Isomorphism test for digraphs with weighted edges

Adolfo Piperno

Dipartimento di Informatica, La Sapienza Università di Roma, Via Salaria 113, I-00198 Rome (Italy)
piperno@di.uniroma1.it

Abstract

Colour refinement is at the heart of all the most efficient graph isomorphism software packages. In this paper we present a method for extending the applicability of refinement algorithms to directed graphs with weighted edges. We use `Traces` as a reference software, but the proposed solution is easily transferrable to any other refinement-based graph isomorphism tool in the literature. We substantiate the claim that the performances of the original algorithm remain substantially unchanged by showing experiments for some classes of benchmark graphs.

1998 ACM Subject Classification G.2.2 Graph Theory, G.4 Mathematical Software

Keywords and phrases Practical Graph Isomorphism, Weighted Graphs, Partition Refinement

Digital Object Identifier 10.4230/OASIcs...

1 Introduction

An *isomorphism* between two graphs is a bijection between their vertex sets that preserves adjacency. An *automorphism* is an isomorphism from a graph to itself. The set of all automorphisms of a graph G form a group under composition called the *automorphism group* $\mathrm{Aut}(G)$ whose *order* is $|\mathrm{Aut}(G)|$. The graph isomorphism problem (GI) is that of determining whether there is an isomorphism between two given graphs. It is convenient to consider GI for vertex coloured graphs, in which case isomorphisms and automorphisms must preserve colours of vertices.

In this paper we will consider GI for coloured graphs and digraphs with weighted edges, in which case isomorphisms and automorphisms must preserve weights of edges, too. Quite surprisingly, none of the existing GI software packages is currently able to treat such class of graphs directly. Existing software can handle weighted digraphs by using layers (as in the `nauty` manual [16]) or by using unweighted gadgets to simulate weighted directed edges (see Figure 1). However, both methods multiply the size of the graph and so increase the running time and space significantly. We will use `Traces` [15, 19] as reference program, but the method that we are going to describe can be adapted to any other GI software.

The most successful GI packages are based on the *individualization-refinement* technique: they can treat graphs with a huge number of vertices and edges quite efficiently. During the computation, these programs spend most of the time in the operation of *colour refinement*, i.e. in the assignment of a minimal numer of colours to vertices of the graph, in a way that vertices with the same colour have neighbours with the same colours. In every GI package, the refinement routines have been the object of subsequent optimizations, sometimes over decades: to add new features to them may not be an easy task.

From their part, refinement algorithms spend most of the time in counting neighbours of vertices. At each iteration, a reference colour c is selected and vertices of the graph are classified according to the number of c-coloured neighbours they have.

In the case of graphs with weighted directed edges – and in the context of graph isomorphism – the main issue to be considered is that the notion of adjacency is not as immediate as in the case of simple graphs. In our setting, the classification of a vertex u must take into account not only weights of edges from u to vertices with the current reference colour, but also weights of their opposite edges. The main motivation of this paper is to go beyond these additional difficulties by keeping the counting mechanism of the refinement algorithm for simple graphs. Shortly, the solution we propose is the adoption of *internal weights* – to be used during the computation, only – which encode the information from both the weights of an edge and its opposite edge.

In particular, it is our aim to show (and prove) that the ability to process weighted graphs and digraphs can be added to `Traces` in a very simple and conservative way: (i) by changing a minimal number of lines of the existing code; (ii) by introducing a negligible overhead – with respect to the whole computation – in preprocessing weights, just in the case of the new families of graphs; (iii) by preserving substantially the same performances in the case of simple graphs. The simplicity of the proposed solution stems from the fact that it exactly captures the additional complexities arising when using graphs with weighted edges.

Towards the aim of the paper, in Section 2 we will briefly review the individualization-refinement technique and we will consider the issues in extending the method to the case of weighted digraphs; in Section 3 we will introduce internal weights, and we will prove their properties. The new algorithm will be presented in Section 4, together with a brief analysis of its complexity. Experimental results will be shown in Section 5. The details of the algorithm for assigning internal weights will be presented in Appendix.

2 Practical aspects of the graph isomorphism problem

The theoretical status of GI, which culminates with Babai's recent quasi-polinomiality result [2], is outside the scope of this paper; a brief historical description can be found in [15].

From a practical perspective, the most successful approach to GI is the "individualization-refinement" method, which originates in [18, 5, 1] and was distributed in a software package, `nauty`, by McKay [13].

Basically, a *colouring (or partition) refinement* function classifies vertices of a graph G, in a way which is invariant under isomorphism, according to the classification of their neighbours. The sets of vertices with a given colour are called *cells*. Vertices with a specific colour (chosen in an isomorphism invariant way, again) are *individualized* one by one in order to distinguish them from other vertices in the same cell. This mechanism produces a search tree, whose nodes represent refined colourings, while branching is determined by the individualization step. Colourings in which all cells are singletons (called *discrete*) appear as leaves of the search tree. Equivalent discrete colourings induce automorphisms of the graph G. Pruning of the tree is obtained by excluding non-matching colourings and by the use of automorphisms. Comparing colourings also allows us to define a *best* leaf, which is used to canonically label the graph G, namely to produce a representative of exactly the isomorphism class of G.

Software distributions based on the individualization-refinement technique such as `nauty`[13, 15, 14, 16], `Traces`[15, 14, 16, 19], `Bliss`[9, 10], `conauto`[11, 12] and `saucy` [6, 7] are the most efficient GI tools currently available, though Neuen and Schweitzer [17] have recently tailored classes of graphs which are not tractable by them.

2.1 Graphs and colourings

A *weighted digraph* is a triple $G = (V, E, w)$, where $(v, v) \notin E$ and $(u, v) \in E \Rightarrow (v, u) \in E, \forall u, v \in V$; $w : E \to \mathcal{W}$ is a function mapping arcs to elements of a finite set $\mathcal{W}$ of possible *weights*. Throughout the paper, we will assume without loss of generality that $\mathcal{W} \subseteq \mathbb{N}$ is a finite set of natural numbers. Note that loops can be easily represented in our setting by colouring vertices.

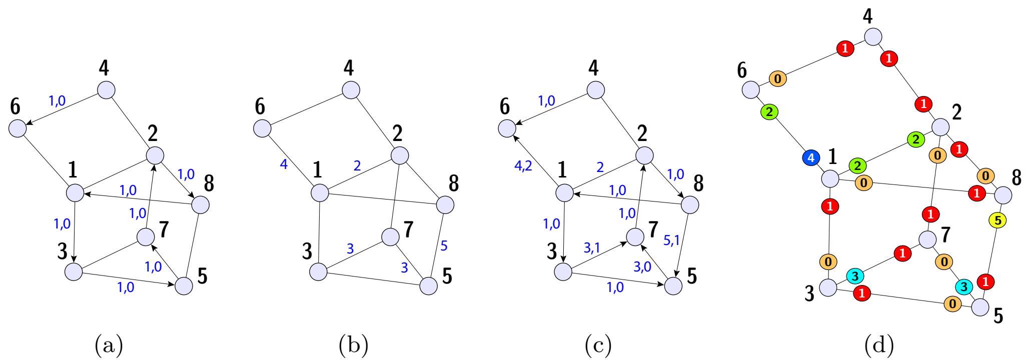


Figure 1 A directed graph (a), a weighted graph (b), a weighted digraph (c) and its encoding by means of a simple coloured graph (d). In these pictures, the arc (u, v) has label "a, b" when the weight of (u, v) is a and the weight of (v, u) is b. The simple edge (u, v) has label "a" when both the weight of (u, v) and that of (v, u) are $a \neq 0, 1$. The simple edge (u, v) has no label when both the weight of (u, v) and that of (v, u) are 1. The graph in (d) is obtained from the one in (c) by adding two vertices for each arc and colouring them according to their weight.

Let $\mathcal{G} = \mathcal{G}_n$ denote the set of graphs with vertex set $V = \{1, 2, \ldots, n\}$ A *colouring* of V (or of $G \in \mathcal{G}$) is a surjective function π from V onto $\{1, 2, \ldots, k\}$ for some k. The number of colours, i.e. k, is denoted by $|\pi|$. A *cell* of π is the set of vertices with some given colour. A *discrete colouring* is a colouring in which each cell is a singleton, in which case $|\pi| = n$. Note that a discrete colouring is a permutation of V.

If π, π' are colourings, then π' is *finer than or equal to* π, written $\pi' \preceq \pi$, if $\pi(v) < \pi(w) \Rightarrow \pi'(v) < \pi'(w)$ for all $v, w \in V$. (This implies that each cell of π' is a subset of a cell of π, but the converse is not true.)

A pair (G, π), where π is a colouring of G, is called a *coloured graph*.

Let S_n denote the symmetric group acting on V. We indicate the action of elements of S_n by exponentiation. That is, for $v \in V$ and $g \in S_n$, v^g is the image of v under g. The same notation indicates the induced action on complex structures derived from V. In particular, if $G = (V, E, w) \in \mathcal{G}$, then: (i) $G^g \in \mathcal{G}$ has u^g adjacent to v^g exactly when u and v are adjacent in G; (ii) if π is a colouring of V, then π^g is the colouring with $\pi^g(v^g) = \pi(v)$ for each $v \in V$; (iii) w^g is such that $w^g(u^g, v^g) = w(u, v)$, for each $(u, v) \in E$; (iv) $(G, \pi)^g = ((V^g, E^g, w^g), \pi^g)$.

2.2 Graph isomorphism

Two coloured graphs $(G = (V, E, w), \pi), (G' = (V, E', w'), \pi')$ are *isomorphic* if there is $g \in S_n$ such that $(G', \pi') = (G, \pi)^g$, in which case we write $(G, \pi) \cong (G', \pi')$. Such a g is called an *isomorphism*. The *automorphism group* $\mathrm{Aut}(G, \pi)$ is the group of isomorphisms of the coloured graph (G, π) to itself; that is,

$$\mathrm{Aut}(G, \pi) = \{g \in S_n : (G, \pi)^g = (G, \pi)\}.$$

Let $\mathit{\Pi} = \mathit{\Pi}_n$ denote the set of colourings. A *canonical form* is a function

$$C : \mathcal{G} \times \mathit{\Pi} \to \mathcal{G} \times \mathit{\Pi}$$

such that, for all $G \in \mathcal{G}$, $\pi \in \mathit{\Pi}$ and $g \in S_n$,

$$C(G, \pi) \cong (G, \pi) \text{ and } C(G^g, \pi^g) = C(G, \pi). \tag{1}$$

In other words, it assigns to each coloured graph an isomorphic coloured graph that is a unique representative of its isomorphism class. It follows from the definition that $(G, \pi) \cong (G', \pi') \Leftrightarrow C(G, \pi) = C(G', \pi')$.

2.3 Refinement

We first review and discuss refinement for simple graphs.

▶ **Definition 1** (the simple graph case)**.** Let $G \in \mathcal{G}$ be a simple graph.

1. A colouring of G is called *equitable* if any two vertices of the same colour are adjacent to the same number of vertices of each colour.
2. For every colouring π of G, a coarsest equitable colouring π' finer than π is called a *(colour) refinement* of π. It is well known that π' is unique up to the order of its cells.

An algorithm for computing π' appears in [13]. We summarize it in Algorithm 1. All refinement algorithms present in the literature are variants of this one. The paper of Berkholz, Bonsma and Grohe [3] has recently presented a deep analysis of refinement algorithms, establishing their complexity in $O((m + n) \log n)$ time, where n is the number of vertices and m the number of edges of the input graph.

Algorithm 1 Refinement algorithm.

```
Data: $\pi$ is the input colouring and $\alpha$ is a sequence of some cells of $\pi$.
Result: The final value of $\pi$ is the output colouring.
while $\alpha$ is not empty and $\pi$ is not discrete do
    Remove some element $W$ from $\alpha$;
    Count the number of edges from vertices in $W$ to each vertex;
    for each cell $X$ of $\pi$ do
        Let $X_1, \ldots, X_k$ be the fragments of $X$ distinguished according
          to the counting of the previous step;
        Replace $X$ by $X_1, \ldots, X_k$ in $\pi$;
        if $X \in \alpha$ then
            Replace $X$ by $X_1, \ldots, X_k$ in $\alpha$;
        else
            Add all but one of the largest of $X_1, \ldots, X_k$ to $\alpha$;
        end
    end
end
```

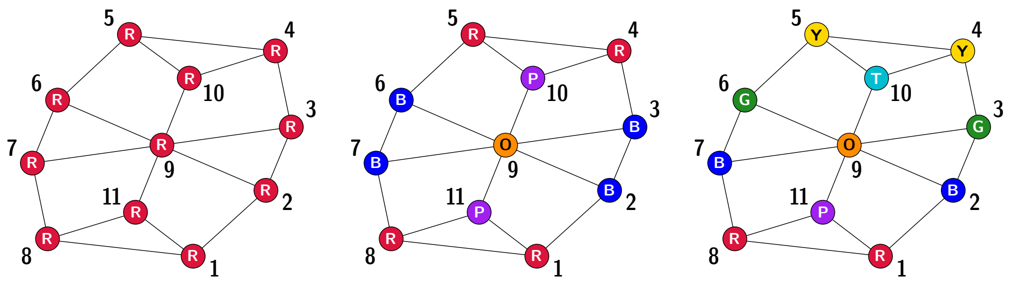


Figure 2 Refinement (simple graphs); individualization of vertex 10 and refinement (right).

▶ **Example 2.** A simple graph (left) and its colour refinement are shown in Figure 2. The rightmost colouring is obtained, by refining, after the individualization of vertex 10.

In Algorithm 1, the cell W causes the splitting of the cell X when two vertices in X have a different number of neighbours in W. We will call W the *reference cell*. The correctness of the algorithm is based on the fact that – at every iteration of the **while** loop – the sequence α contains at least cells which may cause any possible splitting of other cells. In particular, when the refinement function is called at the beginning of the computation, the sequence of all cells of the input colouring is assigned to α, while after an individualization step it is sufficient to refine the colouring by assigning to α only the cell which contains the individualized vertex.

All the programs based on the individualization-refinement method spend most of their time in refining partitions; for its part, the refinement algorithm spends most of its time in counting neighbours of the reference cell. Therefore, the overall efficiency of the algorithm depends to a large degree on the efficiency of the colour refinement procedure. `Traces`, for instance, distinguishes several cases in the counting loop, according to different rates of density of the graph, and gives priority to singleton reference cells.

In this scenario, it is our aim to equip `Traces` with the additional resources needed to treat weighted graphs, without making any change to the neighbour counting algorithms. The main issue to be considered is that a cell must be split not only in conformity with the number of its outgoing edges falling into the reference cell, but also according to the weight of such edges and to the weight of their opposite edges (see e.g. the cell $\{2, 3, 6, 7\}$ in Figure 3).

▶ **Definition 3** (the weighted digraph case). Let $G = (V, E, w) \in \mathcal{G}$ be a weighted digraph with weights from $\mathcal{W} \subseteq \mathbb{N}$.

1. Let $u, v \in V$ be two distinct vertices of G. We say that u is (a, b)-*adjacent* to v if $w(u, v) = a$ and $w(v, u) = b$ and a, b are not both equal to 0. Therefore, if u is (a, b)-adjacent to v, then v is (b, a)-adjacent to u.
2. A colouring of G is called *equitable* if any two vertices of the same colour are (a, b)-adjacent to the same number of vertices of each colour, for any $(a, b) \in \mathcal{W} \times \mathcal{W}$.
3. For every colouring π of G a coarsest equitable colouring π' finer than π is called a *refinement* of π.

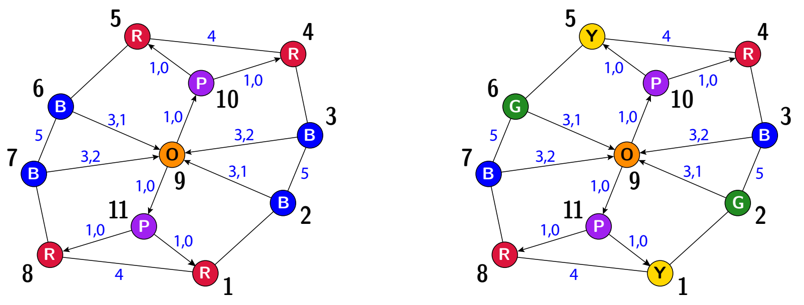


Figure 3 Refinement (weighted digraphs): the splitting of the cell $\{2,3,6,7\}$ into $\{2,6\}\{3,7\}$ is caused by the reference cell $\{9\}$, due to the weights of the edges *from* vertex 9 to $2,3,6,7$.

3 Internal weights

Let $G=(V,E,w)\in\mathcal{G}$ be a weighted digraph with weights from $\mathcal{W}\subseteq\mathbb{N}$. We assign *internal weights* to edges of G, with the aim of making the refinement phase similar as much as possible to that for simple graphs. We will prove that the order of the automorphism group of G with internal weights remains unchanged, and that a canonical form of G can be obtained at the end of the computation simply by restoring the original weights.

▶ **Definition 4.** We define the function $\overline{w}$ which assigns *internal weights* to edges of G in two steps:

1. We define the function

$$\begin{aligned}\phi_w\colon E&\to\mathcal{W}\times\mathcal{W}\\ (u,v)&\mapsto(w(u,v),w(v,u)).\end{aligned}\tag{2}$$

and we denote by $\Phi_w=\{(w(u,v),w(v,u))\mid(u,v)\in E\}$ the image of ϕ_w and by Φ_w^{lex} the lexicographically ordered sequence of elements of Φ_w.

2. Let $\overline{\mathcal{W}}=\{0,1,\dots,|\Phi_w|-1\}$. We define the function

$$\begin{aligned}&\overline{w}\colon E\to\overline{\mathcal{W}}\\ &(u,v)\mapsto\text{the index of }\phi_w(u,v)\text{ in }\Phi_w^{\text{lex}}\text{ (starting from 0).}\end{aligned}\tag{3}$$

3. For any $G=(V,E,w)\in\mathcal{G}$, we denote $\overline{G}=(V,E,\overline{w})$.

▶ **Example 5.** In Figure 4, internal weights are assigned to the leftmost graph. For any edge (u,v) in the second column of the table, the corresponding entry `a,b` $\to$ `i` in the first column shows that $w(u,v)=$ `a`, $w(v,u)=$ `b` and $\overline{w}(u,v)=$ `i`. Therefore, the internal weight `i` carries the information of both the weights of (u,v) and (v,u).

For any pair of edges (u_1,v_1) and (u_2,v_2)

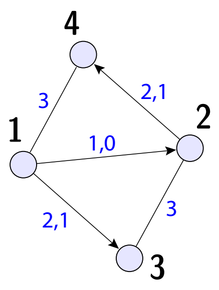
$$\text{(By (2))}\qquad \phi_w(u_1,v_1)=\phi_w(u_2,v_2)\Leftrightarrow\phi_w(v_1,u_1)=\phi_w(v_2,u_2)\tag{4}$$

$$\text{(By (3))}\qquad \overline{w}(u_1,v_1)=\overline{w}(u_2,v_2)\Leftrightarrow\phi_w(u_1,v_1)=\phi_w(u_2,v_2),\tag{5}$$

therefore the internal weight of an edge encodes both $w(u,v)$ and $w(v,u)$.

▶ **Lemma 6.** *Let $G=(V,E,w)\in\mathcal{G}$. Then*

1. $\phi_w(u_1,v_1)=\phi_w(u_2,v_2)\Leftrightarrow\phi_{\overline{w}}(u_1,v_1)=\phi_{\overline{w}}(u_2,v_2)$.



weights → index	edges
0,1 → 0	(2,1)
1,0 → 1	(1,2)
1,2 → 2	(3,1),(4,2)
2,1 → 3	(1,3),(2,4)
3,3 → 4	(1,4),(2,3),(3,2),(4,1)

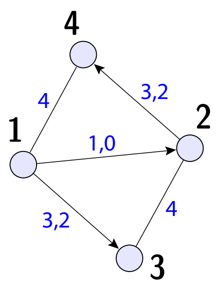


Figure 4 Assignment of internal weights.

2. $\phi_{\overline{w}}(u_1, v_1) = \phi_{\overline{w}}(u_2, v_2) \Leftrightarrow \phi_{\overline{w}}(v_1, u_1) = \phi_{\overline{w}}(w_2, u_2)$.

Proof. These follow from (4) and (5). ◀

▶ Remark. (Idempotence) For any $G = (V, E, w) \in \mathcal{G}$ we have $\overline{G} = \overline{\overline{G}}$.

In fact, by Lemma 6.2, for every $a \in \overline{\mathcal{W}}$ there is one and only one $b \in \overline{\mathcal{W}}$ such that $\phi_{\overline{w}}(u, v) = (a, b)$, for some $(u, v) \in E$. It follows that $\overline{\mathcal{W}} = \overline{\overline{\mathcal{W}}}$ and that the index of (a, b) in $\Phi^{\text{lex}}_{\overline{w}}$ is exactly a. Thus, $\overline{w} = \overline{\overline{w}}$. Note that $(a, b) \in \Phi_{\overline{w}} \Leftrightarrow (b, a) \in \Phi_{\overline{w}}$, therefore the set of pairs $\Phi_{\overline{w}}$ is a bijection on $\overline{\mathcal{W}}$.

▶ **Theorem 7.** *Let $G, G_1, G_2 \in \mathcal{G}$. Then:*
1. $\text{Aut}(G) = \text{Aut}(\overline{G})$.
2. $G_1 \cong G_2 \Rightarrow \overline{G_1} \cong \overline{G_2}$.

Proof. Both **1** and **2** follow from Lemma 6.1.
1. $(\Rightarrow)$ Let $g \in \text{Aut}(G)$ be such that for some vertices u_1, v_1, u_2, v_2 we have $(u_1, v_1)^g = (u_2, v_2)$. Then $\phi_w(u_1, v_1) = \phi_w(u_2, v_2)$, since g preserves weights. By Lemma 6.1 we obtain that $g \in \text{Aut}(\overline{G})$. The converse implication is proven similarly.
2. It is well known that we can decide the isomorphism of two graphs by comparing the order of their automorphism groups with the order of the automorphism group of their union graph. The theorem follows considering the union graph of G_1 and G_2, applying the previous result.

◀

▶ Remark. The converse of Theorem 7.2 does not hold. A simple example can be derived as a consequence of the idempotence property. In fact, $\overline{G} = \overline{\overline{G}} \not\Rightarrow G \cong \overline{G}$. Consider as a further counterexample the graph G in Figure 4 (left), and replace weight 2 with 3 in all its occurrences, thus obtaining a graph G' not isomorphic to G. However, $\overline{G} = \overline{G'}$.

4 Refinement and isomorphism test

The use of internal weights allows refinements of weighted digraphs to be computed with an algorithm only slightly different from Algorithm 1, as shown in Algorithm 2. The counting loop is fractioned according to the internal weights of outgoing edges of elements of the reference cell W.

▶ **Theorem 8.** *(Correctness)*
1. Given an internally weighted digraph G and a colouring π of G, the output colouring of Algorithm 2 is a refinement of π.

Algorithm 2 Refinement algorithm for weighted digraphs.

```
Data: $\pi$ is the input colouring and $\alpha$ is a sequence of some cells of $\pi$.
Result: The final value of $\pi$ is the output colouring.
while $\alpha$ is not empty and $\pi$ is not discrete do
    Remove some element $W$ from $\alpha$;
    for each internal weight $z$ (in ascending order) of outgoing arcs of vertices in $W$ do
        Count the number of edges with weight $z$ from vertices in $W$ to each vertex;
        for each cell $X$ of $\pi$ do
            Let $X_1, \ldots, X_k$ be the fragments of $X$ distinguished according
             to the counting of the previous step;
            Replace $X$ by $X_1, \ldots, X_k$ in $\pi$;
            if $X \in \alpha$ then
                Replace $X$ by $X_1, \ldots, X_k$ in $\alpha$;
            else
                Add all but one of the largest of $X_1, \ldots, X_k$ to $\alpha$;
            end
        end
    end
end
```

2. *In the case of simple graphs, Algorithm 2 coincides with Algorithm 1.*

Proof. 1. The proof follows the pattern of any similar proof in the literature, see e.g. [3]. In a nutshell, (i) the resulting colouring is as coarse as possible since any cell splitting executed by the algorithm is necessary; (ii) it is also sufficiently fine. In fact assume, towards a contradiction, that the final colouring has two cells W_1 and W_2 such that two vertices $u, v \in W_1$ have a different number of (a, b)-neighbours in W_2, for some internal weights a, b. This is impossible if W_2 is present in the sequence α at the beginning of the computation. Therefore W_2 must have been derived by the splitting of some other cell W. Assume W_2 is not one the largest cells coming from splitting W. In this case, W_2 is added to α and subsequently removed from it, thus causing u and v to be distributed into two different subcells of W_1. Otherwise, if W_2 is not added to α after splitting W, then the remaining subcells of W – which are all added to α – cause the same splitting of W_2 (this is a classical result by Hopcroft [8]).

2. In the case of simple graphs, we can assume that only one weight is present. Therefore the highlighted loop in Algorithm 2 consists of only one iteration.

◀

4.1 Invariance by isomorphism and preprocessing

Let $G = (V, E, w) \in \mathcal{G}$ and let π be the initial colouring of G. Weights are chosen in the added loop of Algorithm 2 in ascending order, since this choice is invariant under isomorphism. In order to make the new algorithm easily usable in `Traces`, for each vertex v of G we consider the ordered sequence σ_v of internal weights of its outgoing edges and we store the neighbours of v according to this ordering. In addition, we denote $\sigma_V = \{\sigma_v \mid v \in V\}$ and we refine the colouring π by splitting each cell according to the lexicographic order of elements of σ_V. We observe that in a simple graph the counterpart of this splitting operation is the degree colouring, since in that case sequences in σ_V only differ in their length. At the end of this kind of preprocessing phase, if two vertices u and v appear in the same cell, then $\sigma_u = \sigma_v$ and internal weights of neighbours of u and v will immediately emerge in ascending order in the weight loop of Algorithm 2.

▶ **Example 9.** In Figure 5, the colouring of the leftmost graph is determined conforming to the ordering of σ_V. Two vertices with the same colour, e.g 4 and 5, are such that $\sigma_4 = \sigma_5 = (0, 2, 7)$. We observe that the colouring is not equitable. In fact, 5 has 6 as neighbour, but 4, which appears in the same cell of 5, has no neighbour in the cell of 6. The cell $\{1, 4, 5, 8\}$ is split into $\{1, 5\}\{4, 8\}$ during the execution of Algorithm 2 as soon as the cell $\{2, 6\}$ is removed from α. More precisely, the splitting occurs when considering outgoing edges of elements of the cell $\{2, 6\}$ whose internal weight is 2. The result of the splitting operation is shown in the rightmost graph, whose colouring is equitable.

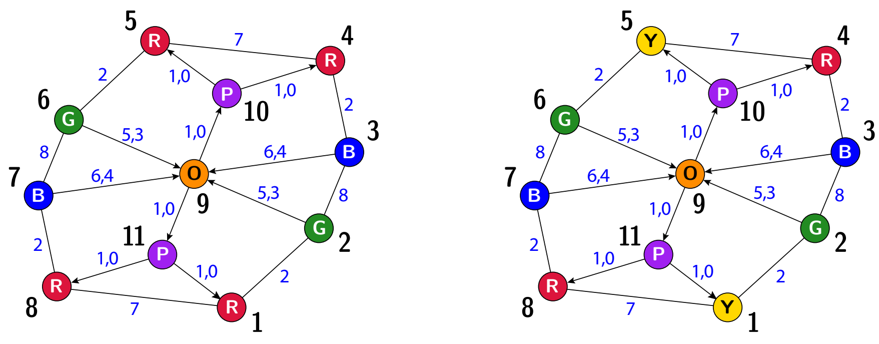


Figure 5 σ_V-sequence colouring and refinement.

4.2 The new GI algorithm: analysis

Algorithm 3 GI algorithm.

```
Data: A coloured graph $(G = (V, E, w), \pi)$.
Result: The order of $\mathrm{Aut}(G, \pi)$ and the canonical labelling $C(G, \pi)$ of $(G, \pi)$.
1   Compute internal weights of $G$;
2   Make a copy of $(V, E)$;
3   Preprocess the new graph $G' = (V, E, \overline{w})$ by sorting the neighbours of each $v \in V$
    according to the sequence $\sigma_v$ and by considering the $\sigma_V$-refinement of $\pi$;
4   Split cells of $\pi$ according to the order of vertices induced by the order of 5$\sigma_V$. Run
    WTraces (namely, Traces with Algorithm 2 in place of Algorithm 1)
6   Restore the original weights in the canonical labelling $C(G', \pi)$: if $p$ is the permutation
    such that $C(G', \pi) = (G'^p, \pi^p)$, take $C(G, \pi) = (G^p, \pi^p)$.
```

Let $(G = (V, E, w), \pi)$ be a coloured graph with $n = |V|$ and $m = |E|$. Algorithm 3 summarizes the method to compute the order of the automorphism group and the canonical form of a weighted digraph that we have described in the previous sections. We observe that:

1. The computation of internal weights requires $O(n + m)$ time under the (reasonable) assumption that $\forall(u, v) \in E : w(u, v) < m$, $O(n+m \log m)$ time otherwise (see Appendix).

2. To make a copy of (V, E) requires $O(n+m)$ time.
3. The preprocessing phase requires $O(m)$ time for sorting the neighbours of vertices according to internal weights of outgoing edges, and $O(m)$ time to order the sequence σ_V. In fact, a radix sort can be used, which runs in $O(nn')$ time, where n' is the average length of sequences in σ_V. In our setting, $O(nn') = O(m)$ since the length of $\sigma_v \in \sigma_V$ is the outdegree of v.
4. For any colouring, `Traces` always mantains its inverse. Using this information, cells of π can be split in conformity to the ordering of σ_V in $O(n)$ time.

Recalling that the refinement function runs in time $O((m+n)\log n)$ and that $m \leq n(n-1)$, it follows that the additional computational effort of Algorithm 3 with respect to `Traces` is less than (or at least comparable to) one single call of the refinement function.

5 Experimental results

In the following figures, we present some experiments for a variety benchmark graphs. The graphs are taken from `http://pallini.di.uniroma1.it/Graphs.html`.

The times given are for a Macbook Pro with 3.1 GHz Intel i7 processor (16GB of RAM), using the LLVM compiler (version 9.0.0) and running in a single thread. The interested reader will find the binary codes at `http://pallini.di.uniroma1.it/Weights.html`, toghether with several other families of graphs.

We recall that `Traces` always computes the order and generators of the automorphism group of the input graph. At the user's request, it computes the canonical form of the graph, too.

Easy graphs are processed multiple times to give more precise times. We usually start from the unit partition, except when specified in the pictures.

The execution of experimental tests with the assigninment of random weights to arcs of graphs from some known relevant families does not give interesting benchmarks since the weight assignment usually breaks all the symmetries of the graph. In order to produce meaningful experiments, for each considered graph G, a weighted version G_w of G is built as follows: we consider the initial refined partition of G, say $(W_1, \ldots, W_m)$, and to each arc (u, v) of G we assign the weight k if $v \in W_k$. This enables to force the program to consider the input as a weighted digraph, therefore executing all the additional steps described in the paper. Note that the graph G_w has the same automorphism group of G.

Four different experiments are reported:

- 🟦 execution of the currently distributed version of `Traces` (v26r10), with canonical form;
- ▲ execution of `WTraces` (the new program) for a simple graph, with canonical form;
- ♦ execution of `WTraces` adding weights to the input graph, with canonical form;
- ● execution of `WTraces`, without canonical form.

All experiments show that `Traces` and `WTraces` have similar performances for simple unweighted graphs. In particular, plots #1-#3 in Figure 6 show that the extra computational cost becomes negligeable as the number of vertices of the graph increases and (#7) as the graph becomes harder. Plots #4,#7,#10 show the the performance of `WTraces` for weightd digraphs, comparing them to their unweighted version. Due to the presence of the preprocessing overhead, some difference is found for very easy graphs, while the performances are similar for harder cases. The same holds in #5,#8,#11, where the initial colouring of the graph is obtained after individualizing one vertex, thus allowing more weights in the graph G_w.

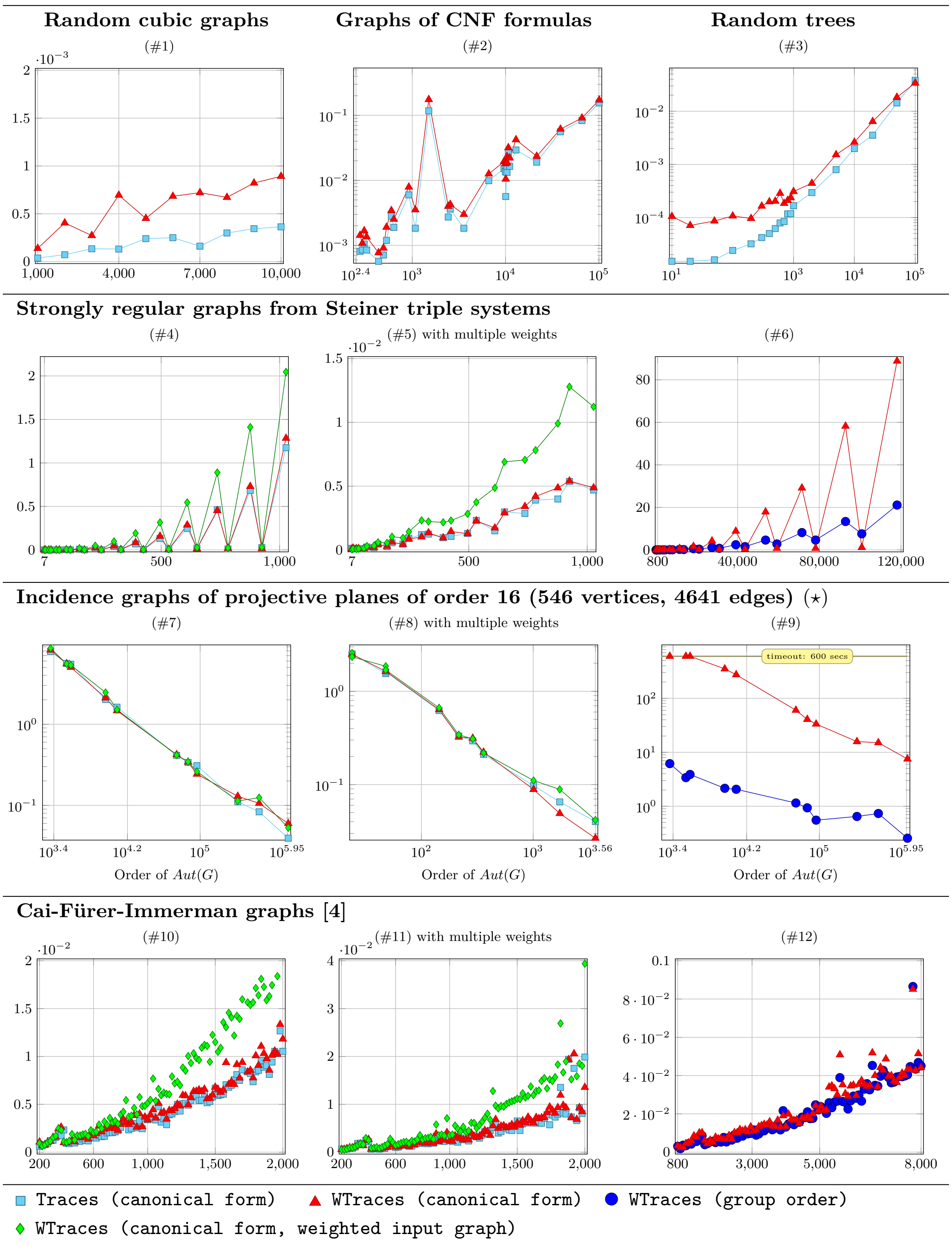


Figure 6 Performance comparison (horizontal: number of vertices (except (⋆)); vertical: time in seconds). (⋆) Incidence graphs of projective planes of order 16 are presented according to the order of their automorphism group.

Finally, plots #6,#9,#12 report the computation time of the simple coloured graphs associated to the weighted digraph reported in plots #4,#7,#10, according to the construction described in Figure 1.(c,d). These plots trivially show that the mentioned construction becomes unfeasible as the density of the graph increases.

6 Concluding remarks

We have presented a method which has enabled us to equip `Traces` with the ability of computing the order of the automorphism group and the canonical labelling of weighted digraphs. The correctness of the method has been proven in the paper. We have executed experimental tests which confirm that the performances of `Traces` remain substantially unchanged. In the case of unweighted digraphs, it would be interesting to compare the behaviour of the presented refinement algorithm with the one in [3]: the notion of (a, b)-adjacency seems to be stronger than the one used by the authors of that paper, since it not only allows for splitting cells according to the number of outgoing edges, but also in conformity with ingoing and undirected edges.

References

1 Arlazarov, V. L., Zuev, I. I., Uskov, A. V. and Faradzev, I. A., An algorithm for the reduction of finite non-oriented graphs to canonical form, Zh. vȳchisl. Mat. mat. Fiz. 14, 737–743, 1974.

2 Babai, L., Graph Isomorphism in Quasipolynomial Time, Preprint 2016. Available at `https://arxiv.org/abs/1512.03547`, 2016.

3 Berkholz, C., Bonsma, P. and Grohe, M., Tight Lower and Upper Bounds for the Complexity of Canonical Colour Refinement", Theory of Computing System 60, 581–614, 2017.

4 Cai, J-Y., Fürer, M, and Immerman, N., An optimal lower bound on the number of variables for graph identification, Combinatorica 12 (4), 389–410, 1992.

5 Corneil, D. G. and Gotlieb, C. C., An efficient algorithm for graph isomorphism, JACM 17, 51–64, 1970.

6 Darga, P. T., Liffiton, M. H., Sakallah, K. A. and Markov, I. L., Exploiting structure in symmetry detection for CNF, In: Proceedings of the 41st Design Automation Conference, 530–534, 2004.

7 Darga, P. T., Sakallah, K. A. and Markov, I. L., Faster Symmetry Discovery using Sparsity of Symmetries, In: Proceedings of the 45th Design Automation Conference, 149–154, 2004.

8 Hopcroft, J., An N Log N Algorithm for Minimizing States in a Finite Automaton, In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, 189–196, Academic Press, 1971.

9 Junttila, T. and Kaski, P., Engineering an efficient canonical labeling tool for large and sparse graphs, In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments and the 4th Workshop on Analytic Algorithms and Combinatorics, 135–149, 2007.

10 Junttila, T. and Kaski, P., Conflict Propagation and Component Recursion for Canonical Labeling, In: Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms, 151–162.

11 López-Presa, J. L. and Fernández Anta, A., Fast algorithm for graph isomorphism testing, In: Proceedings of the 8th International Symposium on Experimental Algorithms, 221–232, 2009.

12 López-Presa, J. L., Fernández Anta, A. and Núñez Chiroque, L., Conauto-2.0: Fast isomorphism testing and automorphism group computation, Preprint 2011. Available at `http://arxiv.org/abs/1108.1060`, 2011.

13 McKay, B. D., Practical graph isomorphism, Congr. Numer. 30, 45–87, 1980.

14 McKay, B. D. and Piperno, A., `nautyTraces`, Software distribution web page, `http://cs.anu.edu.au/~bdm/nauty/` and `http://pallini.di.uniroma1.it/`, 2012.

15 McKay, B. D. and Piperno, A., Practical Graph Isomorphism II, Journal of Symbolic Computation 60, 94–112, 2014.

16 McKay, B. D. and Piperno, A., nauty and Traces User's Guide (Version 2.5), Available at [14], 2012.

17 Neuen, D. and Schweitzer P., Benchmark Graphs for Practical Graph Isomorphism, Preprint 2017. Available at `https://arxiv.org/abs/1705.03686v1`, 2017.

18 Parris, R. and Read, R. C., A coding procedure for graphs, Scientific Report. UWI/CC 10. Univ. of West Indies Computer Centre, 1969.

19 Piperno, A., Search space contraction in canonical labeling of graphs, Preprint 2008–2011. Available at `http://arxiv.org/abs/0804.4881`, 2008.

7 Appendix: assignment of internal weights

The data structures used by `nauty` and `Traces` for representing a weighted digraph G (as described in [16]) are shown in Figure 7, where `nv` is the number of vertices and `nde` is the number of arcs of G; `d` is an array whose i-th position is the degree of vertex i; the neighbours of i are stored in the array `e` from the position $\mathtt{v}[i]$ to $\mathtt{v}[i+\mathtt{d}[i]-1]$. Therefore each entry k of `e` represents an arc (j,k) of G, for some j. We write $i_{\mathtt{e}}(j,k)$ to denote the index of `e` which corresponds to the edge (j,k). The weight of (j,k) appears in $\mathtt{w}[i_{\mathtt{e}}(j,k)]$. Empty entries may appear in `d`,`e` and `w` for the user's convenience.

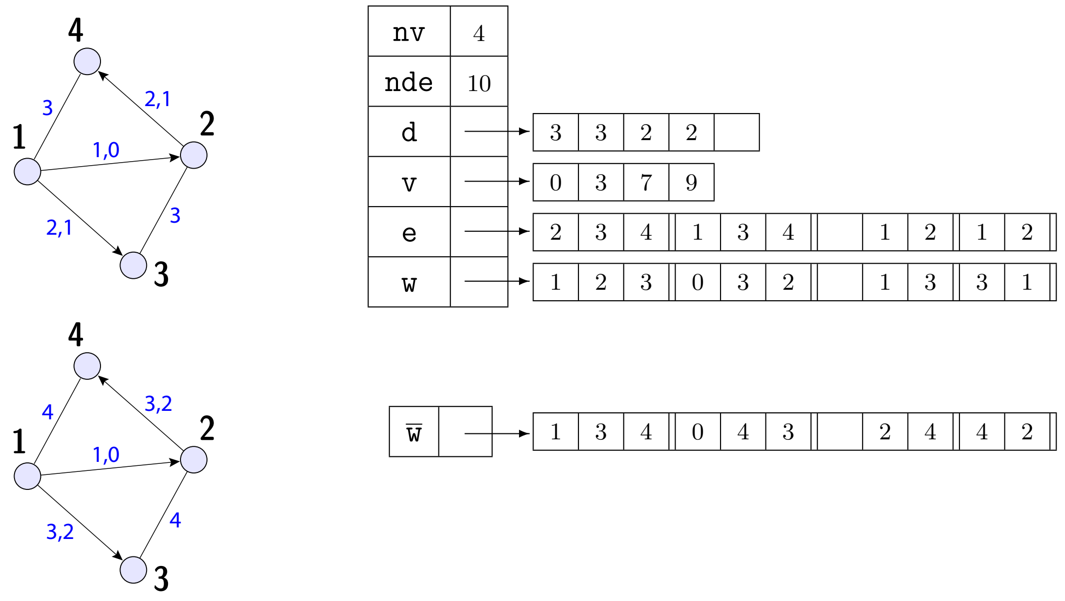

4
3
2,1
1
1,0
2
2,1
3
3
nv 4
nde 10
d 3 3 2 2
v 0 3 7 9
e 2 3 4 | 1 3 4 | | 1 2 | 1 2
w 1 2 3 | 0 3 2 | | 1 3 | 3 1
4
4
3,2
1
1,0
2
3,2
4
3
w̄ 1 3 4 | 0 4 3 | | 2 4 | 4 2

Figure 7 Data structures for weighted digraphs and assignment of internal weights.

We remark that: (i) neighbours of vertices of G are stored in `e` in *increasing order*; (ii) we can *virtually remove* the first neighbour of a vertex i by adding 1 to the value of $\mathtt{v}[i]$ and subtracting 1 to $\mathtt{d}[i]$. In order not to destroy the graph while removing edges, a copy of `d` and `v` will be made at the beginning of the computation. This costs $O(n)$ time and space.

Assume now that we are traversing the edges of the graph G in lexicographic order. By (i) and (ii) above, for each visited edge (u, v) we can easily find the index in $\mathtt{e}$ of its opposite edge (v, u): it will be the first neighbour of v, provided that we have virtually removed all the opposites of the edges visited so far.

Algorithm 4 describes the procedure for assigning internal weights to a graph $G = (V, E, w)$. The traversal of edges of G in lexicographic order is executed by the two nested **for** loops starting at lines 3-4. For each arc (u, v), we use the array A to store two triples $(w(u, v), w(v, u), i_{\mathtt{e}}(u, v))$ and $(w(v, u), w(u, v), i_{\mathtt{e}}(v, u))$. The resulting array is sorted according to the lexicographic order of the first two components of triples. This can be done in $O(m)$ steps via bucket sorting under the assumption that the values original weights are bound by m. Finally, the array of internal weights is build: for each triple (w_1, w_2, i) in A, we assign to $\overline{\mathtt{w}}[i]$ the appropriate value, in the obvious way.

Algorithm 4 Assignment of internal weights.

```
Data: A graph $G(V, E, w)$ and an empty array $A$ (of length $|E|$) of triples of integers.
Result: The array $A$ with all the triples $(w(u,v), w(v,u), i_{\mathtt{e}}(u,v)), \forall (u,v) \in E$, and the
        array $\overline{\mathtt{w}}$ of internal weights of $G$.
1  Make a copy dc of d and a copy vc of v;
2  $\ell = 1$;
3  for $i = 1$ to $n$ do
4      for $k = 1$ to dc[$i$] do
5          $k_1$ = vc[$i$] + $k$;      let $j$ be the $k$-th neighbour of $i$; then $k_1$ is the index of $(i, j)$ in e
6          $k_2$ = vc[e[$k_1$]];                                      and $k_2$ is the index of $(j, i)$ in e
7          $A[\ell] \leftarrow$ (w[$k_1$], w[$k_2$], $k_1$); $\ell = \ell + 1$;          store the triple associated to $(i, j)$
8          $A[\ell] \leftarrow$ (w[$k_2$], w[$k_1$], $k_2$); $\ell = \ell + 1$;          store the triple associated to $(j, i)$
9          vc[$k_2$] = vc[$k_2$] + 1; dc[$k_2$] = dc[$k_2$] − 1;                    virtually remove $(j, i)$
       end
   end
10 Sort $A$ according to the first and then to the second element of triples;
   /* assignment of internal weights                                                         */
11 $newweight = 0$;
12 Let $(a_1, b_1, i_1)$ be the triple in $A[1]$;
13 $\overline{\mathtt{w}}[i_1] = newweight$;
14 for $j = 2$ to $|E|$ do
15     Let $(a_j, b_j, i_j)$ be the triple in $A[j]$;
16     if $(a_j, b_j) \neq (a_{j-1}, b_{j-1})$ then $newweight = newweight + 1$;
17     $\overline{\mathtt{w}}[i_j] = newweight$;
   end
18 Free dc and vc;
```