Color Refinement and its Applications*

Martin Grohe
RWTH Aaachen University

Kristian Kersting
TU Dortmund University

Martin Mladenov
TU Dortmund University

Pascal Schweitzer
RWTH Aaachen University

January 28, 2017

Abstract

Color refinement is a simple algorithm that partitions the vertices of a graph according their "iterated degree sequence." It has very efficient implementations, running in quasilinear time, and a surprisingly wide range of applications. The algorithm has been designed in the context of graph isomorphism testing, and it is used as a subroutine in almost all practical graph isomorphism tools. Somewhat surprisingly, other applications, among them lifted inference, have been discovered recently.

In this chapter, we introduce the basic color refinement algorithm and explain how it can be implemented in quasilinear time. We discuss variations of this basic algorithm and its somewhat surprising characterizations in terms of logic and linear programming. Then we survey the main applications of color refinement to linear programming and to graph kernels.

1 Introduction

Color refinement, also known as *naive vertex classification* and *1-dimensional Weisfeiler-Leman algorithm*, is a combinatorial algorithm that aims to classify the vertices of a graph by similarity. It iteratively partitions, or *colors*, the vertices in a sequence of *refinement rounds*. Initially, all vertices get the same color. Then in each refinement round, any two vertices v, w that still have the same color get different colors if there is some color c such that v and w have a different number of neighbors of color c; otherwise they keep the same color. Thus after the first refinement round, two vertices have the same color if and only they have the same degree (number of incident edges) . After the second round, they have the same color if and only if for each k they have the same number of neighbors of degree k. The refinement process stops if in some refinement round the partition induced by the colors is no longer refined, that is, all pairs of vertices that have the same color before the refinement round still have the same color after the round. We call such a coloring that can no longer be refined a *stable coloring*.

Figure 1 shows how the coloring evolves on a randomly chosen graph with 30 vertices and 30 edges. A stable coloring is already reached after three refinement rounds. Observe that after the first refinement round, the colors reflect degrees: the (small) blue vertices have degree 0, the purple vertices have degree 1, the orange vertices degree 2, et cetera.

A naive implementation of color refinement will have a quadratic running time of $O(nm)$ steps on a graph with n edges and m vertices: we need at most n refinement

*Preliminary Draft of a Chapter that is to appear in: Guy Van den Broeck, Kristian Kersting, Sriraam Natarajan, David Poole, *An Introduction to Lifted Probabilistic Inference*, Cambridge University Press.

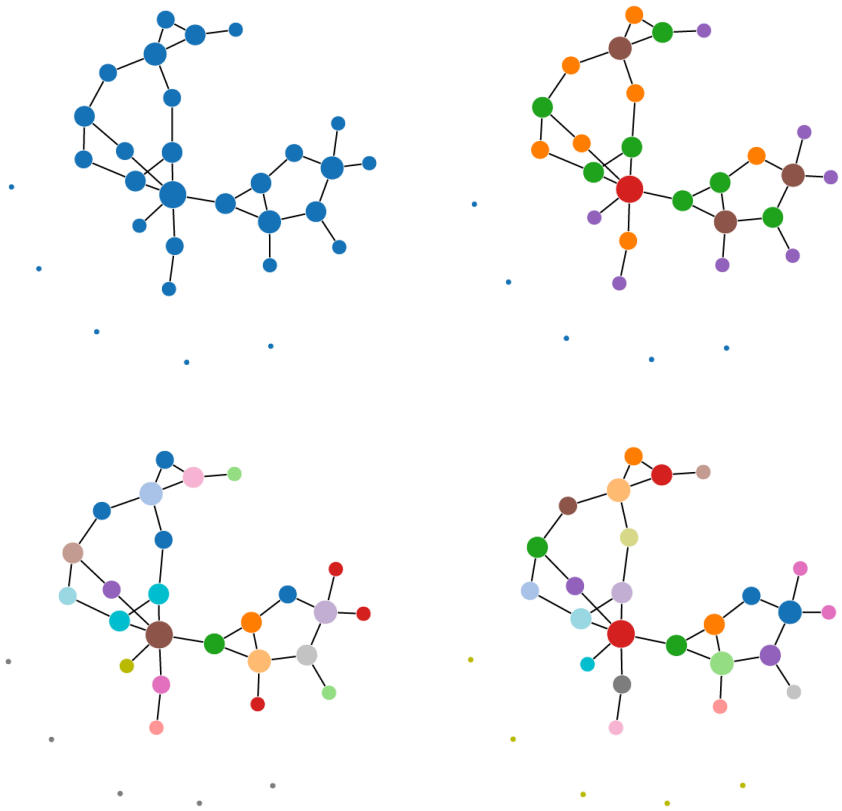


Figure 1. Color refinement on a random graph (from left to right, from top to bottom). A stable coloring is already reached after three refinement rounds. Observe that after the first refinement round, the colors reflect degrees: the (small) blue vertices have degree 0, the purple vertices have degree 1, the orange vertices degree 2, et cetera.

rounds, and each round requires $O(m)$ steps. However, there is a more efficient "asynchronous" refinement strategy, akin to asynchronous belief propagation, which runs in quasi-linear time $O(m \log n)$ [9, 28] (see Section 2).

1.1 Applications

Before we continue with variants of the basic algorithm, let us briefly describe its main applications. The original application of color refinement is *graph isomorphism testing* (see [29]). To test if two graphs, G and H are isomorphic, we can run color refinement on their disjoint union $G \uplus H$ and then compare the color patterns on the two graphs. If they are different, that is, there is a color such that G and H have different numbers of vertices of this color, then the graphs cannot be isomorphic. We say that color refinement *distinguishes* G and H. If the color patterns of G and H are the same, we still do not know if G and H are isomorphic or not. However, very often this simple isomorphism test succeeds. In fact, Babai, Erdös, and Selkow [3] proved that color refinement distinguishes almost all non-isomorphic graphs. More advanced graph isomorphism tests and almost all practical isomorphism tools use color refinement as a subroutine (sse Section 3 for details).

An application that has surfaced much more recently is the use of color refinement as a *pre-processing routine for solving mathematical programs or probabilistic inference*

tasks. The goal is to "compress" the problem instances by identifying objects, for example, nodes of a graphical model or rows and columns of the matrix of a linear program, that receive the same color in a suitable adaptation of color refinement to the instances of the problem at hand (see below). The idea is that objects of the same color are sufficiently similar so that we can combine them to form one "super object". This way, we obtain a smaller compressed instance on which we solve our original task. Then we transform the solution of the compressed instance to a solution of the original instance (see Section 6 for details).

A third application, also fairly recent, is the design of graph kernels based on color refinement. We need a variant of color refinement where we use $1, 2, \ldots$ as color names. It is crucial that the color names are assigned *canonically*, that is, do not depend on the representation of the input graph, but only on its isomorphism class. Then we associate integer vectors with the colorings: with a coloring that assigns colors $1, \ldots, k$ to the vertices of a graph, we associate the vector $(n_1, \ldots, n_k)$ where n_i is the number of vertices of color i. Thus color refinement yields such a vector for each graph, and we can use the scalar product on these vectors as our graph kernel. To avoid overfitting it has turned out to be best to only use the colorings obtained after the first few iterations of color refinement (see Section 7 for details).

1.2 Variants

Color refinement has several interesting variants, including a hierarchy of algorithms known as the Weifeiler-Leman algorithm(s), and generalisations to other structures such as vertex- and edge-colored graphs, directed graphs and arbitrary relational structures, weighted graphs and even matrices. Many of these variations are needed in the applications of color refinement.

The simplest extension of color refinement is the one to vertex-colored graphs: instead of starting the refinement procedure from the coloring that assigns the same color to all vertices, we start from the given vertex coloring. The extension to graphs that (also) have colored edges is not much harder; all we need to do in the refinement step is to consider the degrees with respect to different edge colors separately. Thus in a refinement round, two vertices v, w that still have the same color get different colors if there is some edge-color e and some color c of the current vertex-coloring such that v and w have a different number edges of color e into the color class c. To extend color refinement to directed graphs, we consider in-degrees and out-degrees separately.

Now consider weighted graphs with edge weights in the reals (in fact, the edge weights can be elements of an arbitrary Abelian group). One way of applying color refinement would be to just view the weights as edge-colors and apply the algorithm for edge-colored graphs. But for the "pre-processing" applications described in Section 6, the following variant is better: instead of refining by the number of edges into some color class, we refine by the sum of the weights of these edges. Thus in a refinement round, two vertices v, w that still have the same color get different colors if there is some color c of the current coloring such that the sum of the weights of the edges from v into color class c is different from the sum of the weights of the edges from w into color class c. Since we can interprete matrices as weighted bipartite graphs, this also gives us a natural color refinement procedure for matrices (see Section 5.1 for details).

Let us return to simple graphs (unweighted, undirected graphs containing no loops or multiple edges) and think about other ways of applying the iterative-coloring idea. What, if we color edges instead of vertices? But wait, why not color pairs of non-adjacent vertices as well? Or even color triples of vertices? This leads to the idea of the *Weisfeiler-Leman algorithm*. In the k-dimensional version of this algorithm, we color k-tuples of vertices; the 1-dimensional version is just color refinement. We shall describe

the algorithm in Section 4.

1.3 A Logical Characterisation of Color Refinement

There is an interesting connection between color refinement and logic: the algorithm can be viewed as an equivalence test for the logic C^2, the fragment of first order logic extended by counting quantifiers of the form $\exists^{\geq i}x$ ("there are at least i values for x") consisting of all formulas with just at most two variables. This means that two graphs satisfy the same C^2-sentences if and only if they cannot be distinguished by color refinement. The correspondence can be lifted to the higher-dimensional Weisfeiler-Leman algorithm: the k-dimensional Weisfeiler-Leman algorithm is an equivalence test for the logic C^{k+1}, defined like C^2 with $(k+1)$ variables per formula. This connection goes back to [19, 8]. It places color refinement and the Weisfeiler-Leman algorithm in the context of other logical equivalence tests, for example, *bisimilarity*, which may be viewed as an equivalence test for modal logic on finite transitions systems (that is, colored directed graphs). We shall not explore the connection between color refinement and logic in this chapter and refer the reader to [8] and [12, Chapter 3] for details.

2 Color Refinement in Quasilinear Time

We mentioned in the introduction that a naive implementation of color refinement has (at least) a quadratic running time; each of the n refinement rounds requires time $O(m)$, because all vertices and edges of the input graph need to be inspected to update the coloring. In this section, we shall describe a more efficient implementation running in time $O(m \log n)$. It uses a trick sometimes described as "processing the smaller half", which goes back to Hopcroft's quasilinear time algorithm for DFA-minimisation [18]. Actually, we present a simplified version of the algorithm (due to McKay [26]) running in time $O(n^2 \log n)$ and only hint at the improvements (due to Cardon and Crochemore [9], also see [28, 6]) reducing the time to $O(m \log n)$.

Let $G = (V, E)$ be a graph. For each vertex-coloring C of G, we let C' be the coloring obtained from C by applying one refinement round (as described in the introduction). At the moment, we do not care about the actual colors used, but only about the partition of the vertices into color classes that the coloring induces. We call a coloring C *stable* if C and C' induce the same partition. We say that a coloring C is *coarser* than a coloring D if the partition induced by D refines the partition induced by C. We observe that the refinement operation is monotone with respect to this partial order: if C is at least as coarse as D then C' is at least as coarse as D'. A consequence of this observation is that *color refinement computes the coarsest stable coloring.*

Now consider the algorithm displayed in Figure 2 for computing the coarsest stable coloring of a given graph G. Starting from the constant coloring, it repeatedly picks a color q and *refines all other colors with respect to* q, that is, for each color c it splits the vertices v of color $C(v) = c$ according to the number $D(v)$ of neighbors of color q they have, and then assigns new colors to these vertices according to theses degrees. The colors q that are used for splitting are stored in a queue Q, and the algorithm stops once this queue is empty. Whenever a color class c is split into new classes $B_{k_1}, \ldots, B_{k_2}$, then the new colors of the vertices (assigned in line 17) will be c_{max+i} for $k_1 \leq i \leq k_2$, where $c_{\max}$ is the last color used in the previous step. We add all of these new colors to the queue Q *except for the* i^* *such that the size* B_{i^*} *is maximum* (in line 12). If there are several B_i of maximum size, we take the first as B_{i^*}.

To see that the algorithm is correct, imagine first that we add all new colors to the queue Q, that is, we replace line 12 by $Q \leftarrow Q \cup \{c_{\max} + i \mid k_1 \leq i \leq k_2\}$. Then the algorithm would do exactly the same as the "normal" color refinement. But why does it also work if we do not add the largest new colors to the queue? To understand

```
COLOR-REFINEMENT(G)
Input: Graph G = (V, E)
Output: Coarsest stable coloring C of G
 1. C(v) ← 1 for all v ∈ V                                   // initial coloring
 2. P(1) ← V                          // P associates with each color the vertices of this color
 3. c_min ← 1; c_max ← 1                  // color names are always between c_min and c_max
 4. initialise queue Q ← {1}          // Q contains colors that will be used for refinement
 5. while Q ≠ ∅ do
 6.    q ← DEQUEUE(Q)
 7.    D(v) ← |N_G(v) ∩ P(q)| for all v ∈ V          // number of neighbors of v of color q
 8.    (B_1, ..., 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 ≤ k_2 such that P(c) = B_{k_1} ∪ ... ∪ B_{k_2}   // color class of c will be split
                                                                   // into classes B_{k_1}, ..., B_{k_2}
11.       i* ← arg max_{k_1 ≤ i ≤ k_2} |B_i|
12.       Q ← Q ∪ {c_max + i | k_1 ≤ i ≤ k_2, i ≠ i*}     // add all colors except the one
                                                          // with the largest class to queue Q
13.    end for
14.    c_min ← c_max + 1; c_max ← c_max + k                          // new color range
15.    for b = c_min to c_max do
16.       P(b) ← B_{b+1−c_min}                                       // new coloring
17.       C(v) ← b for all v ∈ P(b)
18.    end for
19. end while
20. return C
```

Figure 2. Efficient color refinement algorithm in pseudocode.

this, suppose that we split color c into new colors $d_1, \ldots, d_\ell$ and that, without loss of generality, d_ℓ is the largest one, which we omit from the queue. Now consider some other color c'. If we first refine c' with c and then with $d_1, \ldots, d_{\ell-1}$, then we have already taken d_ℓ into account, because the number of neighbors of a vertex in v of color d_ℓ is the number of neighbors of color c minus the sum of the number of neighbors of colors $d_1, \ldots, d_{\ell-1}$.

As described, the algorithm runs in time $O(n^2 \log n)$. Obviously, the running time is dominated by the time spent in the main loop in lines 5–19. To compute the degrees $D(v)$ in line 7, we go through the neighbors of all vertices v in the color class $P(q)$, this requires time $O(n \cdot |P(v)|)$. It turns out that this is the most expensive step opf the whole loop: lines 8–19 only require time $O(n)$; to create the orderd partition in line 8 we can use bucket sort. To bound the overall running time, we charge a time of $O(n)$ to each vertex in $P(q)$.

Now the crucial observation is that each vertex appears at most $O(\log n)$ times in a color class $P(q)$ for some q taken from the q in line 6. To see this, note that whenever a new color $b_i = c_{\max} + i$ is added to the queue Q in line 12 its color class B_i has at most half the size of the class $P(c)$. Now suppose a vertex v that appears on the queue ℓ times, say, in the colors $q_1, q_2, \ldots, q_\ell$. Then $q_{i+1} \leq q_i/2$ and thus $\ell \leq \log n$.

This means that the overall running time is $O(n^2 \log n)$, because each iteration of the loop requires time $O(n)$ per vertex in q, and each of the n vertices appears at most $\log n$ times.

To improve the running time to $O(m \log n)$, or more precisely $O(n+m \log n)$, we need to implement the main loop in such a way that one iteration requires time proportional to the number of edges incident with the vertices in the color class $P(q)$. Then we can charge time proportinal to its degree to every vertex in $P(q)$, and essentially the same analysis as above yields the desired overall runnning time of $O(m \log n)$. To be able to execute lines 6–18 in time proportional to the number of edges out of $P(q)$ we need to make one significant change: all vertices that are not adjacent to vertex in $P(q)$ keep their old color, so we never need to touch these vertices. The rest mainly amounts to a careful choice of data structures. We refer the reader to [6] for details.

We remark that Berkholz, Bonsma, and Grohe [6] proved that no faster color refinement algorithm is possible under some reasonable assumption about the type of algorithm, which includes all known approaches.

3 Application: Graph Isomorphism Testing

The *graph isomorphism problem (GI)* is the algorithmic problem of deciding whether two given graphs are isomorphic. Recall that an *isomorphism* between two graphs is a bijective mapping that preserves adjacencies and non-adjacencies. The question for the complexity of the isomorphism problem is one of the best known open problem in computer science; it was already mentioned in Karp's seminal paper on NP-completeness [20] more than 40 years ago. In a recent breakthrough, Babai [2] proved that GI is solvable in quasipolynomial time $n^{(\log n)^{O(1)}}$.

The most straightforward way to use color refinement as an isomorphism test for two graphs G, H is to run it on the disjoint union $G \uplus H$ (the two graphs are "added together", with no new edges) and then compare the color patterns obtained on the graphs. If they differ, then we know that the graphs are not isomorphic; we say that color refinement *distinguishes* the two graphs. However, if the color pattern on the two graphs is the same, this does not mean that they are isomorphic. Thus color refinement (used this way) is an *incomplete* isomorphism test. Let us say that color refinement *identifies* a graph G if it distinguishes G from all graphs H that are not isomorphic to G. Babai, Erdös, and Selkow [3] proved that color refinement identifies almost all graphs, in the sense that the fraction of graphs of order n that color refinement identifies converges to 1 as n goes to infinity. The convergence is exponentially fast [4]. Color refinement also identifies all trees and forests [19].

However, there are also very simple non-isomorphic graphs not distinguished by color refinement. The simplest example is a cycle of length 6 vs the disjoint union of two triangles. Clearly, these two graphs are not isomorphic, but color refinement assigns the same color to all vertices and effectively does not do anything. In fact, color refinement cannot distinguish any two regular graphs (every vertex has the same degree) of the same degree.

Even though color refinement fails to be a complete isomorphism test as a standalone algorithm, it is a key component of almost all practical graph isomorphism tools, which built on the so called *individualisation/refinement* (for short: *IR*) paradigm. The prototype for all these isomorphism tools is McKays "nauty" [26], which is still one of the best isomorphism tools.

IR-algorithms are usually implemented as canonization algorithms. They take a single graph G as input and return a graph G^* such that G^* is isomorphic to G and for isomorphic input graphs G and H the graphs G^* and H^* are identical. Typically, G^* will have an initial segment of the positive integers as its vertex set. Obviously, a canonization algorithm yields an isomorphism test: given two graphs G and H we simply compute G^* and H^* and compare them.

We call a coloring of a graph *discrete* if all vertices have different colors. We always

assume that the colors of our colorings come from some ordered set, typically the positive integers. Then from a discrete coloring D of a graph G we obtain an isomorphic copy $G^*(D)$ with vertex set $1, \ldots, n$ by simply renaming the vertices to $1, \ldots, n$ in the order of their colors. Recall that color refinement is a *canonical* coloring algorithm, that is, if f is an isomorphism from a graph G to a graph H and C_G, C_H are the stable colorings that color refinement produces, then $C_G(v) = C_H(f(v))$ for all vertices v of G.

The most basic version of an IR algorithm proceeds as follows. It runs color refinement on the input graph G. If the stable coloring C_G is discrete, it returns an isomorphic copy $G^*(C_G)$ corresponding to this coloring. If the coloring is not yet discrete, it *individualizes* a vertex in a color class with more than one color, that is, assigns a fresh color to the vertex. Then it runs color refinement again. If the coloring is still not discrete, it individualizes another vertex, and it repeats the individualization and refinement steps until eventually a discrete coloring is reached. This way we obtain a discrete coloring, but not in a canonical way. To fix this, whenever we individualize a vertex we have to branch on all vertices of the same color, individualizing one vertex in each branch. We build the *tree* of all these colorings. This tree is isomorphism-invariant. Associated with each leaf ℓ of the tree is a discrete coloring D_ℓ of the input graph G. We let $G^*_\ell = G^*(D_\ell)$ be the isomorphic copy of G corresponding to this discrete coloring. Let S_ℓ be the string of length n^2 that consists of the rows of the adjacency matrix of G^*_ℓ. Recall that the vertices of G^*_ℓ are $1, \ldots, n$; hence we can order the rows and columns of the adjacency matrix by the natural order on the vertices. Then our algorithm returns G^*_ℓ for a leaf ℓ with the lexicographically minimal string S_ℓ.

Examples 3.1. Figure 3 shows part of the tree of colorings built on a simple, six-vertex graph. At the root, we have the coloring obtained by color refinement. At the left child of the root, we have individualised vertex d and run color refinement again, at the middle child we have individualised e, and at the right child (not shown) we have individualised f. On the next level, on the leftmost branch we have individualised e, et cetera.

If the colors are orderd red $<$ green $<$ yellow $<$ blue $<$ orange $<$ dark red, then the graph G^*_ℓ associated with the leftmost leaf ℓ is obtained from G by renaming the vertices as follows: $f \mapsto 1$, $c \mapsto 2$, $d \mapsto 3$, $a \mapsto 4$, $e \mapsto 5$, $b \mapsto 6$. The string S_ℓ is 010000 100101 000100 011001 000001 010100. Curiously, for this graph the string is the same at all leaves.

The running time of our algorithm is exponential in the worst case, but we hope that usually most of the work is done by color refinement. Unfortunately, this is not always the case, in particular if the input graph has many symmetries. A key idea that makes the algorithm efficient in practice is that we can exploit symmetries to prune the tree of colorings. Whenever we arrive at a leaf ℓ of the tree such that $S_k = S_\ell$ for some leaf k processed before, or equivalently, $G^*_k = G^*_\ell$, then the colored graphs (G, D_k) and (G, D_ℓ) are isomorphic. The (unique) mapping $\gamma : V \to V$ that preserves the colors is an isomorphism between the colored graphs and hence an automorphism of G. We can exploit this information in the following way. Consider the lowest common ancestor t of the leaves k, ℓ, and let T_k and T_ℓ be the subtrees rooted at the children of t that contain k, ℓ, respectively. Let v_k, v_ℓ be the vertices individualised in the respective branches. Then we know that the whole trees T_k and T_ℓ of colorings are isomorphic, because the automorphism γ maps v_k to v_ℓ and fixes all vertices that have been individualised on the path from the root to t. Thus there is no need to explore the tree T_k any further. There is a second way in which we use the automorphisms γ found this way. We store the automorphisms and gradually build up a subgroup Γ of the automorphism group of G. Then, whenever we are about to enter a new branch of the tree, say, by individualising a vertex v_ℓ of the same color as a vertex v_k that has been individualised before (that is, v_k is a sibling of v_k on the left of v_k), and we find an automorphism γ in our group Γ

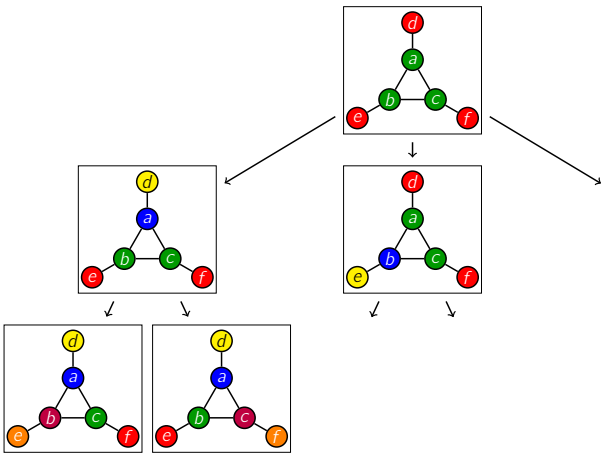


Figure 3. Individualisation/Refinement. Shown is a part of the tree of colorings built on a simple, six-vertex graph. The root is the coloring obtained by color refinement. The leafs are discrete colorings.

such that $\gamma(v_k) = v_\ell$ and γ fixes all vertices that have been individualised on the path from the root to the current node, then there is no need to enter the new branch where v_ℓ is individualised at all, because the subtree will look exactly like the subtree for v_k.

Examples 3.2. Figure 4 shows the full tree of colorings for the same graph G as in Example 3.1 if we apply the pruning techniques just described. Let ℓ_1, ℓ_3, ℓ_3 be the three leaves of the tree. Once we arrive at ℓ_2, we find that $S_{\ell_1} = S_{\ell_2}$ and hence that the permutation $\gamma = (b\ c)(e\ f)$ (in cycle notation, more explicitly, $g = \{a \mapsto a, b \mapsto c, c \mapsto b, d \mapsto d, e \mapsto f, f \mapsto e\}$) is an automorphism of G. Then, when we arrive at the leaf ℓ_3, we find that $S_{\ell_1} = S_{\ell_3}$ and hence that the permutation $\delta = (a\ b)(d\ e)$ is an automorphism. The lowest common ancestor of ℓ_1 and ℓ_3 is the root, and this means we do not have to explore the subtree rooted in the second child of the root any further. Hence we immediately jump back to the root, where in the next branch we individualise the node f (after we have individualised d, e at the first two children). However, we have already found the automorphism γ which maps e to f, and since we have already explored the e-branch, there is no need to do so for the f-branch. Thus the tree is complete.

4 The Weisfeiler-Leman Algorithm

The *k-dimensional Weisfeiler-Leman algorithm* (for short: *k-WL*) is a version of color refinement that colors k-tuples of vertices instead of single vertices. This makes the algorithm more powerful. For example, with high probability 2-WL distinguishes two randomly chosen d-regular graphs of the same size [7], and there is a k such that k-WL identifies all planar graphs [13]. The price we pay for this additional power is a significantly higher runtime: the best-known implementation of k-WL runs in time $O(k^2 n^{k+1} \log n)$ [19].

To describe the k-dimensional Weisfeiler-Leman algorithm, let us start with a different, somewhat more formal, view on the color refinement algorithm. Let $G = (V, E)$ be a graph. Let C_0, defined by $C_0(v) = 1$ for all $v \in V$, be the initial coloring. Denoting the coloring obtained after the ith refinement round by C_i, the new coloring after the $(i+1)$st round can be defined by assigning to each vertex the pair consisting of its old

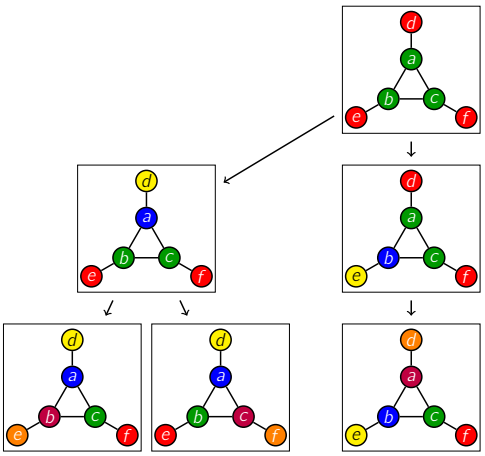


Figure 4. Full tree of colorings due to pruning the search tree of individualisations/refinements.

color and the multiset[1] of colors of its neighbors:

$$C_{i+1} := \Big(C_i(v), \big\{\!\!\big\{ C_i(w) \mid vw \in E \big\}\!\!\big\}\Big).$$

To avoid the awkwardness of handling nested multisets, in an implementation we would usually sort the multisets lexicographically and label them by integers, which we then use as the new colors. Note that the coarsest stable coloring is $C_\infty = C_i$ for the smallest i such that for all $v, w \in V$ it holds that $C_i(v) = C_i(w) \iff C_{i+1}(v) = C_{i+1}(w)$.

We need one more definition. The *atomic type* $\mathrm{atp}(\overline{v})$ of a k-tuple $\overline{v} = (v_1, \ldots, v_k)$ of vertices of a graph G describes the labeled subgraph induced by G on this tuple; formally we may describe it by a $(k \times k)$-matrix A with entries $A_{ij} = 2$ if $v_i = v_j$ and $A_{ij} = 1$ if $v_i v_j \in E$ and $A_{ij} = 0$ otherwise; the crucial property of atomic types that we need is that for tuples $\overline{v} = (v_1, \ldots, v_k), \overline{w} = (w_1, \ldots, w_k)$ we have $\mathrm{atp}(\overline{v}) = \mathrm{atp}(\overline{w})$ if and only if the mapping $v_i \mapsto w_i$ is well-defined and an isomorphism from the induced subgraph $G[\{v_1, \ldots, v_k\}]$ to the induced subgraph $G[\{w_1, \ldots, w_k\}]$.

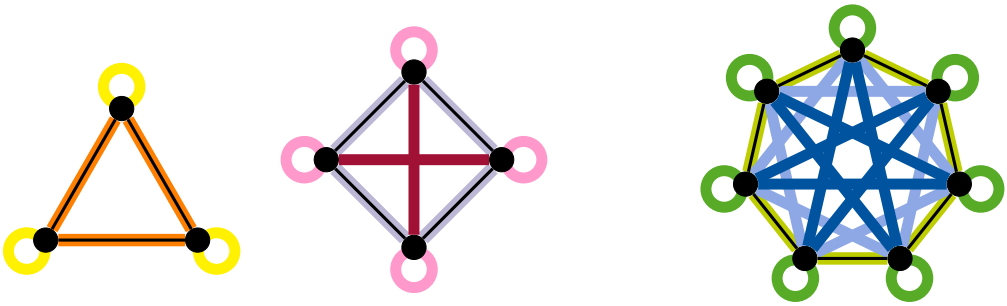
Now we can describe k-WL as the algorithm that, given a graph $G = (V, E)$, computes the following sequence of "colorings" C^k_i of V^k for $i \geq 0$ until it returns $C^k_\infty = C^k_i$ for the smallest i such that for all $\overline{v}, \overline{w}$ it holds that $C^k_i(\overline{v}) = C^k_i(\overline{w}) \iff C^k_{i+1}(\overline{v}) = C^k_{i+1}(\overline{w})$. The intial coloring C^k_0 assigns to each tuple its atomic type: $C^k_0(\overline{v}) := \mathrm{atp}(\overline{v})$. In the $(i+1)$st refinement round, the coloring C^k_{i+1} is defined by $C^k_{i+1}(\overline{v}) := \big(C^k_i(\overline{v}), M_i(\overline{v})\big)$, where $M_i(\overline{v})$ is the multiset

$$\Big\{\!\!\Big\{ \big(\mathrm{atp}(v_1, \ldots, v_k, w), C^k_i(v_1, \ldots, v_{k-1}, w), C^k_i(v_1, \ldots, v_{k-2}, w, v_k), \ldots, C^k_i(w, v_2, \ldots, v_k)\big) \Big| \; w \in V \Big\}\!\!\Big\},$$

for $\overline{v} = (v_1, \ldots, v_k)$. If $k \geq 2$, we can omit the entry $\mathrm{atp}(v_1, \ldots, v_k, w)$ from the tuples in the multiset, because all the information it contains is also contained in the entries $C^k_i(\ldots)$. It is easy to see that the coloring C^1_i computed by 1-WL coincides with the colorings C_i computed by color refinement, in the sense that $C_i(v) = C_i(w) \iff C^1_i(v) = C^1_i(w)$ for all vertices v, w.

A fairly straightforward modification of the algorithm discussed in Section 2 computes the coloring C^k_∞ in time $O(k^2 n^{k+1} \log n)$ [19].

[1] A *multiset* is an unordered collection of not necessarily distinct elements (whose multiplicities are counted). Formally, we may view a multiset as a mapping from a set to the positive integers.



(a) (b)

Figure 5. Colorings obtain by running 2-dimensional Weisfeiler-Leman (a) on a disjoint union of a cycle of length 3 and a cycle of length 4 as well as (b) on a cycle of length 7.

Examples 4.1. Let G be the graph consisting of the disjoint union of a cycle of length 3 and a cycle of length 4, say, with vertex set $V = \{1, \dots, 7\}$, where vertices $1, 2, 3$ form a triangle and vertices $4, \dots, 7$ form a 4-cycle. If we run color refinement on G, all vertices get the same color: $C_\infty(v) = C_0(v) = 1$ for all $v \in V$.

Now let us run 2-WL on G. There are three atomic types of pairs of elements, let us call them equal, adjacent, and non-adjacent. For example, $C_0^2(4,4) = \mathrm{e}$, $C_0^2(4,5) = \mathrm{a}$, and $C_0^2(4,3) = C_0^2(4,6) = \mathrm{n}$. For the coloring C_1^2 after the first refinement round we have, for example,

$$C_1^2(4,3) = \Big(\mathrm{n}, \big\{\!\!\big\{(\mathrm{n},\mathrm{a}), (\mathrm{n},\mathrm{a}), (\mathrm{n},\mathrm{e}), (\mathrm{e},\mathrm{n}), (\mathrm{a},\mathrm{n}), (\mathrm{n},\mathrm{n}), (\mathrm{a},\mathrm{n})\big\}\!\!\big\}\Big),$$

$$C_1^2(4,6) = \Big(\mathrm{n}, \big\{\!\!\big\{(\mathrm{n},\mathrm{n}), (\mathrm{n},\mathrm{n}), (\mathrm{n},\mathrm{n}), (\mathrm{e},\mathrm{n}), (\mathrm{a},\mathrm{a}), (\mathrm{n},\mathrm{e}), (\mathrm{a},\mathrm{a})\big\}\!\!\big\}\Big),$$

where we omit the atomic types from the tuples in the multiset.

The coloring C_1^2 is not yet the stable coloring. C_2^2 does not refine C_1^2 on the pairs of distinct vertices, but it does on the pairs of equal vertices. The reader can easily verify that C_1^2 takes the same value on all pairs (v, v), but we have $C_2^2(1,1) \neq C_2^2(4,4)$. The coloring C_2^2 is stable. See Figure 5(a) for an illustration. The coloring is symmetric, that is, $C_2^2(v,w) = C_2^2(w,v)$ for all v, w, hence we can display the colors as undirected edges.

By comparison, Figure 5(b) shows the coloring obtained by running 2-WL on a cycle of length 7. Since the color pattern is different from the one on the graph G in Figure 5(a), 2-WL distinguishes the two graphs. Color refinement does not, because both graphs are 2-regular.

5 Fractional Isomorphism

Suppose we try to solve the graph isomorphism problem by linear programming. We want to describe the isomorphisms between G and H as the solutions of an integer linear program, or rather, a system of linear equalities and inequalities.[2] Quite surprisingly, there is a close correpondence between the real solutions to this system and the stable colorings of the disjoint union $G \uplus H$.

[2]Graph isomorphism is not an optimisation problem, hence our "linear program" has no objective function, or rather, we are only interested in the feasible solutions. Still, it will sometimes be convenient, to use linear programming terminology.

For the rest of the section, let G and H be two graphs with vertex sets V, W of the same size n, and let A and B be their adjacency matrices. Without loss of generality we assume that V and W are disjoint. A and B are $n \times n$ matrices with $\{0,1\}$-entries $A_{vv'}$ and $B_{ww'}$. We think of the rows and columns of A as being indexed by vertices of G and the rows and columns of B as being indexed by vertices of H. We write $A \in \{0,1\}^{V \times V}$ and $B \in \{0,1\}^{W \times W}$.

The graphs G and H are isomorphic if and only if there is a permutation matrix $X \in \{0,1\}^{V \times W}$ such that $X^t A X = B$. Recall that a *permutation matrix* is a $\{0,1\}$-matrix with exactly one 1-entry in each row and in each column. A permutation matrix $X \in \{0,1\}^{V \times W}$ encodes the bijective mapping $\pi_X : V \to W$ defined by $\pi_X(v) = w$ for the unique w with $X_{vw} = 1$. Then $X^t A X$ is the adjency matrix of the graph $\pi_X(A)$. To turn it into a system of of linear equations, we equivalently rewrite the equation $X^t A X = B$ as

$$AX = XB, \tag{A}$$

using the fact that permutation matrices X are orthogonal, that is, the transpose X^t is the inverse X^{-1}. Equation (A) combined with equations forcing X to be a permutation matrix leads to the following system of linear equations and inequalities in the variables X_{vw} representing the entries of X:

$\mathrm{ISO}(G,H)$

$$\sum_{v' \in V} A_{vv'} X_{v'w} = \sum_{w' \in W} X_{vw'} B_{w'w} \tag{B}$$

$$\sum_{w \in W} X_{vw} = \sum_{v \in V} X_{vw} = 1 \tag{C}$$

$$X_{vw} \geq 0 \qquad \text{for all } v \in V, w \in W. \tag{D}$$

Our discussion shows that the integer solutions to this system correspond to the isomorphisms between G and H. But what about the real (or rational) solutions? Let us call them *fractional isomorphisms*. Note that fractional isomorphism are doubly stochastic matrices X satisfying equation (A). A matrix is *doubly stochastic* if its row sums and column sums are 1, that is, if it satisfies equations (C). Tinhofer [35] proved the following beautiful theorem. We present a proof of this theorem due to Selman [32].

Theorem 5.1 ([35]). *There is a fractional isomorphism from G to H if and only if color refinement does not distinguish G and H.*

Proof. We first prove the easier backward direction. Suppose that color refinement does not distinguish G and H. Let $U_1, \ldots, U_\ell \subseteq V \cup W$ be the color classes of the coarsest stable coloring C_∞, and let $V_i = V \cap U_i$ and $W_i = W \cap U_i$. Since color refinement does not distinguish G and H, we have $|V_i| = |W_i|$. We define a matrix $X \subseteq \mathbb{R}^{V \times W}$ by

$$X_{vw} = \begin{cases} \frac{1}{|V_i|} & \text{if } v, w \in U_i \text{ for some } i, \\ 0 & \text{othewise.} \end{cases}$$

Since $|V_i| = |W_i|$, this matrix is doubly stochastic and thus satisfies equations (C) and (D). To see that it satisfies (B), let $v \in V \cap U_i$ and $w \in W \cap U_j$. Then

$$\begin{aligned} \sum_{v' \in V} A_{vv'} X_{v'w} &= \sum_{v' \in V \cap U_j} \frac{A_{vv'}}{|V_j|} = \frac{2|N_G(v) \cap U_j|}{|U_j|} \\ &= \frac{2|N_H(w) \cap U_i|}{|U_i|} = \sum_{w' \in W \cap U_i} \frac{B_{w'w}}{|W_i|} = \sum_{w' \in W} X_{vw'} B_{w'w}. \end{aligned}$$

Here the first and last equalities follows from the definition of X and the second and fourth from the definition of the adjacency matrices and the fact that $|U_i| = 2|V_i| = 2|W_i|$

and $|U_j| = 2|V_j| = 2|W_j|$. The crucial third equality follows from the fact that the coloring is stable, which implies that there are numbers n_{ij} and n_{ji} such that every vertex in color class U_i has n_{ij} neighbors in class U_j and every vertex in class U_j has n_{ji} neighbors in class U_i. Then the number of edges between the classes U_i and U_j is $n_{ij}|U_i| = n_{ji}|U_j|$. As $n_{ij} = |N_G(v) \cap U_j|$ and $n_{ji} = |N_H(w) \cap U_i|$, this implies the equality.

To prove the forward direction, suppose that X is fractional isomorphism from G to H. Let K be the graph with vertex set $V \cup W$ and edge set $\{vw \mid X_{vw} > 0\}$. Let $K_1, \ldots, K_\ell$ be the connected components of K, and let U_i be the vertex set of K_i. We define a coloring $C : V \cup W \to \{1, \ldots, \ell\}$ by $C(u) = i$ if $u \in U_i$. We shall prove that C is a stable coloring satisfying

$$|V \cap C^{-1}(i)| = |W \cap C^{-1}(i)| \qquad \text{for all } i. \tag{E}$$

Once we have proved this, we are done, because the coarsest stable coloring C_∞ refines the stable coloring C and thus by (E) satisfies $|V \cap C_\infty^{-1}(c)| = |W \cap C_\infty^{-1}(c)|$ for all colors c. Hence color refinement does not distinguish G and H.

Equation (E), or equivalently $|V \cap U_i| = |W \cap U_i|$ is proved by a straightfowrad calculation:

$$|V \cap U_i| = \sum_{v \in V \cap U_i} \sum_{w \in W} X_{vw} = \sum_{v \in V \cap U_i} \sum_{w \in W \cap U_i} X_{vw} = \sum_{w \in W \cap U_i} \sum_{v \in V} X_{vw} = |W \cap U_i|.$$

where the first and fourth equalities follow from (C) and the second and third equalities from the fact that $X_{vw} = 0$ unless v and w belong to the same set U_j.

To prove that the coloring C is stable, we consider color classes U_i, U_j. We need to prove that for all $v, w \in U_i$ we have $|N(v) \cap U_j| = |N(w) \cap U_j|$. Here by $N(u)$ we mean $N_G(u)$ if $u \in V$ and $N_H(u)$ if $u \in W$. It will be important to distinguish $N(u)$ from $N_K(u) = \{u' \mid X_{uu'} > 0 \text{ or } X_{u'u} > 0\}$. So let $v \in U_i$, where without loss of generality we assume that $v \in V$. Then

$$\begin{aligned} |N(v) \cap U_j| &= \sum_{v' \in V \cap U_j} A_{vv'} = \sum_{v' \in V \cap U_j} A_{vv'} \sum_{w \in W} X_{v'w} = \sum_{w \in W \cap U_j} \sum_{v' \in V} A_{vv'} X_{v'w} \\ &= \sum_{w \in W \cap U_j} \sum_{w' \in W} X_{vw'} B_{w'w} = \sum_{w' \in W} X_{vw'} \sum_{w \in W \cap U_j} B_{w'w} = \sum_{w' \in N_K(v)} X_{vw'} |N(w') \cap U_j|. \end{aligned}$$

Here the second equality follows from (C) and the third from the fact that $X_{v'w} \neq 0$ only if v' and w are from the same color class U_j. The fourth equality follows from (B), and the last equality follows from the fact that $X_{vw'} \neq 0$ only if w' is neighbor of v in the graph K. Note that we have written $|N(v) \cap U_j|$ as a *positive convex combination* of the numbers $|N(w) \cap U_j|$ for $w \in N_K(v)$, that is, a linear combination with positive coefficients whose sum is 1.

Now we use the following simple fact: *if f is a real valued function defined on the vertex set of a connected graph, and for all v the value $f(v)$ is a positive convex combination of the values $f(w)$ for the neighbors w of v, then f is constant.* We apply this fact to the graph K_i and the function $f(u) = |N(u) \cap U_j|$. Thus $|N(u) \cap U_j| = |N(u') \cap U_j|$ for all $u, u' \in U_i$, which proves that the coloring C ist stable. □

Tinhofer's Theorem has a nice generalisation to the higher dimensional Weisfeiler-Leman algorithm. To put it in context, let us briefly review some ideas from combinatorial optimization. Suppose we want to solve an integer linear program L. We may start by looking at the real solutions to L, that is, drop the integrality constraints, and then maybe round them to integral solutions. But chances are that the polytope $P_{\mathbb{R}}$ of real solutions is too far from the polytope $P_{\mathbb{Z}}$ generated by the integer solutions, the

latter being the one we are interested in. Then what we can do is add additional linear constraints in such a way that the resulting linear program L' has the same integer solutions. Let $P'_{\mathbb{R}}, P'_{\mathbb{Z}}$ be the polytopes corresponding to L'. Then $P_{\mathbb{Z}} = P'_{\mathbb{Z}} \subseteq P'_{\mathbb{R}} \subseteq P_{\mathbb{R}}$. Maybe $P'_{\mathbb{R}}$ is sufficiently close to $P'_{\mathbb{Z}}$ for our purposes. If it is not, we may repeat the process and add further linear constraints so the we get a linear program L'', then maybe a linear program L''', et cetera. There are systematic ways of doing this, leading to so-called "lift-and-project"[3] hierarchies [5, 24, 33, 22].

One of the most important of these hierarchies is the *Sherali-Adams hierarchy* [33]. Atserias and Maneva [1] and independently Malkin [25] established a close connection between the Sherali-Adams hierarchy over the linear program $\mathrm{ISO}(G, H)$ and distinguishability by k-WL. The following linear program $\mathrm{ISO}^{(k)}(G, H)$ is a combination of of the equations of the kth level and the $(k+1)$th level of the Sherali-Adams hierarchy over $\mathrm{ISO}(G, H)$. The variables of this linear program are X_p for $p \subseteq V \times W$ with $|p| \leq k+1$. To understand the equations, we should view the indices p as relations between $(k+1)$ vertices of G and H. It can be shown that the equations and inequalities imply that $X_p > 0$ only if p is a *partial isomorphism*, that is, an injective mapping that preserves adjacencies and non-adjacencies.

$\mathrm{ISO}^{(k)}(G, H)$

$$\sum_{v' \in V} A_{vv'} X_{p \cup \{(v',w)\}} = \sum_{w' \in W} X_{p \cup \{(v,w')\}} B_{w'w} \qquad \text{for all } p \text{ of size } |p| \leq k-1 \text{ and all } v, w$$

$$\sum_{w \in W} X_{p \cup \{(v,w)\}} = \sum_{v \in V} X_{p \cup \{(v,w)\}} = X_p \qquad \text{for all } p \text{ of size } |p| \leq k \text{ and all } v, w$$

$$X_\emptyset = 1$$

$$X_p \geq 0 \qquad \text{for all } p \text{ of size } |p| \leq k+1$$

Theorem 5.2 ([1, 16, 25]). $\mathrm{ISO}^{(k)}(G, H)$ *has a rational solution if and only if k-WL does not distinguish G and H.*

5.1 Color Refinement and Fractional Isomorphism of Matrices

There is an interesting and, as we shall see in the next section, quite useful generalization of Tinhofer's theorem to weighted graphs, which we may also view as symmetric real matrices. We adopt color refinement to weighted graphs by refining not by degree, but by the sum of the edge weights. So let $A \in \mathbb{R}^{V \times V}$ be a symmetric matrix. We may view A as the adjacency matrix of a weighted graph G with vertex set V and an edge with weight A_{vw} between v and w for all v, w such that $A_{vw} \neq 0$. By a *coloring of A* we mean a coloring of the index set V. Color refinement computes colorings $C_0, C_1, \ldots$ of A in a sequence of refinement rounds. The initial coloring C_0 assigns the same color to all $v \in V$. In the $(i+1)$th refinement round, the algorithm assigns the same color to vertices v, w if and only if $C_i(v) = C_i(w)$ and for all colors c in the range of C_i,

$$\sum_{w \in V \cap C_i^{-1}(c)} A_{vw} = \sum_{w' \in V \cap C_i^{-1}(c)} A_{v'w'}. \tag{F}$$

We let $C_\infty = C_i$ for the smallest i such that C_{i+1} induces the same partition of V into color classes as C_i.

We call a coloring C of A *stable* if it satisfies (F) (with C instead of C_i) for all $v, w \in V$ with $C(v) = C(w)$. It is easy to see that C_∞ is the coarsest stable coloring

[3] Please note that "lift" here is used in a different way then in "lifted" probabilistic inference.

of A. Adopting the algorithm described in Section 2, we can compute C_∞ in time $O(n^2 \log n)$, where $n = |V|$, and even $O(m \log n)$, where m is the number of nonzero entries of A, assuming the matrix is suitably represented.

Now let $A \in \mathbb{R}^{V \times V}$ and $B \in \mathbb{R}^{W \times W}$, where $|V| = |W|$ and V and W are disjoint, be two symmetric matrices. The following matrix corresponds to the disjoint union of the weighted graphs:

$$A \uplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \in \mathbb{R}^{(V \cup W) \times (V \cup W)}.$$

Let C_∞ be the coarsest stable coloring of $A \uplus B$. We say that color refinement *distinguishes* A and B if for all colors c in the range of C_∞ we have $|V \cap C_\infty^{-1}(c)| = |W \cap C_\infty^{-1}(c)|$.

A *fractional isomorphism* from A to B is a doubly stochastic matrix $X \in \mathbb{R}^{V \times W}$ such that $AX = XB$. We obtain a direct generalisation of Tinhofer's Theorem, with essentially the same proof.

Theorem 5.3 ([14]). *There is a fractional isomorphism from A to B if and only if color refinement does not distinguish A and B.*

We can further generalise this to arbitrary, not necessarily symmetric matrices. Let $A \in \mathbb{R}^{V_1 \times V_2}$, where V_1 and V_2 are disjoint (but not necessarily of the same size). We say that a *stable bicoloring* of A is a pair (C_1, C_2) of colorings of V_1, V_2, respectively, such that

$$\sum_{v_{3-i} \in V_{3-i} \cap C_{3-i}^{-1}(c)} A_{v_1 v_2} = \sum_{v'_{3-i} \in V_{3-i} \cap C_{3-i}^{-1}(c)} A_{v'_1 v'_2}$$

for $i = 1, 2$, all $v_i, v'_i \in V_i$ with $C_i(v_1) = C_i(v'_i)$, and all c in the range of C_{3-i}. Equivalently, we may view a stable bi-coloring as a stable coloring C of the matrix

$$\begin{pmatrix} 0 & A \\ A^t & 0 \end{pmatrix} \in \mathbb{R}^{(V_1 \cup V_2) \times (V_1 \cup V_2)}$$

that refines the initial coloring with two color classes V_1 and V_2. This shows that we can compute the coarsest stable bicoloring of a matrix in time $O(m \log (n_1 + n_2))$, where $n_i = |V_i|$ and m is the number of nonzero entries of A.

We can then define a notion of a *fractional bi-isomorphism* between two matrices $A \in \mathbb{R}^{V_1 \times V_2}$, $B \in \mathbb{R}^{W_1 \times W_2}$ as a pair of doubly stochastic matrices $X_1 \in \mathbb{R}^{V_1 \times W_1}$, $X_2 \in \mathbb{R}^{V_2 \times W_2}$ such that $AX_2 = X_1 B$. It is not difficult to prove the analogue of Theorem 5.3 for fractional bi-isomorphism. We only state a technical lemma (for later reference) that follows from the easy direction of the proof.

Lemma 5.4 ([14]). *Let $A \in \mathbb{R}^{V_1 \times V_2}$ and (C_1, C_2) a stable bi-coloring of A. For $i = 1, 2$, let $X_i \in \mathbb{R}^{V_i \times V_i}$ be the matrix with entries*

$$(X_i)_{vv'} = \begin{cases} \frac{1}{|C_i^{-1}(c)|} & \text{if } C_i(v) = C_i(v') = c \text{ for some } c \text{ in the range of } C_i, \\ 0 & \text{if } C_i(v) \neq C_i(v'). \end{cases}$$

Then (X_1, X_2) is a fractional automorphism *of A, that is, X_1, X_2 are doubly stochastic and $AX_2 = X_1 A$.*

6 Application: Linear Programming

The results of the previous section establish a surprising connection between color refinement and linear algebra. In this section, we will show how this connection can be used for pre-processing linear programs in order to "compress" ("lift", in the sense of lifted probabilistic inference) them to smaller equivalent linear programs.

To explain the method, we start with the problem of solving a system $A\boldsymbol{x} = \mathbf{1}$, of linear equations, where $A \in \mathbb{R}^{m\times n}$ and $\mathbf{1} = (1,\dots,1)^t \in \mathbb{R}^m$. Let (C, D) be a stable bi-coloring of A. Suppose that the range of C is $\{1,\dots,k\}$ and the range of D is $\{1,\dots,\ell\}$. We define two matrices $P \in \mathbb{R}^{m\times k}$ and $P^* \in \mathbb{R}^{k\times m}$ by

$$P_{ip} = \begin{cases} 1 & \text{if } C(i) = p, \\ 0 & \text{otherwise,} \end{cases} \qquad P^*_{pi} = \begin{cases} \frac{1}{|C^{-1}(p)|} & \text{if } C(i) = p, \\ 0 & \text{otherwise.} \end{cases} \tag{G}$$

Observe that $X = PP^* \in \mathbb{R}^{m\times m}$ is the matrix with entries $X_{ii'} = 1/|C^{-1}(p)|$ if for i, i' with $C(i) = C(i') = p$ and $X_{ii'} = 0$ for i, i' with $C(i) \neq C(i')$. Similarly, we define matrices $Q \in \mathbb{R}^{n\times\ell}$ and $Q^* \in \mathbb{R}^{\ell\times n}$ by

$$Q_{jq} = \begin{cases} 1 & \text{if } D(j) = q, \\ 0 & \text{otherwise,} \end{cases} \qquad Q^*_{qj} = \begin{cases} \frac{1}{|D^{-1}(q)|} & \text{if } D(j) = q, \\ 0 & \text{otherwise} \end{cases} \tag{H}$$

and let $Y = QQ^* \in \mathbb{R}^{n\times n}$. Observe that P, P^*, Q and Q^* are all stochastic matrices and that X and Y are doubly stochastic. By Lemma 5.4, we have

$$AY = XA. \tag{I}$$

Moreover, a simple calculation shows

$$P^* = P^*X, \qquad Q = YQ. \tag{J}$$

Now we let $A' = P^*AQ \in \mathbb{R}^{k\times\ell}$ and $\mathbf{1}' = (1,\dots,1)^t \in \mathbb{R}^k$. Note that the matrix A' may be substantially smaller than A; this happens if A has many symmetries or at least "regularities". We claim that the system $A'\boldsymbol{x}' = \mathbf{1}'$ is equivalent to our original system in the sense that the solution spaces can be mapped into each other by simple linear transformation. Indeed, if $\boldsymbol{x}$ is a solution to $A\boldsymbol{x} = \mathbf{1}$, then $\boldsymbol{x}' = Q^*\boldsymbol{x}$ is a solution to $A'\boldsymbol{x}' = \mathbf{1}'$:

$$A\boldsymbol{x}' = P^*AQQ^*x = P^*AYx = P^*XAx = P^*X\mathbf{1} = P^*PP^*\mathbf{1} = \mathbf{1}', \tag{K}$$

where the last equality holds because the matrices P and P^* are stochastic and thus $P^*\mathbf{1} = \mathbf{1}'$ and $P\mathbf{1}' = \mathbf{1}$. Conversely, suppose that $\boldsymbol{x}'$ is a solution to $A'\boldsymbol{x}' = \mathbf{1}'$, and let $\boldsymbol{x} = Q\boldsymbol{x}'$. Then

$$A\boldsymbol{x} = AQ\boldsymbol{x}' = AYQ\boldsymbol{x}' = XAQ\boldsymbol{x}' = PA'\boldsymbol{x}' = P\mathbf{1}' = \mathbf{1}. \tag{L}$$

The method easily extends to linear programs. Consider a linear program in standard form:

$$\begin{aligned} \text{L} \qquad & \min \quad \boldsymbol{c}^t\boldsymbol{x} \\ & \text{subject to} \quad A\boldsymbol{x} = \boldsymbol{b}, \quad \boldsymbol{x} \geq \mathbf{0}, \end{aligned}$$

where $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{b} = (b_1,\dots,b_m)^t \in \mathbb{R}^m$, and $\boldsymbol{c} = (c_1,\dots,c_n)^t \in \mathbb{R}^n$. We define an initial bi-coloring (C_0, D_0) of the matrix A by $C_0(i) = b_i$ and $D_0(j) = c_j$, and we let (C, D) be the coarsest stable bi-coloring of A that refines (C_0, D_0). We define the matrices P, P^*, Q, Q^*, X, Y as above and note that (I) and (J) still hold. We let $A' = P^*AQ$ and $\boldsymbol{b}' = P^*\boldsymbol{b}$ and $\boldsymbol{c}' = Q^*\boldsymbol{c}$. We obtain a new linear program:

$$\begin{aligned} \text{L}' \qquad & \min \quad (\boldsymbol{c}')^t\boldsymbol{x}' \\ & \text{subject to} \quad A\boldsymbol{x}' = \boldsymbol{b}', \quad \boldsymbol{x}' \geq \mathbf{0}. \end{aligned}$$

Let $\boldsymbol{x}$ be a feasible solution to L and $\boldsymbol{x}' = Q^*\boldsymbol{x}$. Then $\boldsymbol{x}'$ is nonnegative, and the same calculation as in (K) shows that $A'\boldsymbol{x}' = \boldsymbol{b}'$. Hence $\boldsymbol{x}'$ is a feasible solution to L$'$.

Conversely, let $\boldsymbol{x}'$ is a feasible solution to L$'$, and let $\boldsymbol{x} = Q\boldsymbol{x}'$. Then $\boldsymbol{x}$ is nonnegative, and the calculation (L) shows that $A\boldsymbol{x} = P\boldsymbol{b}'$. As the stable coloring C refines our initial coloring C_0, for all i, i' with $C(i) = C(i')$ we have $b_i = b_{i'}$. Thus

$$(P\boldsymbol{b}')_i = (X\boldsymbol{b})_i = \sum_{i'=1}^{m} X_{ii'}b'_i = \sum_{i' \in C^{-1}(i)} \frac{1}{|C^{-1}(i)|} b_{i'} = b_i \sum_{i' \in C^{-1}(i)} \frac{1}{|C^{-1}(i)|} = b_i$$

for all i. Hence $P\boldsymbol{b}' = \boldsymbol{b}$, and therefore $\boldsymbol{x}$ is a feasible solution to L.

Further calculations show that if $\boldsymbol{x}$ is an optimal solution to L then $\boldsymbol{x}' = Q^*\boldsymbol{x}$ is an optimal solution to L$'$ and, conversely, if $\boldsymbol{x}'$ is an optimal solution to L$'$ then $\boldsymbol{x}' = Q^*\boldsymbol{x}$ is an optimal solution to L (see [15] for details).

What this all means is that for a given linear program L, color refinement gives us a potentially much smaller linear program L$'$ and two linear mappings Q^* and Q transforming feasible solutions to L into feasible solutions to L$'$ and vice versa and preserving optimality. Thus instead of solving L directly, we can first compute L$'$, then solve it and transform the solution back to a solution to L using Q. Experiments [15] have shown that for linear programs with many regularities, this often yields a substantial speed-up in total processing time.

7 Application: Weisfeiler-Leman Graph Kernels

Kernels are a concept that can be used to perform pattern analysis, a basic task in machine learning. Normally, in pattern analysis algorithms, each object is assigned a vector in a high-dimensional feature space and the similarity of two vectors arising is interpreted measuring the similarity the objects the vectors were generated from. In contrast to this general approach, a kernel is an implicitly given similarity measure, typically as a scalar product of the corresponding feature vectors. A crucial point (often referred to as the kernel trick) is that kernel methods can avoid the explicit computation of the feature vector and directly evaluate the similarity value of a pair of objects. With such a similarity measure at hand, it is then possible to perform tasks such as classification and clustering using the by now well developed *kernel methods* from machine learning (see [17]). Among these methods are techniques like support vector machines, kernel regression, and principal component analysis.

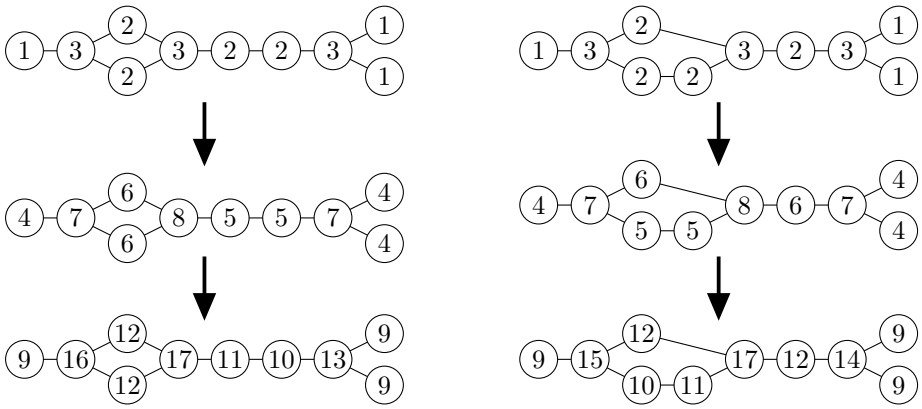
An important and growing branch of machine learning deals with *graph-structured data*. Here data comes in form of annotated (that is edge- and/or vertex-labeled) graphs. *Graph Kernels* constitute the application of said kernel methods to graph structure data. While early research focused on similarity of vertices within one given graph [21], the focus moved to comparing graphs to each other [10, 36]. Over time, numerous graph kernels have been considered. We refer to [36] for a unified treatment. Most of these kernels have in common that the counts of some form of substructure (e.g., subgraphs, walks between paths, etc.) are used to compile the feature vector. Of course, whether two graphs should be considered similar always depends on the specific application. However, experiments show that counting such substructures yields satisfactory results for the desired pattern analysis tasks.

A major drawback of the kernel methods that count such substructures is their excessive running time. In fact, they do not scale particularly well to larger inputs. This is where color-refinement and the Weisfeiler-Leman Graph Kernel come into play. For an integer h, the *number of iterations*, the *Weisfeiler-Leman Graph Kernel* assigns to every pair of graphs G and G' a number $k^h(G, G')$ as follows. As before we let C_i be the coloring after the i-th round of color refinement performed on the disjoint union $G \uplus G'$.

Then we define

$$k^h(G, G') = \sum_{v \in V(G)} \sum_{v' \in V(G)} \delta(C_1(v), C_1(v')) + \delta(C_2(v), C_2(v')) + \ldots + \delta(C_h(v), C_h(v')),$$

where $\delta(a, b)$ is 1 if $a = b$ and 0 otherwise. Thus, we count the number of vertex pairs from the two graphs of equal color and do so for each of the first h iterations. Figure 6 shows an example. In this example, in order to keep color names short, color in the later iterations are repeatedly renamed to previously unused integers.



histogram	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
G	3	4	3	3	2	2	2	1	3	1	1	2	1	0	0	1	1
G'	3	4	3	3	2	2	2	1	3	1	1	2	0	1	1	0	1

Figure 6. The figure shows the computation of the implicit feature vectors of two graphs G (left) and G' (right) in the Weisfeiler-Leman Graph Kernel. The figure also shows the histograms of colors that appear in the two graphs highlighting their similarity.

Experiments show that the information collected by the kernel is sufficient to adequately perform desired classification tasks [34]. These experiments also indicate that choosing the number of iterations as $h \approx 5$ is best. Indeed, since almost all graphs are distinguished by color refinement, performing too many iterations means that the colors contain too much (global) information about the graphs they are situated in and cannot be used as a meaningful measure of similarity. (In fact, two graphs will most likely not share any vertices of the same color if the number of rounds is too large.)

As with the other applications for color refinement, a major advantage of the methods is its running time. Since an iteration of color refinement can be performed in time $O(m)$, to compute the Weisfeiler-Leman Graph Kernel that uses h iterations on a pair of graphs on m edges, we require time $O(mh)$. However, in applications, we intend to apply the kernel between all pairs of graphs coming from a large set of examples, of size N say. A crucial observation is that instead of performing color refinement on the disjoint union of the two graphs we can also perform it seperately on each graph. Doing so, we cannot simply rename the new arising colors, since this might be inconsistent between graphs. However, using a suitable hash function we can remedy the situation. We obtain a running time of $O(Nhm + N^2hn)$ to compute the kernels between every pair of graphs from a set of size N. Here the first summand arises from color refinement on N graphs, while the second summand is the computation of the scalar product for each of the N^2 pairs of graphs.

It is possible to combine the concept of the Weisfeiler-Leman Graph Kernel with other kernels in a more general framework. We refer to [34] for details. With this general applicability and due to its fast computation time, the Weisfeiler-Leman Graph Kernel has found its application in various machine learning areas. For example it has been applied for Malware detection [31] in the context of code similarity [23], fMRI analysis [11] and Resource Description Framework data [30]. While the Kernel itself hinges on the annotation of given the data (i.e., the labels) coming from a discrete domain, recent work [27] shows how randomization and hashing can be applied to transform the seemingly intrinsic discrete kernel into a kernel suitable for continuous data.

8 Conclusions

In this chapter, we reviewed the basic color refinement algorithm and explained how it can be implemented in quasilinear time. In particular, we connected it to lifted inference and graph kernels.

References

[1] A. Atserias and E. Maneva. Sherali–Adams relaxations and indistinguishability in counting logics. *SIAM Journal on Computing*, 42(1):112–137, 2013.

[2] L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing (STOC '16)*, pages 684–697, June 2016.

[3] L. Babai, P. Erdös, and S. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9:628–635, 1980.

[4] L. Babai and L. Kučera. Canonical labelling of graphs in linear average time. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 39–46, 1979.

[5] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[6] C. Berkholz, P. Bonsma, and M. Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory of Computing Systems*, doi:10.1007/s00224-016-9686-0, 2016.

[7] B. Bollobás. Distinguishing vertices of random graphs. *Annals of Discrete Mathematics*, 13:33–50, 1982.

[8] J. Cai, M. Fürer, and N. Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12:389–410, 1992.

[9] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85 – 98, 1982.

[10] Thomas Gärtner, Peter A. Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *COLT*, volume 2777 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2003.

[11] Katerina Gkirtzou, Jean Honorio, Dimitris Samaras, Rita Z. Goldstein, and Matthew B. Blaschko. fMRI analysis with sparse weisfeiler-lehman graph statistics. In *MLMI*, volume 8184 of *Lecture Notes in Computer Science*, pages 90–97. Springer, 2013.

[12] M. Grohe. Descriptive complexity, canonisation, and definable graph structure theory. Manuscript available at `http://www.lii.rwth-aachen.de/de/mitarbeiter/13-mitarbeiter/professoren/39-book-descriptive-complexity.html`.

[13] M. Grohe. Fixed-point definability and polynomial time on graphs with excluded minors. *Journal of the ACM*, 59(5), 2012.

[14] M. Grohe, K. Kersting, M. Mladenov, and E. Selman. Dimension reduction via colour refinement. *ArXiv (CoRR)*, abs/1307.5697, 2013.

[15] M. Grohe, K. Kersting, M. Mladenov, and E. Selman. Dimension reduction via colour refinement. In A. Schulz and D. Wagner, editors, *Proceedings of the 22nd Annual European Symposium on Algorithms*, volume 8737 of *Lecture Notes in Computer Science*, pages 505–516. Springer-Verlag, 2014.

[16] M. Grohe and M. Otto. Pebble games and linear equations. *Journal of Symbolic Logic*, 80(3):797–844, 2015.

[17] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *Ann. Statist.*, 36(3):1171–1220, 2008.

[18] J.E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[19] N. Immerman and E. Lander. Describing graphs: A first-order approach to graph canonization. In A. Selman, editor, *Complexity theory retrospective*, pages 59–81. Springer-Verlag, 1990.

[20] R.M. Karp. Reducibilities among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[21] Risi Kondor and John D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, pages 315–322. Morgan Kaufmann, 2002.

[22] J.B. Lasserre. An explicit equivalent positive semidefinite program for nonlinear 0-1 programs. *SIAM Journal on Optimization*, 12(3):756–769, 2002.

[23] Wenchao Li, Hassen Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. Detecting similar programs via the weisfeiler-leman graph kernel. In *ICSR*, volume 9679 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2016.

[24] L. Lovász and L. Schrijver. Cones of matrices and set-functions and 0–1 optimization. *Cones of Matrices and Set-Functions and 0–1 Optimization SIAM Journal on Optimization*, 1(2):166–190, 1991.

[25] P. Malkin. Sherali–adams relaxations of graph isomorphism polytopes. *Discrete Optimization*, 12:73–97, 2014.

[26] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[27] Christopher Morris, Nils M. Kriege, Kristian Kersting, and Petra Mutzel. Faster kernels for graphs with continuous attributes via hashing. *CoRR*, abs/1610.00064, 2016.

[28] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[29] R.C. Read and D.G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[30] Petar Ristoski and Heiko Paulheim. RDF2Vec: RDF graph embeddings for data mining. In *International Semantic Web Conference (1)*, volume 9981 of *Lecture Notes in Computer Science*, pages 498–514, 2016.

[31] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *EISIC*, pages 141–147. IEEE Computer Society, 2012.

[32] E. Selman. On fractional isomorphisms, 2013. Diplomarbeit at the Department of Mathematics, Humboldt-Universität zu Berlin.

[33] H. D. Sherali and W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3(3):411–430, 1990.

[34] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[35] G. Tinhofer. A note on compact graphs. *Discrete Applied Mathematics*, 30:253–264, 1991.

[36] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.