two vectors, one of integers (*remap*) and the other of booleans (*buckets*). The first of these vectors represents a ***reverse*** mapping g between the identifiers defined by f and the position their counterimage (the k-tuples) has in a list ordered by colour: $g(2)$ is the identifier of the second tuple in this colour sorted list, so $f^{-1}(g(2))$ is the tuples itself. The second, boolean, vector is instead a dynamic bitset whose bits are set to 1 to signal the beginning of a bucket: the first bit is always one; if a bit i is set and the next set bit is the j^{th} , this indicates that all the tuples $f^{-1}(g(k)), i \leq k < j$ are in the same bucket and thus have the same colour; of course the end of the vector signals the end of the last bucket. It's clear from the definition of these two vectors that they are enough to define a stable colouring, but not to compute it, since to compile the feature set we need to retrieve the specific colouring of a tuple, possibly in constant time, and this is why a third support vector that stores old colourings and feature multi-sets is also used.
After all these data structures have been initialised by the partition on the ATPs, for each refinement round a method named orderSetOfSetsBuckets is called. This method has been implemented to be as generic as possible, and accepts as outer sets to sort any data structure that has a method called getSetVector through a wrapper class called, in fact, Wrapper. It also allows for the possibility of computing the sets to sort on the fly by accepting two functors that build (or retrieve) each set and then destroy it, respectively. To sort the k-tuples on their feature multi-sets (which still haven't been constructed, mind) the method sorts bucket by bucket, and for each one accumulates its constructed elements in a temporary vector, to then call the method implementing ordPartition and use the resulting partition to update *remap* and *buckets* accordingly. This approach works because the k-tuples' colours (which are used while sorting the other buckets to build the multi-sets) are not updated at this point in the method, only the positions in the sorted list of tuples and the topography of that specific bucket are, and they are never used while sorting the other buckets. An addition is due on the fact that this only works due to how the colours are given out: since they are assigned based on the entries in *remap* at the end of the method, we can simply notice that if two tuples t_1, t_2 are, at any moment in time, in buckets i and j s.t. $i < j$, it will never happen that $C(t_2) \leq C(t_1)$; it's the same principle used in classic partition algorithms really:

once an element is moved in a bucket that spans from position i to position j, the tuple will never be in any position k such that $k < i \vee k > j$.

The fine details that were not present in the pseudocode have been explained, so all that's left to do is explaining how the three unspecified methods were implemented.

computeATP

To create the first colouring ATPs are computed by this method, creating a class that initially only holds their vertex set; when the content of the ATP is actually requested, an instance of the class VectorView is created, which is a class that takes a vertex set V and an adjacency matrix A and can be iterated on as if it was the adjacency matrix of the ATP induced by V on A, while never actually constructing such a matrix: each value, when requested, is retrieved on the fly from A at no extra cost.

computeFeatureSet

This method is pretty simple, as it consists in simply creating a vector of vectors *vec* and then, as described in chapter 3, going through each vertex v in V and substituting it for each element in the tuples it's working on: for each of these new tuples, their colouring is stored in a vector specific to v. Each of this sub-vectors is then added to *vec* and the latter is sorted. Finally, *vec* is flattened and saved as the tuples feature multi-set.

There's only one exception to this behaviour, and that is the case for $k = 1$. When this is the case, given an input 1-tuples on vertex v, for each inner vector on vertex u the 4 values of the adjacency matrix that represents the ATP for (u, v) are prepended to the vector before it's added to *vec*.

OrdPartition

To both sort the multisets (or ATPs) and divide the tuples into classes based on them, a hybrid sorting algorithm was implemented which used either a bucket counting sort or the standard C++ sort implementation depending on the size of the set to be sorted and on the range of values therein.

The choice of using a bucket counting sort was due to the fact that each multiset was really reduced to and ordered vector of integers, and that an ATP could also be considered a vector if we rearrange its adjacency matrix into one, thus every need of the algorithm was satisfied by sorting sets of fixed, equally long, array of integers.

The idea was then to first order the whole set of sets based on each set's first

value using counting sort, then divide the outer set into buckets where each bucket's elements had the same first value, and call the sorting procedure on each newly built bucket. At this point, the procedure will decide based on the range of values in the bucket and on its size if C++'s sort would be better than another counting sort or not, and sort the bucket accordingly on its second value; the same would then be done for the third value, the fourth, and so on, until a bucket ended up having only one item and was thus kept as is, to be later chained with the others to obtain the final sorted set of sets. Of course, as already stated, the results of this algorithm are given as a pair of vectors with the same format and usage of *remap* and *buckets*.

4.2.2 Complexity analysis

Space complexity

It's not very difficult to gauge the memory usage since most structures are initialised once and then never change their size, they are just re-utilised as needed. This is true for the *remap* and *buckets* vector, as they both have one entry per tuple in V^k and a size per entry of $O(1)$, occupying $O(n^k)$ space in total. It's also true in a way for the vector that represents the mapping f, as it too has n^k entries, but the entry space needed in this case is $O(k)$ since we have to store one k-tuple and one integer for each entry, making the total $O(k \cdot n^k)$. The adjacency matrix is also fixed at $O(n^2)$ space, and the ATPs occupy the space of their vertex set for each tuple, for a total of $O(k \cdot n^k)$. The only data structure that frequently varies in size is the vector that stores the colourings/multi-sets, since after computing the refinement of each colour class the computed multi-sets (that always have size $O(n \cdot k)$) are deleted since they're not needed anymore, to save space; even in this case though, we are lucky because we can calculate the worst case for the number of multi-sets needed at any one time, and that is n^k: it's not a very strict upper limit, since it's very difficult that all the tuples will have the same initial ATP and be in the same colour class, but we'll see why this does not affect our total space complexity.
So if we sum all the partial results we have, we obtain $O(k \cdot n^k) + O(k \cdot n^{k+1})$, where the first addend is the partial sum and the second is the space occupied by the colourings in our worst, worst case. It's clearly evident, though, that not all the tuples can be in the same colour class (save for the trivial case where $|V| = 1$), since at least the tuples of the type $(v, v, v, \dots)$ for some $v \in V$ must be in a colour class of their own. If we factor this in our calculations, we see that the absolute worst space complexity for the colourings is also $O(k \cdot n^k)$, so this is also our total space complexity for this algorithm's implementation, and it is a very strict upper limit, since the f vector has to take at least that space, in any possible case.

Time complexity

Creating the f mapping, since the implementation uses hashmaps to do it, takes time linear on the size of the mapping, that is $\mathrm{O}(n^k)$.
Initialising the colouring takes time $O(n^k)$ to write the colourings for each k-tuples, plus the amount of time needed to compute and sort the ATPs. Computing the ATPs takes time $O(k)$ for each tuple, that is $O(k \cdot n^k)$, since we only save a copy of their vertex set; calculating the amount of time needed to sort them, though, is a little trickier, but we can still approximate it by excess by noticing that OrdPartition is developed in a way to make it at most as costly as a bucket (counting) sort, but using C++'s sort when that would take too much time due to the range of integer values to sort. In general, we can say that since the algorithm has to sort at most k^2 buckets (since that's the size of the ATPs) each with at most n^k values, and the range of unique labels for vertices and edges is in the very worst conceivable case $[-n, n^2 - n]$, the total worst time for a complete bucket counting sort would be $O(k^2 \cdot n^2)$, and that is a very relaxed estimate.
After sorting the ATPs, compiling *remap* and *buckets* is linear in their size, that is $O(n^k)$.
Finally, we have the main loop of the algorithm to calculate the complexity of. Computing the feature multi-sets takes at most $O(k \cdot n \log n)$ since we have to retrieve $n \cdot k$ elements and order n sets of k elements each. Sorting the multi-sets will take $O(n \cdot k \cdot t)$ time using a bucket counting sort algorithm, where t is the number of elements in the colour class we are refining. This means that in a round computing the multi-sets will take a total of $O(k \cdot n^{k+1} \log n)$, while sorting them and partitioning will take $O(k \cdot n^{k+1})$, but since the $\log n$ factor in the first result could be eliminated by implementing a more efficient way of sorting each multi-set internally, we will drop it.
As stated in [4], there is a version of k-WL that would reduce the size of the *considered* colour class by at least half after each refinement. Such a version, while often mentioned, hasn't been found in any paper during my literature search, nor have I been able to devise it from scratch. For this reason this "naive" version was first developed, where the classes could be reduced by a constant amount of elements after each refinement and so the number of rounds could very well be linear in the size of the largest initial colour class, that is $O(n^{k-1})$ in the extreme worst case. Such a result compels me to state that the total time complexity is of $O(k \cdot n^{2k})$, but it's also important to remember that this only happens in limit cases and the complexity function seems to roughly follow $O(k \cdot n^{k+1+c})$, for some rather slowly increasing real parameter $0 < c \leq k - 1$, according to empirical analysis.

4.3 k-WL second algorithm

4.3.1 Implementation details

The specifics of this second algorithm are somewhat simpler to described, as there is only one function that wasn't described in the pseudocode, and that is canonicalFormVertexSet. Truth is, createFingerprint also received some tweaks w.r.t. the version described in chapter 3, but we'll get there.
The algorithm defines at the beginning two queues to hold the colour classes and two different adjacency matrices, but initialises just one set, which will be used as a reference for the refinement, while the other set is used to keep track of the new colour classes and colourings for the edges. Once the round is finished, the two sets are swapped in constant time thanks to the move and swap semantics of *C++11*.

The fingerprinting is also peculiar, since to save space each new isomorphism type discovered for the $(k+1)$-cliques is stored in a C++ *unordered_map* with a unique identifier; this mapping (let's call it f_map) is then used while computing fingerprints: during such a process the algorithm keeps count of which elements of f_map have been encountered and how many times, through a C++ map $fingerprint$ indexed on f_map values (not keys) and indexing the number of times each element has been encountered (N.B: elements never encountered are not in $fingerprint$, they don't have value 0); each time an ATP is produced, the algorithm checks if its canonical form is already in f_map (that is if we already encountered its isomorphism type), and if it is increases the corresponding counter in $fingerprint$ by 1, otherwise the canonical form is added to f_map with identifier $|f\_map|+1$ and $fingerprint[|f\_map|+1]$ is set to 1. At the end $fingerprint$ is returned and translated into an ordered *vector* in the obvious way so as to flatten it (ordered on the pairs, not on the single values).
This vector is called the fingerprint vector, and we will refer to it as $f_v(u,v)$ for each edge $(u,v) \in V^2$.
It's easy to see that all this indexing does not compromise the correctness of the algorithm: if two edges are supposed to generate the same ATPs, the first one to be processed will add all needed isomorphism types to f_map, and since the identifiers are fixed and can never be changed, it doesn't matter the order in which the second edge generates them, the sorted list of pairs $(identifier, counter)$ will stay the same. Conversely, if an edge adds a a new isomorphism type to f_map, then none of the previous edges could have been in its same colour class at the end of the round.

One last interesting detail is how the ATPs are generated. A VectorView almost identical to the one defined in section 4.2.1 is created for each generated ATP's canonical form's vertex set, it hash computed and cached. To avoid useless repetitions the algorithm generates vertex sets to compute the canonical form of in a way that avoids generating a permutation of an already processed vertex set: the first two elements of the vertex set are always the extremes of the edge whose fingerprint we are working on, while each of the other $k-1$ vertices is chosen so that it cannot be smaller than its predecessor; in this way each $k-1$ set of vertices generated is unique, the extremes of the edges are always included, and there's no waste of time due to processing permutations.

After computing all the fingerprints for a colour class, the time has come to split them into equivalence classes based on them. This can be achieved in two different ways, both of which have been tried and have been found out to be almost identical, with the first one being a little faster for small values of k (in the order of tenths of a second) but extremely more concise, and the second one being around 10% faster for values of k above 5 or 6, but quite clunky. The methods are the following:

1. Set up a C++ unordered_map the assigns a set of edges to a C++ vector of integers. This approach requires a hashing function of vectors of integer, which was taken from the open source implementation of boost::hash_combine [26] and is the same used to hash each VectorView, and is linear on the size of the colour class up to a factor needed to hash and compare (usually only once) each fingerprint vector.
   It also has the nice advantage of being able to retrieve the colour classes by simply iterating over the values of the unordered_map.

2. Utilise the same ordPartition function described in section 4.2.1 to order a vector of fingerprint vectors and then divide them into classes in the obvious way. This approach, while having different implementation constants, is also linear in the size of the colour class times the maximum length of a fingerprint vector.

The last piece of code to be described is the function used to compute canonical forms. The particular canonical form computed by the algorithm is the vertex set order that has a lexicographically minimal adjacency matrix.
The basic idea is trying every possible vertex set permutation and keeping track of the one with the minimal matrix, but this naive approach could take $O(k! \cdot k^2)$, which is a bit too much considering how many times this function will be called. A large

number of improvements and adjustments has been made to this implementation to allow a lower running time, but most of them, while reducing the runtime by a huge margin, complicate in an equally huge way the calculation of an asymptotic time complexity. Said improvements follow:

- Stop trying permutations on the second to last element; since the permutations are tried incrementally, that is starting from index 0, swapping element 0 with element 1 and recursing, then element 0 with 2 and recursing, and so on, with each recursion starting from an index increased by 1 (e.g. the first level of recursion starts on index 0, the second level on index 1, etc.), it makes no sense to recurse further once the starting index is on the second to last element, simply switch the last two vertices, check that permutation and then return.

- When comparing the current permutation with the best one currently found (that is, when comparing their adjacency matrices), interrupt as soon as difference is found and return different values depending on where the first difference is located; each of the following considerations is made assuming that the current permutation was found lexicographically bigger than the current best one; if the different entry has both row and column indices smaller than the starting index of the current recursion level, then there's no way that any other permutation tried in this branch of recursion will be better than the best one found until now, so we can backtrack; if instead the different entry has row or column index equal to the starting index, while the other index is either equal or smaller, then the current swap choice is not optimal, and there's no need to progress further into this recursion branch, but no need to backtrack either, so we can just go to this branch's next sibling; if none of this conditions holds, we can't say anything on the goodness of the recursion branch, so we have to keep visiting it;

- Last improvement, both in this list and chronologically, was finding and memorising a graph's automorphisms while looking for its canonical form; before swapping two vertices in the vertex set, the algorithm checks if they belong to the same orbit in the clique's automorphism group by comparing their rows and columns in its adjacency matrix; if they do, there's clearly no use in swapping them, since the adjacency matrix would stay the same, if they don't then try and swap them; of course, in both cases the result is cached in a $n \times n$ matrix.

- The cache matrix from the point above is static, in that there's no change in its size nor in its usage. It's possible to allocate it once at the beginning of the algorithm, zero it out, use it to compute a canonical form and then reset it, by only zeroing the at most $(k+1)^2$ used entries instead of the entire matrix though. This allows us to have the advantage of caching the vertices that are in the same orbit, without having a fixed cost of $O(n^2)$ for each call to the function.

4.3.2 Complexity analysis

Space complexity

$O(n^2)$ space is needed to store the two adjacency matrices, the two colour classes and the caching matrix described above. During a round of refinement, *f_map* is constructed, so we will need to account for that, and then during the refinement of each colour class $O(N_c \cdot s + n^2)$ space is needed to store the data structure needed to compute the colour classes, where N_c is the number of new colour classes that are going to be generated, s is the maximum size of a fingerprint and is the same as the maximum number of elements in *f_map* during the colour class refinement, and the factor $O(n^2)$ is due to the fact that a new copy of the edges will need to be stored in the data structure.
Finally, to get a total figure we need to calculate what could be the maximum number of elements for *f_map*. We know from chapter 3 that, without aggregating by isomorphism type or adjusting for permutations, we would have n^{k-1} different vertex sets for each orbital in the worst case. Since we adjust for permutations, the number drops down to

$$\binom{n+k-2}{k-1} = \frac{(n+k-2)!}{(n-1)! \cdot (k-1)!}$$

which is also

$$O\left(\frac{n^{k-1}}{(k-1)!}\right)$$

Of course we now have to think about the number of times this amount could differ or be equal for each edge. As we stated, the number depends on the number of orbitals, since each pair of edges in an orbital is going to have the same exact set of cliques built on each of them. Now if each edge is in a different orbital, then the worst possible space complexity for *f_map* would be the last figure shown above times the number of edges and the size of a vertex set, that is $O\left(k \cdot \frac{n^{k+1}}{(k-1)!}\right)$, which

is only a factor of $O((k-1)!)$ away from the first algorithm's space complexity. Truth is that such a case is extremely unlikely, as a lot of the k-cliques we create, especially for values of k that are less then 10, end up being isomorphic to each other, and almost never we're going to see a graph with n^2 orbitals, especially for highly regular graphs. Still, calculating an average case complexity taking into account cliques' isomorphisms is practically impossible, so we will have to rely on empiric data to show us proof that the memory consumption of this second algorithm is much lower than the first one's and content ourselves with a not very strict final worst-case space complexity of

$$O\left(k \cdot N_{orbitals} \cdot \left(\frac{n^{k-1}}{(k-1)!}\right)\right)$$

Time complexity

Since indexing the fingerprints is done in linear time in the size of the fingerprint times the number of them, and we already established in the previous section that such a figure is do minated by the number of generated vertex sets, the time complexity entirely depends on the graph generating part.
We don't have to deal with isomorphism or anything like that, since we still need to generate graphs even if they are isomorphic to some other already processed graph.

A note is in order, as we are about to approximate by excess the number of permutations that we need to try to find the canonical order for the vertex set of a graph: the reason for that is again the fact that search space pruning (done for example by avoiding swapping vertices in the same clique's automorphism's orbit or cutting dead recursion branches and backtracking) reduces by huge margins the time needed, but it doesn't offer any guarantee on the time reduction factor as it depends on the structure of the graph and the order the permutations are traversed in; it's worth then to keep in mind that the figures presented are for limit worst cases and that, empirically, the algorithm seems to generate each isomorphism type's vertex set only $O(1)$ times.

As shown in the previous section, each edge needs generating at most $O\left(\frac{n^{k-1}}{(k-1)!}\right)$ vertex sets, but then for each of them a canonical order is computed in (remember, extremely worst case) $O(k^2 \cdot (k+1)!)$ time. This means that each edge takes $O\left(k^4 \cdot n^{k-1}\right)$ to have its fingerprint computed, so our extreme worst case running time is that multiplied by the number of edges and then by the number of rounds. For the same considerations applied to the first algorithm, the number of rounds

could very well be $O(n^2)$, and such is also the number of edges, so the very worst case time complexity is

$$O\left(k^4 \cdot n^{k+3}\right)$$

which is only a

$$O\left(k^2 \cdot \frac{n^2}{\log n}\right)$$

factor over the best k-WL implementation known in exchange for improved space complexity over it as detailed in the relative section, and this is using strict upper bounds for the standard k-WL implementations and quite relaxed ones (at least on average) for the current one.
Furthermore, it's extremely more efficient than the first implementation as can be seen by comparing the two final time complexities.

4.3.3 Multi-threaded implementation

Almost at the end of my project I happened to read Martin Furer's slides for his talk on *"The power of WL[k]"* in the Conference in Algebraic Graph Theory *"Simmetry vs Regularity"*. During his presentation he mentioned the fact that almost all implementations of k-WL (especially those that don't rely on the "process the smaller half" approach) are highly parallelizable, as the processing of each k-tuple (or edge, in our specific case) is usually quite costly but independent from the rest of the workload, save for a few common data structures; at the same time, Furer wondered what performance improvement could be achieved using multi-threading, both theoretically and in practice.
Due to time constraints, I didn't have the time to produce a full complexity analysis of this multi-threaded implementation which, admittedly, was initially developed in just a few hours as a simple exercise in style, without any intent to include it either in SageMath or in this dissertation. Due to the impressive improvement in computation speed during a series of tests and the fact that it was an apparently novel approach to k-WL, this implementation made its way into the final version of the project.

There's nothing too complex in the code itself, as the parallelisation simply splits the edges in each colour class in p chunks, where p is the number of threads to be used, and creates multiple copies of the caching matrix and the support vector used during fingerprint generation, supplying each set to a different thread. Still, some form of synchronisation was needed because, to avoid a huge spike in memory and runtime, the isomorphism types database and the fingerprint indexed map were

kept common among all threads; while for the latter a simple C++ exclusive_lock proved sufficient, for the former there was the issue that each thread constantly needed to access it to read isomorphism types' identifiers, while sporadically adding a new type itself; this meant that a writer/reader lock was needed, specifically based on C++14's shared_timed_mutex, which could allow threads to access the data structure at the same time for reading purposes, while also allowing them to lock it exclusively when inserting new elements. Sadly, no implementation of a lock-free or even thread-safe hashmap is provided in any current version of the C++ Standard Library, so it was not possible to implement a finer locking strategy.
In the end, as we will see in the next chapter, this peculiar version of k-WL proved to be 2 to 4 times faster than its sequential mate, depending on the size of the graph.

5 Tests and benchmarks

5.1 Test suite

As mentioned in section 2.2.1, there are some specific families of graphs that interact in a particular way with k-WL. In order to properly test both my implementations and benchmark them (fundamentally between them, since as we'll see there is only one other publicly available implementation of WL I have had access to, and it only computes 2-WL) I then decided to add to SageMath's library a few helper functions that could be used both to ease my testing and to favour repeatability of the experiments we are about to run.
Specifically, I've developed functions to generate members of the Cai-Furer-Immerman, Egawa and Hamming families of graphs, though we will make proper use only of the first two, and also a method that can check the stable colouring returned by k-WL for correctness and an interface that allows SageMath to test isomorphism and compute automorphisms in graphs through nauty [8].
In this section I'll present the results of the tests for my first and second k-WL implementations (both the sequential and parallel versions) but, before that, I will provide a short description of the support functions I created; since, however, the generators simply implement in pure python the construction detailed in the graphs' original papers, we will focus our attention on the correctness checker and on the interface with nauty.

5.1.1 Correctness test

This tool was needed in order to check that the results provided by my implementations of k-WL were actually either the orbits/orbitals or a partition by them refined.

In particular, the function was needed to test the result when dealing with orbitals, since in SageMath the orbits are easily retrievable with a method run on the graph one's interested in.
In order to produce orbitals that were surely correct (up to bugs in the code), I followed the theory detailed in [27] and implemented the algorithm presented in the following pseudocode.

Algorithm 3 1-WL Second algorithm pseudocode

```
1: function GenerateOrbitals(G)                                   ▷ G(V, E)
2:     representers ← ∅
3:     orbitals ← ∅
4:     f : representers → orbitals
5:     for all (x, y) ∈ V² do
6:         representers ← representers ∪ {(x, y)}
7:         for all (u.v) ∈ representers do
8:             if ∃g, h ∈ Aut(G) : g(x) = u ∧ g(y) = h(v) then
9:                 representers ← representers \ (x, y)
10:                f((u, v)) ← f((u, v)) ∪ (x, y)
11:            end if
12:        end for
13:        if (x, y) ∈ representers then
14:            new_orbital ← ∅
15:            orbitals ← orbitals ∪ new_orbital
16:            f((x, y)) = new_orbital
17:        end if
18:    end for
19: end function
```

In summary, what happens is that each possible pair of vertices is tested against a list of currently known orbital representatives, if none of them is in the same orbital as the current edge e, a new orbital is added and e made its representative.
The condition that tells us if two edges are in the same orbital is very simple, and due to the link between orbitals of G and the orbits of point stabilizers of its automorphism group: given two edges (x, y) and (u, v), if and only if there is an automorphism in G that maps x to u, and that same automorphism maps y to a vertex that is in the same orbit of G's automorphism group as v, then the two edges are in the same orbital.

After having computed the orbitals, the test compares the orbitals and k-WL results. If they are equal, it returns a *Correct* response.
If they are not, it checks if each orbit is a subset of a single colour class returned by k-WL. If this is true, then it means that the orbits refine k-WL's result and the response of the test is *Refinable*, otherwise the method outputs *Wrong*.

A second correctness test is provided but not used in these tests because the code is quite complex in dealing with special graphs, such as multi-edged labelled

ones, and would only introduce an ulterior possible point of failure, even if the algorithm itself is faster than the one described above. In a few words, this other test simply computes the line graph $\mathcal{L}_G$ of the edge-labelled union of the original graph and its complement, and computes its orbits in order to obtain the orbitals of G (without the orbits though, that will have to be added separately).

5.1.2 Nauty interface

My work was heavily based on *pynauty*, a *GPLv3* python module that allows python code to interface with nauty and use it for computing automorphism groups and testing for isomorphism.
While keeping the same name to give credit, the package I developed and included in Sage applies a large number of modifications to pynauty's source, basically keeping only the C helper functions used to call on nauty.
After discarding the Graph building part included in pynauty since it was not needed for Sage and modifying the python code so that a SageMath's graph could be passed to it, I then modified the C code to make it parse such a graph and directly convert it into a representation useful for nauty, without any wasteful middle transformations. Finally, I made it so that the returned automorphism groups were displayed in a format that was coherent with the one used by the current Sage's automorphism_group method; of course, the isomorphism checking part didn't need any output adjustments, since it returned a simple boolean.

My work on the interface, anyway, didn't finish here, since nauty's functions, while very fast, in this form were basically useless for Sage, since they required in input a (di)graph whose edges were unlabelled and without multiple edges, a very strong restriction for a generic graph library.
The only way I could find around this issue was described in section 14 of nauty's documentation, which can be found on the library's website, to which I added a quick fix to allow for multiple edges:

1. If the graph is a multigraph, the multiple edges are removed and their multiplicity is translated to a label: in a number if the original edge had no label, in a pair with the original label as the first member and the multiplicity as the second otherwise; of course, this means that multiple edges on the same extremes should at least have the same labels.

2. If the graph is edge labelled (or if it was a multigraph), convert the labels into m consecutive integers starting from 1, where m is the number of unique edge labels in the graph; if we're dealing with an isomorphism checking problem,

then this step must be done on the disjoint union of the two graphs being checked, that is if two edges in the two graphs have the same label, the edges must end up with the same integer label.

3. Finally, each of the n vertices is converted into a (strongly) connected component of $\log m$ vertices (for a total of $n \log m$ vertices) and the edges between the new sets of nodes are determined by the binary representation of the original edge labels; e.g: if between a vertex a and b there was an edge with label 5, the new graph will have edges (a_0, b_0), (a_2, b_2), since the number 5 has only its first and third bits set.

4. Now, the only thing needed is restricting the action of the computed automorphism group to all of the vertices of the form v_0, while respecting the eventual original partition provided to the method; this gives us the automorphism group of the edge labelled (multi)graph.

5.2 Test results

We will now present the results, divided by test run. Each test case will have a code snippet usable to replicate the experiment, followed by a summary of the results for each implementation. The listing of the test outputs will be provided at the end of this document.

5.2.1 Egawa graphs

As mentioned in chapter 2, Egawa graphs have orbitals that require a higher order k-WL to be fully discovered. In a particular case, when we have an Egawa graph of the type $I(p, 0)$, 3-WL **must** be able to identify all the orbitals of its argument. We are going to use the first two Egawa graphs of this type in order to test correctness, combinatorial power and also to benchmark the implementations, as the second Egawa graph is pretty massive with its 256 vertices and a tight knitted set of edges.

Egawa graph with parameters $(1, 0)$

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(1,0) #Isomorphic to the Shrikhande graph
sage: res = WL(G, 1, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 1)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 2)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 3)
```

```
sage: coc(res, G, cardinality=2)
sage: sage: res = WL(G, 4)
sage: coc(res, G, cardinality=2)
sage: sage: res = WL(G, 5)
sage: coc(res, G, cardinality=2)
sage: sage: res = WL(G, 6)
sage: coc(res, G, cardinality=2)
```

1st algorithm

Output *Correct*, *Not tested*, *Refinable*, *Correct*, *Correct*, *Correct*, *Not tested*

Time $440\mu s, None, 2.17ms, 59.2ms, 1.79s, 1m29s, None$

2nd algorithm

Output *Correct*, *Refinable*, *Refinable*, *Correct*, *Correct*, *Correct*, *Correct*

Time $3.31ms, 1.5ms, 2.77ms, 42.3ms, 486s, 3.8s, 26.3s$

2nd algorithm - multithreaded version

Output *Correct*, *Refinable*, *Refinable*, *Correct*, *Correct*, *Correct*, *Correct*

Time $10.3ms, 5.72ms, 3.8ms, 56.3ms, 333ms, 2.43s, 10.2s$

Summary

As we can see all the algorithm report the results we would expect. Due to the fact that for k=1 the first algorithm works on single vertices, it was impossible to generate orbitals and that's why that test case was skipped. The last test case was skipped again for algorithm 1 because given the runtime of the previous test case, it would have taken more than 50 minutes without giving us any interesting information.

It's interesting to see a trend that we will often spot for what regards the multithreaded version of the second algorithm, that is the inconsistent time taken to run small test cases, either with low values of k or n, since at that point the slight synchronisation overhead dominates and messes up our results.

Egawa graph with parameters $(2, 0)$

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(2,0)
sage: res = WL(G, 1, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 1)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 2)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 3)
sage: coc(res, G, cardinality=2)
```

1st algorithm

Output *Correct*, *Not tested*, *Refinable*, *Out of memory*

Time $19.6ms, None, 15.9s, > 60m(timeout)$

2nd algorithm

Output *Correct*, *Refinable*, *Refinable*, *Correct*

Time $1.43s, 1.44s, 12.5s, 58m39s$

2nd algorithm - multithreaded version

Output *Correct*, *Refinable*, *Refinable*, *Correct*

Time $1.45s, 1.4ms, 4.72s, 10m8s$

Summary

Here we can see the real improvements that the two versions of the second algorithm bring to the table: with 256 vertices, the first implementation can't run 3-WL on the graph without going over an hour of time and running out of memory. The second algorithm too takes almost a full hour though, and here the parallel version shines, taking only 10 minutes, almost one sixth of its sequential variant's time.

To put into perspective what I mean by the first algorithm using too much memory, on this example only I'll list the peak memory consumption of each implementation, but the ranking established for this test is the same for any other, when scaled down to size accordingly:

First algorithm's memory peak

10 Gigabytes before it was shut down, predicted peak usage of about 24 Gigabytes

Second algorithm's memory peak

6.7 Megabytes

Parallel second algorithm's memory peak

16.74 Megabytes

The list speaks for itself, but for the sake of transparency, I want to specify that the predicted peak usage was calculated by running the first algorithm's version of 3-WL on the Egawa graph(1,0), writing down the peak memory usage of 2.1 Gigabytes and then computing how much memory should have been used by substituting the correct values into the space complexity formula, so as to

obtain the relative constant factor between theoretical analysis and practice. At this point, the same space complexity formula was used to calculate the theoretical number of Gigabytes used and that figure was multiplied by the factor to obtain about 24 Gigabytes total.

5.2.2 Planar graphs

Overview

As proved in [12], any planar graph can be recognised with at most 3-WL. To somewhat test this, and thus also test the implementations' correctness, we take one planar graph with n vertices for each n between 1 and 64 and try to compute its orbitals with 3-WL. We expect all of them to be recognised, without any particular space or time concerns.
Since there are so many results, the times will be condensed in a line graph and the complete output left to the listings at the end of the document. The test code snippet is shown below, while the plot is in fig. 5.1.

Code

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: for i in range(1,65):
....:     G = graphs.planar_graphs(i).next()
....:     print("**********\%i**********" \% i)
....:     res = WL(G,3)
....:     coc(res, G, cardinality=2)
```

Summary

While the first and second implementations behaved as predicted, with the former being only slightly slower than the latter for values of n and k so small, the real surprise is the multithreaded version that seems to not be able to parallelise anything when run on planar graphs, and fares worse than its own sequential variant. Further inspection is due on why this is.
Nonetheless, all three algorithms correctly recognised the orbitals with 3-WL, confirming the results of the paper cited above.

5.2.3 Cai-Furer-Immerman graphs

This is a very important series of tests for k-WL, as Cai-Furer-Immerman graphs are regarded as one of the toughest edge cases for this algorithm [7] [28] and, admittedly, were devised as such.

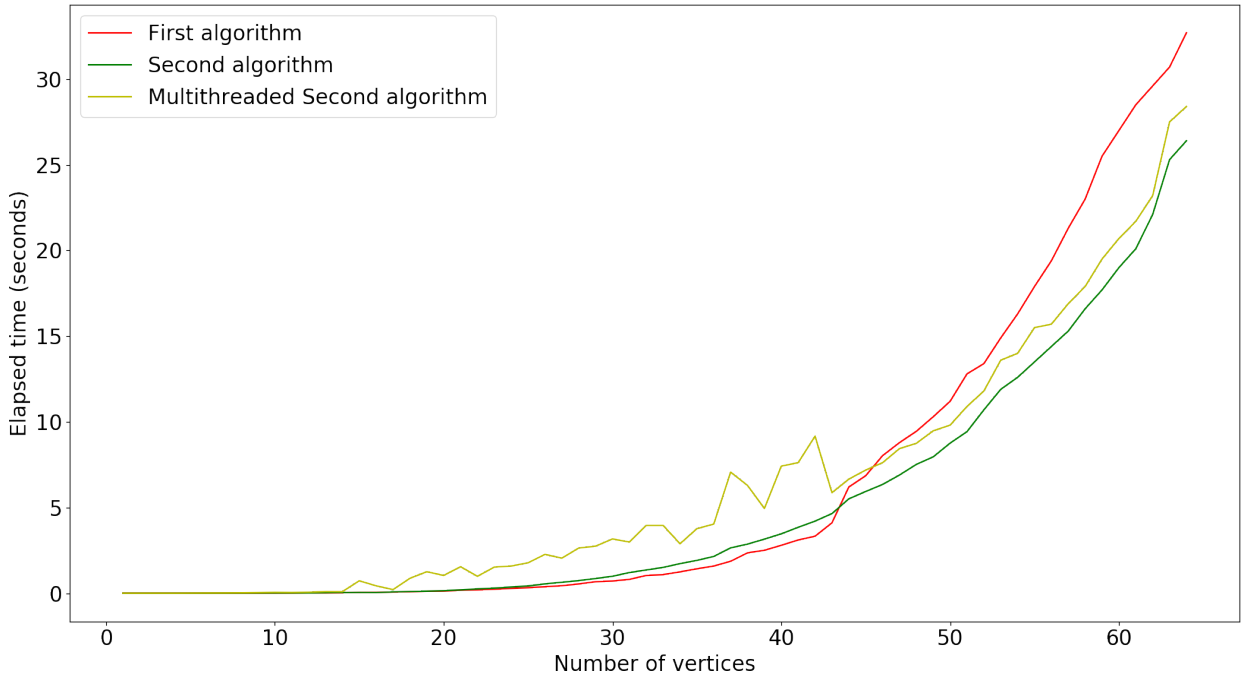

Figure 5.1: Plot of the time each implementation of k-WL took to compute the orbitals of a planar graph of order n

These tests will be both a check for correctness and efficiency, since other than having a specific value of k they need to be recognised, they are also computationally very expensive to process [29].

Cai-Furer-Immerman graph of order 2

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.CycleGraph(4)
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: res = WL(Final, 1, partition=final_p, result="vertex_classes")
sage: coc(res, Final, partition=final_p, cardinality=1)
sage: res = WL(Final, 2, partition=final_p, result="vertex_classes")
sage: coc(res, Final, partition=final_p, cardinality=1)
```

1st algorithm

Output *Refinable*, *Correct*

Time $2.31ms, 137ms$

2nd algorithm

Output *Refinable*, *Correct*

Time $46.4ms, 159ms$

2nd algorithm - multithreaded version

Output *Refinable*, *Correct*

Time $85.3ms, 612ms$

Summary

Nothing of note, really. The main points to take from these tests is that the first implementation confirms itself to be the best for small values of n and k, and for those same kind of values the parallel version's results simply fluctuate all over the place.

Cai-Furer-Immerman graph of order 3

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = Graph()
sage: G.add_edges([(0,1),(0,2),(0,3),(1,4),(2,4),(3,4),(1,2),(2,3)])
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: res = WL(Final, 1, partition=final_p, result="vertex_classes")
sage: coc(res, Final, partition=final_p, cardinality=1)
sage: res = WL(Final, 2, partition=final_p, result="vertex_classes")
sage: coc(res, Final, partition=final_p, cardinality=1)
sage: res = WL(Final, 3, partition=final_p, result="vertex_classes")
sage: coc(res, Final, partition=final_p, cardinality=1)
```

1st algorithm

Output *Refinable*, *Refinable*, *Correct*

Time $12ms, 1.23s, 10m25s$

2nd algorithm

Output *Refinable*, *Refinable*, *Refinable*

Time $521ms, 1.43s, 2m4s$

2nd algorithm - multithreaded version

Output *Refinable*, *Refinable*, *Refinable*

Time $469ms, 2.1s, 1m11s$

Summary

This, contrary to the previous one, is an incredibly important result. While the timings do nothing more than confirm the by now established performance rankings, the outputs of the tests are of uttermost relevance: both the implementations derived by the space saving algorithm failed to recognise the Cai-Furer-Immerman graph for $k = 3$, when they clearly should have, as the first implementation's results confirmed.

To test how much of an issue this was, an additional test has been performed only on the single-threaded version of the new implementation in which it tries to distinguish H and Ht using $k = 4$. After 4 hours and 13 minutes, having used a peak of 8 Gigabytes of memory, the algorithm was successful in distinguishing them. We will see in chapter 6 what this entails.

5.2.4 Paley digraphs

This family of tests, while run on a relatively small graph, will allow us to both test how the implementations cope with increasing values of k (as a graph this small will allow us to get to a relatively high value), without however providing them with a trivial subject to process. Furthermore, in [14] the graphs are only proved to not be recognised by $k = 2$, so we will try to gather some empirical evidence of which value of k might be enough to recognise at least some of them.

Paley digraph of order 4

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: n = 4
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
....: from sage.matrix.constructor import identity_matrix, matrix
....: H = skew_hadamard_matrix(4*n)
....: M = H[1:].T[1:] - identity_matrix(4*n-1)
sage: nmap = {0:0,1:1,-1:0}
sage: M = M.apply_map(lambda x: nmap[x])
sage: G = DiGraph(M, format='adjacency_matrix')
sage: res = WL(G, 1, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 2, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 3, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 4, result="vertex_classes")
sage: coc(res, G, cardinality=1)
sage: res = WL(G, 2)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 3)
sage: coc(res, G, cardinality=2)
```

```
sage: res = WL(G, 4)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 5)
sage: coc(res, G, cardinality=2)
sage: res = WL(G, 6)
sage: coc(res, G, cardinality=2)
```

1st algorithm

Output *Refinable*, *Refinable*, *Correct*, *Correct*, *Refinable*, *Refinable*, *Correct*, *Correct*, *Correct*, *Correct*

Time $2.06ms, 3.28ms, 68.6ms, 1.68s, 2.41ms, 66.9ms, 1.56s, 1m23s, 35m42s$

2nd algorithm

Output *Refinable*, *Refinable*, *Refinable*, *Correct*, *Refinable*, *Refinable*, *Refinable*, *Correct*, *Correct*, *Correct*

Time $2.89ms, 2.33ms, 19.4ms, 527ms, 2.65ms, 18.6ms, 518ms, 2.9s, 22.3s$

2nd algorithm - multithreaded version

Output *Refinable*, *Refinable*, *Refinable*, *Correct*, *Refinable*, *Refinable*, *Refinable*, *Correct*, *Correct*, *Correct*

Time $2.31ms, 2.48ms, 8.2ms, 347ms, 2.34ms, 8.7ms, 349ms, 1.55s, 13.2s$

Summary

Again, we find ourselves in a situation in which the first algorithm correctly recognises a graph with $k = 3$ while the other two implementations are unable to. However, this time we can see that those other two are actually able to recognise it with only $k = 4$, and this might suggest that the same could happen with the Cai-Furer-Immerman if we tried.

Putting aside the recognising power, this tests also tell us that the two implementations derived from the second algorithm scale *a lot* better with increasing values of k.

6 Conclusions

6.1 Summary of the work

During my final project I have collaborated with developers from SageMath in order to implement the k-WL algorithm as detailed in chapter 1. The final result comprises a series of testing tools for automorphism/isomorphism related results, in particular an interface to nauty [8], a tool currently regarded as the fastest canonical form computing software in the field, a series of test case/graph generators that provide examples with interesting k-WL related properties, and two fast algorithms that check with different approaches if a given partition of a graph's vertices/edges is equal to its orbits/orbitals, or could become so with some refining; other than the testing tools, the main part of the project has consisted in developing three different implementations of k-WL, although with two of them being derived from the same algorithm, each one with strengths and weaknesses that were tested, analysed and commented on during chapter 4 and chapter 5, and that make them suited to different tasks, while implementing the same hierarchy of algorithms.
Each one of this pieces of work is also accompanied by documentation, both internal and external, unit tests and examples that ensure their correct functioning. All of this was implemented, as noted above, in an extensive and powerful open source mathematical software/library, that provided the backbone in which my code is installed. This will allow future users to experiment freely with k-WL and possibly try applying it to different branches of mathematics, potentially discovering new applications for it that will go together with the ones described in chapter 2.

Initially the aim of implementing a good, efficient version of k-WL seemed achieved with the first implementation already, but further testing proved that

while quite time efficient for an algorithm with a high-degree polynomial complexity, the memory requirements of the implementation were a major drawback that would preclude the chance of working with large graphs in any capacity: while time inefficiency can be accounted for (up to a certain limit) by simply waiting more time for the solution to be returned, a poor memory efficiency causes the program to crash and not be able to produce any output at all.
While thinking about ways to reduce the memory needs of my code, I realised that, for how much I could lower it, due to the fact that the algorithm deals with and has to store the colourings of k-tuples, the necessary space could never be less than n^k. This meant that my implementation, while surely improvable in terms of space, was not completely at fault in that regard, and that an algorithm different from those commonly presented in literature had to be devised in order to obtain asymptotic complexity improvements.
This brings us to the second implementation of k-WL I implemented for my project, that used k-cliques built on the fly on top of each possible pair of vertices in order to compute the orbitals of a graph. This proved to be a novel approach, not really tried or reviewed in literature and also fast and memory light, as soon as I fixed a few parts of the code that wasted time for no reason, but against all these positive properties, a major downside stands: we've just witnessed through the tests in chapter 5 that this implementation, while meeting every parameter I set for myself at the beginning of my work, and agreeing with a standard k-WL implementation (the first implementation) on a *huge* number of examples, has trouble recognising and distinguishing some graphs that standard k-WL shouldn't have any issues for. While it's true that some test cases were failed for my second implementation, I want to focus my final thoughts on two important points:

The cause Where do the discrepancies between the behaviours of the two algorithms come from? Given the spurious nature of the errors, one would tend to believe it to be a bug in the code that causes the second implementation not to return the correct result that it should, but it's also true that when k is increased, the faulty program rectifies its answer, does its job and produces the orbits of the graph, as requested. This suggests that the root cause might be located in the algorithm itself, that appears to behave like k-WL and does so most of the time, but is slightly weaker in recognising power.

The history Has it ever happened before? Were there any other algorithms that seemed to be equivalent to k-WL but proved to be weaker? The answer is yes, as Cai mentions in [7] an algorithm devised by the authors of the original WL

paper that was initially thought to be more powerful than k-WL and able to solve the isomorphism problem in polynomial time, to then fail miserably as any other attempt in the current history of research in the field. The way it failed is relevant though, as it was found to be slightly weaker than k-WL but similar in almost all ways, so much so that it was called a "special" k-dim Weisfeiler Lehman algorithm.

Given this two points, it's safe to assume that this algorithm could be equivalent not to k-WL but to its special version; however, due to time constraints this hypothesis could not be tested nor verified and as such remains a simple speculation on my part. It's also interesting that the only correctness proof we have been able to provide for the second algorithm ranks the two implementation as at least equally powerful when the second one is using a value of k one higher than the first's, which seems to be the case given experimental evidence. Furthermore, the proof in chapter 3 doesn't make any claims about whether a $m + 1$ run of the second implementation of k-WL is more powerful than a m run of the first, so it doesn't contradict any of the results obtained in chapter 5.

To wrap up the summary of what I've done for my final project, it's relevant adding that the code is currently in a peer-review like phase in the SageMath's integration pipeline, and is being checked and analysed by SageMath's senior contributors in order to merge it into the main code base, so I feel like the project has turned out to be successful in that regard, too.

6.2 Future developments

While a lot has been done, a lot more yet has to be in the scope of my project. A few things that come to mind that would be very interesting to work on in the future are:

1. Implementing specialised versions of k-WL for small values of k, since, as an example, for $k = 1$ it's possible to implement a specialised version that runs in $O(m \log n)$ as described in [11], while for $k = 2$ it's possible to create a highly optimised version that outperforms any implementation of 2-WL I've come across, such as the one implemented in [30]

2. Testing out the efficiency and usability of my implementations of k-WL in the several fields described in chapter 2, since due to time constraints the only tests that could be carried out were those for graph automorphism groups and isomorphism.

3. Given a way of quickly determining if a graph has been recognised or not, implementing an incremental k-WL that starts running 1-WL, than 2-WL and so on until the graphs orbits/orbitals have been correctly returned. The current correctness tests I implemented are not suitable for the task, since while efficient, they are still slower than any k-WL implementation.

4. Providing a formal proof of whether or not my second implementation is equivalent to k-WL, special k-WL or none of the above. In particular, further testing on the fact that increasing k by 1 seems to suffice to fix the issue would be warranted.

5. Trying out different approaches to the parallelisation of k-WL, maybe using different synchronisation procedures or even a different work split altogether, in order to improve performances with fewer bottlenecks and less memory overhead.

Appendices

Note

While the following appendices, containing the listings for the tests in chapter 5, should normally count towards the word count because appendices, they are also source code of a script and its output, and as such are excluded from the word count.

A

First algorithm test listing

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.CycleGraph(4)
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 2.31 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 137 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Correct'
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = Graph()
sage: G.add_edges([(0,1),(0,2),(0,3),(1,4),(2,4),(3,4),(1,2),(2,3)])
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 12 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 1.23 s
```

```
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 3, partition=final_p, result="vertex_classes")
Wall time: 10min 25s
sage: coc(res, Final, partition=final_p, cardinality=1)
'Correct'
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(2,0)
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 19.6 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
ValueError: Cannot return edge_classes for k = 1
sage: %time res = WL(G, 2)
Wall time: 15.9 s
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3) 7.6GB PEAK, >13GB foreseen
TIMEOUT
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(1,0) #Isomorphic to the Shrikhande graph
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 440 µs
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
ValueError: Cannot return edge_classes for k = 1
sage: %time res = WL(G, 2)
Wall time: 2.17 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 59.2 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 4)
Wall time: 1.79 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 5)
Wall time: 1min 29s
sage: coc(res, G, cardinality=2)
'Correct'
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: for i in range(1,65):
....:     G = graphs.planar_graphs(i).next()
....:     print("**********%i**********" % i)
....:     %time res = WL(G,3)
....:     coc(res, G, cardinality=2)
**********1**********
Wall time: 155 µs
'Correct'
**********2**********
Wall time: 230 µs
'Correct'
**********3**********
Wall time: 319 µs
'Correct'
**********4**********
Wall time: 397 µs
```

```
'Correct'
**********5**********
Wall time: 797 μs
'Correct'
**********6**********
Wall time: 1.38 ms
'Correct'
**********7**********
Wall time: 3.52 ms
'Correct'
**********8**********
Wall time: 6.32 ms
'Correct'
**********9**********
Wall time: 6.87 ms
'Correct'
**********10**********
Wall time: 11.7 ms
'Correct'
**********11**********
Wall time: 14 ms
'Correct'
**********12**********
Wall time: 30.9 ms
'Correct'
**********13**********
Wall time: 22.6 ms
'Correct'
**********14**********
Wall time: 32.5 ms
'Correct'
**********15**********
Wall time: 44.5 ms
'Correct'
**********16**********
Wall time: 52.6 ms
'Correct'
**********17**********
Wall time: 73.6 ms
'Correct'
**********18**********
Wall time: 87.4 ms
'Correct'
**********19**********
Wall time: 121 ms
'Correct'
**********20**********
Wall time: 131 ms
'Correct'
**********21**********
Wall time: 179 ms
'Correct'
**********22**********
Wall time: 200 ms
'Correct'
**********23**********
Wall time: 235 ms
'Correct'
**********24**********
Wall time: 283 ms
'Correct'
**********25**********
Wall time: 327 ms
'Correct'
**********26**********
Wall time: 382 ms
'Correct'
**********27**********
Wall time: 441 ms
'Correct'
```

```
**********28**********
Wall time: 546 ms
'Correct'
**********29**********
Wall time: 676 ms
'Correct'
**********30**********
Wall time: 715 ms
'Correct'
**********31**********
Wall time: 811 ms
'Correct'
**********32**********
Wall time: 1.04 s
'Correct'
**********33**********
Wall time: 1.09 s
'Correct'
**********34**********
Wall time: 1.25 s
'Correct'
**********35**********
Wall time: 1.43 s
'Correct'
**********36**********
Wall time: 1.59 s
'Correct'
**********37**********
Wall time: 1.87 s
'Correct'
**********38**********
Wall time: 2.36 s
'Correct'
**********39**********
Wall time: 2.51 s
'Correct'
**********40**********
Wall time: 2.8 s
'Correct'
**********41**********
Wall time: 3.11 s
'Correct'
**********42**********
Wall time: 3.33 s
'Correct'
**********43**********
Wall time: 4.11 s
'Correct'
**********44**********
Wall time: 6.2 s
'Correct'
**********45**********
Wall time: 6.86 s
'Correct'
**********46**********
Wall time: 8.03 s
'Correct'
**********47**********
Wall time: 8.79 s
'Correct'
**********48**********
Wall time: 9.45 s
'Correct'
**********49**********
Wall time: 10.3 s
'Correct'
**********50**********
Wall time: 11.2 s
'Correct'
**********51**********
```

```
Wall time: 12.8 s
'Correct'
**********52**********
Wall time: 13.4 s
'Correct'
**********53**********
Wall time: 14.9 s
'Correct'
**********54**********
Wall time: 16.3 s
'Correct'
**********55**********
Wall time: 17.9 s
'Correct'
**********56**********
Wall time: 19.4 s
'Correct'
**********57**********
Wall time: 21.3 s
'Correct'
**********58**********
Wall time: 23 s
'Correct'
**********59**********
Wall time: 25.5 s
'Correct'
**********60**********
Wall time: 27 s
'Correct'
**********61**********
Wall time: 28.5 s
'Correct'
**********62**********
Wall time: 29.6 s
'Correct'
**********63**********
Wall time: 30.7 s
'Correct'
**********64**********
Wall time: 32.7 s
'Correct'
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: n = 4
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
....: from sage.matrix.constructor import identity_matrix, matrix
....: H = skew_hadamard_matrix(4*n)
....: M = H[1:].T[1:] - identity_matrix(4*n-1)
sage: nmap = {0:0,1:1,-1:0}
sage: M = M.apply_map(lambda x: nmap[x])
sage: G = DiGraph(M, format='adjacency_matrix')
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 2.06 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 2, result="vertex_classes")
Wall time: 3.28 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 3, result="vertex_classes")
Wall time: 68.6 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time sage: res = WL(G, 4, result="vertex_classes")
Wall time: 1.68 s
sage: sage: coc(res, G, cardinality=1)
'Correct'
```

```
sage: %time res = WL(G, 2)
Wall time: 2.41 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 66.9 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 4)
Wall time: 1.56 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 5)
Wall time: 1min 23s
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 6)
Wall time: 35min 42s
sage: coc(res, G, cardinality=2)
'Correct'
```

B

Second algorithm test listing

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.CycleGraph(4)
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 46.4 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 159 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = Graph()
sage: G.add_edges([(0,1),(0,2),(0,3),(1,4),(2,4),(3,4),(1,2),(2,3)])
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 521 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 1.43 s
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
```

```
sage: %time res = WL(Final, 3, partition=final_p, result="vertex_classes")
Wall time: 2min 4s
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(2,0)
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 1.43 s
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
Wall time: 1.44 s
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 2)
Wall time: 12.5 s
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3) //6.5 MB PEAK
Wall time: 58min 39s
sage: coc(res, G, cardinality=2)
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(1,0) #Isomorphic to the Shrikhande graph
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 3.31 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
Wall time: 1.5 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 2)
Wall time: 2.77 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 42.3 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 4)
Wall time: 486 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 5)
Wall time: 3.8 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 6)
Wall time: 26.3 s
sage: coc(res, G, cardinality=2)
'Correct'



sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: for i in range(1,65):
....:     G = graphs.planar_graphs(i).next()
....:     print("**********%i**********" % i)
....:     %time res = WL(G,3)
....:     coc(res, G, cardinality=2)
**********1**********
Wall time: 374 µs
'Correct'
**********2**********
```

```
Wall time: 366 µs
'Correct'
**********3**********
Wall time: 325 µs
'Correct'
**********4**********
Wall time: 531 µs
'Correct'
**********5**********
Wall time: 1.23 ms
'Correct'
**********6**********
Wall time: 2.19 ms
'Correct'
**********7**********
Wall time: 3.55 ms
'Correct'
**********8**********
Wall time: 5.29 ms
'Correct'
**********9**********
Wall time: 8.07 ms
'Correct'
**********10**********
Wall time: 14.2 ms
'Correct'
**********11**********
Wall time: 12.6 ms
'Correct'
**********12**********
Wall time: 20.2 ms
'Correct'
**********13**********
Wall time: 30.3 ms
'Correct'
**********14**********
Wall time: 38.2 ms
'Correct'
**********15**********
Wall time: 47.6 ms
'Correct'
**********16**********
Wall time: 41.3 ms
'Correct'
**********17**********
Wall time: 78.7 ms
'Correct'
**********18**********
Wall time: 106 ms
'Correct'
**********19**********
Wall time: 124 ms
'Correct'
**********20**********
Wall time: 155 ms
'Correct'
**********21**********
Wall time: 195 ms
'Correct'
**********22**********
Wall time: 260 ms
'Correct'
**********23**********
Wall time: 303 ms
'Correct'
**********24**********
Wall time: 367 ms
'Correct'
**********25**********
Wall time: 432 ms
```

```
'Correct'
**********26**********
Wall time: 549 ms
'Correct'
**********27**********
Wall time: 644 ms
'Correct'
**********28**********
Wall time: 740 ms
'Correct'
**********29**********
Wall time: 862 ms
'Correct'
**********30**********
Wall time: 993 ms
'Correct'
**********31**********
Wall time: 1.21 s
'Correct'
**********32**********
Wall time: 1.36 s
'Correct'
**********33**********
Wall time: 1.51 s
'Correct'
**********34**********
Wall time: 1.73 s
'Correct'
**********35**********
Wall time: 1.92 s
'Correct'
**********36**********
Wall time: 2.15 s
'Correct'
**********37**********
Wall time: 2.65 s
'Correct'
**********38**********
Wall time: 2.87 s
'Correct'
**********39**********
Wall time: 3.16 s
'Correct'
**********40**********
Wall time: 3.47 s
'Correct'
**********41**********
Wall time: 3.85 s
'Correct'
**********42**********
Wall time: 4.21 s
'Correct'
**********43**********
Wall time: 4.65 s
'Correct'
**********44**********
Wall time: 5.51 s
'Correct'
**********45**********
Wall time: 5.94 s
'Correct'
**********46**********
Wall time: 6.35 s
'Correct'
**********47**********
Wall time: 6.9 s
'Correct'
**********48**********
Wall time: 7.52 s
'Correct'
```

```
**********49**********
Wall time: 7.96 s
'Correct'
**********50**********
Wall time: 8.76 s
'Correct'
**********51**********
Wall time: 9.43 s
'Correct'
**********52**********
Wall time: 10.7 s
'Correct'
**********53**********
Wall time: 11.9 s
'Correct'
**********54**********
Wall time: 12.6 s
'Correct'
**********55**********
Wall time: 13.5 s
'Correct'
**********56**********
Wall time: 14.4 s
'Correct'
**********57**********
Wall time: 15.3 s
'Correct'
**********58**********
Wall time: 16.6 s
'Correct'
**********59**********
Wall time: 17.7 s
'Correct'
**********60**********
Wall time: 19 s
'Correct'
**********61**********
Wall time: 20.1 s
'Correct'
**********62**********
Wall time: 22.1 s
'Correct'
**********63**********
Wall time: 25.3 s
'Correct'
**********64**********
Wall time: 26.4 s
'Correct'
```

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: n = 4
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
....: from sage.matrix.constructor import identity_matrix, matrix
....: H = skew_hadamard_matrix(4*n)
....: M = H[1:].T[1:] - identity_matrix(4*n-1)
sage: nmap = {0:0,1:1,-1:0}
sage: M = M.apply_map(lambda x: nmap[x])
sage: G = DiGraph(M, format='adjacency_matrix')
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 2.89 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 2, result="vertex_classes")
Wall time: 2.33 ms
sage: coc(res, G, cardinality=1)
'Refinable'
```

```
sage: %time res = WL(G, 3, result="vertex_classes")
Wall time: 19.4 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 4, result="vertex_classes")
Wall time: 527 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 2)
Wall time: 2.65 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 18.6 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 4)
Wall time: 518 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 5)
Wall time: 2.9 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 6)
Wall time: 22.3 s
sage: coc(res, G, cardinality=2)
'Correct'
```

C

Multithreaded second algorithm test listing

```
sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.CycleGraph(4)
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 85.3 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 612 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = Graph()
sage: G.add_edges([(0,1),(0,2),(0,3),(1,4),(2,4),(3,4),(1,2),(2,3)])
sage: H, p = graphs.CaiFurerImmermanGraph(G, False)
sage: Ht, pt = graphs.CaiFurerImmermanGraph(G, True)T
sage: Final = H.disjoint_union(Ht)
sage: final_p = []
sage: for part in p:
....:     temp_p = []
....:     for el in part:
....:         temp_p.append((0,el))
....:         temp_p.append((1,el))
....:     final_p.append(temp_p)
sage: %time res = WL(Final, 1, partition=final_p, result="vertex_classes")
Wall time: 469 ms
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
```

```
sage: %time res = WL(Final, 2, partition=final_p, result="vertex_classes")
Wall time: 2.1 s
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'
sage: %time res = WL(Final, 3, partition=final_p, result="vertex_classes")
Wall time: 1min 11s
sage: coc(res, Final, partition=final_p, cardinality=1)
'Refinable'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(2,0)
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 1.45 s
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
Wall time: 1.4 s
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 2)
Wall time: 4.72 s
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3) //16.74 MB PEAK
Wall time: 10min 8s
sage: coc(res, G, cardinality=2)
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: G = graphs.EgawaGraph(1,0) #Isomorphic to the Shrikhande graph
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 10.3 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 1)
Wall time: 5.72 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 2)
Wall time: 3.8 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 56.3 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 4)
Wall time: 333 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 5)
Wall time: 2.43 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: sage: %time res = WL(G, 6)
Wall time: 10.2 s
sage: coc(res, G, cardinality=2)
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: for i in range(1,65):
....:     G = graphs.planar_graphs(i).next()
....:     print("**********%i**********" % i)
....:     %time res = WL(G,3)
....:     coc(res, G, cardinality=2)
**********1**********
Wall time: 698 µs
```

```
'Correct'
**********2**********
Wall time: 945 μs
'Correct'
**********3**********
Wall time: 3.58 ms
'Correct'
**********4**********
Wall time: 1.15 ms
'Correct'
**********5**********
Wall time: 7.11 ms
'Correct'
**********6**********
Wall time: 13.1 ms
'Correct'
**********7**********
Wall time: 16.7 ms
'Correct'
**********8**********
Wall time: 28.1 ms
'Correct'
**********9**********
Wall time: 40.9 ms
'Correct'
**********10**********
Wall time: 57 ms
'Correct'
**********11**********
Wall time: 49.2 ms
'Correct'
**********12**********
Wall time: 65.2 ms
'Correct'
**********13**********
Wall time: 99.4 ms
'Correct'
**********14**********
Wall time: 96.7 ms
'Correct'
**********15**********
Wall time: 730 ms
'Correct'
**********16**********
Wall time: 435 ms
'Correct'
**********17**********
Wall time: 216 ms
'Correct'
**********18**********
Wall time: 873 ms
'Correct'
**********19**********
Wall time: 1.26 s
'Correct'
**********20**********
Wall time: 1.04 s
'Correct'
**********21**********
Wall time: 1.55 s
'Correct'
**********22**********
Wall time: 990 ms
'Correct'
**********23**********
Wall time: 1.53 s
'Correct'
**********24**********
Wall time: 1.59 s
'Correct'
```

```
**********25**********
Wall time: 1.78 s
'Correct'
**********26**********
Wall time: 2.27 s
'Correct'
**********27**********
Wall time: 2.05 s
'Correct'
**********28**********
Wall time: 2.64 s
'Correct'
**********29**********
Wall time: 2.75 s
'Correct'
**********30**********
Wall time: 3.17 s159
'Correct'
**********31**********
Wall time: 2.99 s
'Correct'
**********32**********
Wall time: 3.96 s
'Correct'
**********33**********
Wall time: 3.96 s
'Correct'
**********34**********
Wall time: 2.89 s
'Correct'
**********35**********
Wall time: 3.77 s
'Correct'
**********36**********
Wall time: 4.04 s
'Correct'
**********37**********
Wall time: 7.07 s
'Correct'
**********38**********
Wall time: 6.29 s
'Correct'
**********39**********
Wall time: 4.95 s
'Correct'
**********40**********
Wall time: 7.42 s
'Correct'
**********41**********
Wall time: 7.62 s
'Correct'
**********42**********
Wall time: 9.17 s
'Correct'
**********43**********
Wall time: 5.87 s
'Correct'
**********44**********
Wall time: 6.66 s
'Correct'
**********45**********
Wall time: 7.19 s
'Correct'
**********46**********
Wall time: 7.62 s
'Correct'
**********47**********
Wall time: 8.44 s
'Correct'
**********48**********
```

```
Wall time: 8.75 s
'Correct'
**********49**********
Wall time: 9.48 s
'Correct'
**********50**********
Wall time: 9.81 s
'Correct'
**********51**********
Wall time: 10.9 s
'Correct'
**********52**********
Wall time: 11.8 s
'Correct'
**********53**********
Wall time: 13.6 s
'Correct'
**********54**********
Wall time: 14 s
'Correct'
**********55**********
Wall time: 15.5 s
'Correct'
**********56**********
Wall time: 15.7 s
'Correct'
**********57**********
Wall time: 16.9 s
'Correct'
**********58**********
Wall time: 17.9 s
'Correct'
**********59**********
Wall time: 19.5 s
'Correct'
**********60**********
Wall time: 20.7 s
'Correct'
**********61**********
Wall time: 21.7 s
'Correct'
**********62**********
Wall time: 23.2 s
'Correct'
**********63**********
Wall time: 27.5 s
'Correct'
**********64**********
Wall time: 28.4 s
'Correct'

sage: from sage.graphs.weisfeiler_lehman import WeisfeilerLehman as WL
sage: from sage.graphs.weisfeiler_lehman import check_orbit_correctness as coc
sage: n = 4
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
....: from sage.matrix.constructor import identity_matrix, matrix
....: H = skew_hadamard_matrix(4*n)
....: M = H[1:].T[1:] - identity_matrix(4*n-1)
sage: nmap = {0:0,1:1,-1:0}
sage: M = M.apply_map(lambda x: nmap[x])
sage: G = DiGraph(M, format='adjacency_matrix')
sage: %time res = WL(G, 1, result="vertex_classes")
Wall time: 2.31 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 2, result="vertex_classes")
Wall time: 2.48 ms
sage: coc(res, G, cardinality=1)
'Refinable'
```

```
sage: %time res = WL(G, 3, result="vertex_classes")
Wall time: 8.2 ms
sage: coc(res, G, cardinality=1)
'Refinable'
sage: %time res = WL(G, 4, result="vertex_classes")
Wall time: 347 ms
sage: coc(res, G, cardinality=1)
'Correct'
sage: %time res = WL(G, 2)
Wall time: 2.34 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 3)
Wall time: 8.7 ms
sage: coc(res, G, cardinality=2)
'Refinable'
sage: %time res = WL(G, 4)
Wall time: 349 ms
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 5)
Wall time: 1.55 s
sage: coc(res, G, cardinality=2)
'Correct'
sage: %time res = WL(G, 6)
Wall time: 13.2 s
sage: coc(res, G, cardinality=2)
'Correct'
```

References

[1] B. Yu. Weisfeiler and A. A. Lehman. “A reduction of a graph to canonical form and an algebra arising during this reduction”. In: (1968).

[2] Ronald C. Read and Derek G. Corneil. “The graph isomorphism disease”. In: *Journal of Graph Theory* 1.4 (1977), pp. 339–363. DOI: 10.1002/jgt.3190010410.

[3] Sergei Evdokimov, Marek Karpinski, and Ilia Ponomarenko. “On a New High Dimensional Weisfeiler-Lehman Algorithm”. In: *Journal of Algebraic Combinatorics* 10.1 (July 1, 1999), pp. 29–45. DOI: 10.1023/A:1018672019177.

[4] Neil Immerman and Eric Lander. “Describing Graphs: A First-Order Approach to Graph Canonization”. In: *Complexity Theory Retrospective*. Ed. by Alan L. Selman. New York, NY: Springer New York, 1990, pp. 59–81. DOI: 10.1007/978-1-4612-4478-3_5.

[5] László Babai, Paul Erdo˝s, and Stanley M. Selkow. “Random Graph Isomorphism”. In: *SIAM Journal on Computing* 9.3 (Aug. 1980), pp. 628–635. DOI: 10.1137/0209047.

[6] Laszlo Babai and Ludik Kucera. “Canonical labelling of graphs in linear average time”. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. 20th Annual Symposium on Foundations of Computer Science (sfcs 1979). San Juan, Puerto Rico: IEEE, Oct. 1979, pp. 39–46. DOI: 10.1109/SFCS.1979.8.

[7] Jin-Yi Cai, Martin Furer, and Neil Immerman. “An optimal lower bound on the number of variables for graph identification”. In: *Combinatorica* 12.4 (Dec. 1992), pp. 389–410. DOI: 10.1007/bf01305232.

[8] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, II”. In: *Journal of Symbolic Computation* 60 (Jan. 2014), pp. 94–112. DOI: 10.1016/j.jsc.2013.09.003.

[9] László Babai. “Graph Isomorphism in Quasipolynomial Time”. In: *arXiv:1512.03547 [cs, math]* (Dec. 11, 2015). arXiv: 1512.03547.

[10] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.3.rc0)*. http://www.sagemath.org. 2018.

[11] Martin Grohe et al. “Color Refinement and its Applications”. In: Guy Van den Broeck et al. *An introduction to Lifted Probabilistic Inference*. Cambridge University Press, 2017.

[12] Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. “The Weisfeiler-Leman dimension of planar graphs is at most 3”. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). Reykjavik, Iceland: IEEE, June 2017. DOI: 10.1109/lics.2017.8005107.

[13] Yoshimi Egawa. "Characterization of H(n, q) by the parameters". In: *Journal of Combinatorial Theory, Series A* 31.2 (Sept. 1981), pp. 108–125. DOI: 10.1016/0097-3165(81)90007-8.

[14] Dmitrii V. Pasechnik. "Skew-symmetric association schemes with two classes and strongly regular graphs of type $L\sb 2n-1(4n-1)$". In: *Acta Applicandae Mathematicae. An International Survey Journal on Applying Mathematics and Mathematical Applications* 29.1 (1992), pp. 129–138.

[15] Martin Grohe et al. "Dimension Reduction via Colour Refinement". In: *arXiv:1307.5697 [cs, math]* (July 22, 2013). arXiv: 1307.5697.

[16] G. Tinhofer. "Graph isomorphism and theorems of Birkhoff type". In: *Computing* 36.4 (Dec. 1986), pp. 285–300. DOI: 10.1007/BF02240204.

[17] M. V. Ramana, E. R. Scheinerman, and D. Ullman. "Fractional isomorphism of graphs". In: *Discrete Mathematics* (1994), 132:247–265.

[18] Hanif D. Sherali and Warren P. Adams. "A hierarchy of relaxations and convex hull characterizations for mixed-integer zero—one programming problems". In: *Discrete Applied Mathematics* 52.1 (July 1994), pp. 83–106. DOI: 10.1016/0166-218X(92)00190-W.

[19] S. V. N. Vishwanathan et al. "Graph Kernels". In: *arXiv:0807.0093 [cs]* (July 1, 2008). arXiv: 0807.0093.

[20] Nino Shervashidze et al. "Weisfeiler-Lehman Graph Kernels". In: *The Journal of Machine Learning Research* 12 (Nov. 2011), pp. 2539–2561.

[21] Christopher Morris et al. "Faster Kernels for Graphs with Continuous Attributes via Hashing". In: *arXiv:1610.00064 [cs, stat]* (Sept. 30, 2016). arXiv: 1610.00064.

[22] D.G. Higman. "Coherent algebras". In: *Linear Algebra and its Applications* 93 (Aug. 1987), pp. 209–239. DOI: 10.1016/S0024-3795(87)90326-0.

[23] I. A. Faradžev, M. H. Klin, and M. E. Muzichuk. "Cellular Rings and Groups of Automorphisms of Graphs". In: *Investigations in Algebraic Theory of Combinatorial Objects*. Ed. by I. A. Faradžev et al. Red. by M. Hazewinkel. Vol. 84. Dordrecht: Springer Netherlands, 1994, pp. 1–152. DOI: 10.1007/978-94-017-1972-8_1.

[24] Martin Fürer. "On the Combinatorial Power of the Weisfeiler-Lehman Algorithm". In: *arXiv:1704.01023 [cs]* (Apr. 4, 2017). arXiv: 1704.01023.

[25] A. Cardon and M. Crochemore. "Partitioning a graph in O(¦A¦log2¦V¦)". In: *Theoretical Computer Science* 19.1 (July 1982), pp. 85–98. DOI: 10.1016/0304-3975(82)90016-0.

[26] Beman Dawes and David Abrahams. *Boost C++ Libraries*. 1999.

[27] Peter J. Cameron. *Permutation Groups*. Cambridge: Cambridge University Press, 1999. DOI: 10.1017/cbo9780511623677.

[28] L Finkelstein, W M. Kantor, and Takunari Miyazaki. *The Complexity of McKay's Canonical Labeling Algorithm*. Mar. 2, 2000.

[29] Martin Fürer. "Weisfeiler-Lehman Refinement Requires at Least a Linear Number of Iterations". In: *Automata, Languages and Programming*. Ed. by Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2076. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 322–333. DOI: 10.1007/3-540-48224-5_27.

[30] Luitpold Babel et al. "Algebraic Combinatorics in Mathematical Chemistry. Methods and Algorithms. II. Program Implementation of the Weisfeiler-Leman Algorithm". In: *arXiv:1002.1921 [math]* (Feb. 9, 2010). arXiv: 1002.1921.