A parametric filtering algorithm for the graph isomorphism problem

Sébastien Sorlin, Christine Solnon

A parametric filtering algorithm for the graph isomorphism problem

Sébastien Sorlin, Christine Solnon
LIRIS, CNRS UMR 5205, bât. Nautibus, University of Lyon I
43 Bd du 11 novembre, 69622 Villeurbanne cedex, France
`{sebastien.sorlin,christine.solnon}@liris.cnrs.fr`

February 15, 2008

Abstract

We introduce a new filtering algorithm, called IDL(d)-filtering, for a global constraint dedicated to the graph isomorphism problem —the goal of which is to decide if two given graphs have an identical structure. The basic idea of IDL(d)-filtering is to label every vertex with respect to its relationships with other vertices around it in the graph, and to use these labels to filter domains by removing values that have different labels. IDL(d)-filtering is parameterized by a positive integer value d which gives a limit on the distance between a vertex to be labelled and the set of vertices considered to build its label. We experimentally compare different instantiations of IDL(d)-filtering with state-of-the-art dedicated algorithms and show that IDL(d)-filtering is more efficient on regular sparse graphs and competitive on other kinds of graphs.

1 Introduction

Graphs are widely used in real-life applications to model structured objects, *e.g.*, molecules, images, or networks. In many of these applications, one has to compare graphs to decide if their structures are identical. This problem is known as the Graph Isomorphism Problem (GIP). This problem can also be used to detect symmetries into constraint satisfaction problems [Pug05, ZDD06].

More formally, a *graph* is defined by a couple (V, E) such that V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a bijective function $f : V \rightarrow V'$ such that for every pair of vertices $(u, v) \in V \times V$, we have $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. We shall say that f is an *isomorphism function.* The GIP consists in deciding if two given graphs are isomorphic.

There exist many dedicated algorithms for solving GIPs such as, *e.g.*, [Ull76, McK81, CFSV04, DLSM04]. These algorithms are often very efficient (even though their worst case time complexities are exponential). However, such dedicated algorithms cannot be used to solve more specific problems, such as isomorphism problems with additional constraints, or larger problems that include graph isomorphism subproblems.

An attractive alternative to these dedicated algorithms is to use Constraint Programming (CP), which provides a generic framework for solving any kind of Constraint Satisfaction Problems (CSPs). Indeed, GIPs can be transformed into CSPs in a very straightforward way [McG79], so that one can use generic constraint solvers to solve them. However, when transforming a GIP into a CSP, the global semantic of the problem is lost and decomposed into a set of binary constraints. As a consequence, using CP to solve GIPs may be less efficient than using dedicated algorithms which have a global view of the problem.

Motivations. In order to allow constraint solvers to solve GIPs more efficiently without loosing CP's flexibility, we have introduced in [SS04] a global constraint dedicated to GIPs and a first filtering algorithm, called Label-filtering. In [SS07], we have proposed another filtering algorithm, called ILL-filtering. Both ILL-filtering and Label-filtering are based on the computation of a so called "isomorphic-consistent" labelling, *i.e.*, a labelling of the graph vertices such that two vertices which have different labels cannot be matched by an isomorphism function. However, to compute the label of a vertex u, ILL-filtering only considers the direct neighborhood of u whereas Label-filtering considers all vertices of the graph. Also, ILL-filtering iteratively strengthens labels until a fix-point is reached whereas Label-filtering only iterates once the strengthening procedure.

We introduce in this article a new parametric filtering algorithm, called IDL(d)-filtering, which is also based on the computation of an isomorphic-consistent labelling. The label of a vertex u is computed by considering all vertices that are at most at distance d from u, where d is a parameter. IDL(d)-filtering is a generalization of ILL-filtering —which corresponds to IDL(1)-filtering— and Label-filtering —which corresponds to the first two iterations of IDL(∞)-filtering.

Outline of the paper. Section 2 gives an overview of existing approaches for solving the graph isomorphism problem, including CP approaches. Section 3 introduces a labelling procedure based on an invariant distance property. Section 4 shows how to use this distance-based labelling to define IDL(d)-filtering. Section 5 illustrates IDL(d)-filtering on a graph isomorphism problem instance for $d = 1$ and $d = \infty$. Section 6 experimentally compares different instantiations of IDL(d)-filtering with state-of-the-art approaches.

2 Existing approaches for solving graph isomorphism problems

2.1 Complexity of the graph isomorphism problem

The theoretical complexity of the GIP is not exactly stated: the problem is in NP but it has not been shown to be in P nor to be NP-complete [For96] and its own complexity class, *isomorphism-complete*, has been defined. However, when adding some topological restrictions on graphs (e.g., planar graphs [HW74], trees [AHU74] or bounded valence graphs [Luk82]) this problem becomes solvable in polynomial time.

2.2 Dedicated algorithms

To solve a GIP, one has to find a one to one mapping between the vertices of the two graphs. The search space composed of all possible mappings may be explored in a "Branch and Cut" way: at each node of the search tree, some graph properties (such as edge distribution or vertex neighborhood) can be used to prune the search space [CFSV04, Ull76]. This kind of approach is rather efficient and can be used to solve GIPs up to a thousand or so vertices very quickly (in less than one second). In [SD76], Schmidt *et al.* propose such an algorithm that prunes the search tree by using a distance matrix.

McKay [McK81] proposes another approach, which has been originally designed to detect graph automorphisms, *i.e.*, non trivial isomorphisms between a graph and itself. The main idea is to compute a canonical representation of a graph such that two graphs have the same representation if and only if they are isomorphic. This canonical representation is an ordered partition of the vertices such that all vertices within a same part are equivalent (with respect to an isomorphism function). This partition is computed, starting from an initial partition that groups all vertices into a same part, by iteratively applying an ordered set of vertex invariants to split parts. This approach is implemented in *Nauty* which is, to our knowledge, the most efficient solver for the graph isomorphism problem in the general case: *Nauty* is comparable to "Branch and Cut" methods but *Nauty* is often the quickest for large graphs [FSV01]. In [DLSM04], Darga *et al.* propose a similar algorithm called *Saucy* which is specialized for sparse graphs (with very low edge densities) and which is faster than *Nauty* on this kind of graphs. Puget [Pug05] proposes another algorithm for the graph automorphism problem which is even faster on sparse graphs. Finally, McKay has recently proposed an adaptation of *Nauty* called *Sparetest* dedicated to sparse graphs (the code has been sent to us in a personal communication).

All these dedicated algorithms can efficiently solve GIPs in practice, even though their worst case complexities are exponential. However, they are not suited for solving more specific problems, such as GIPs with additional constraints. In particular, vertices and edges may be associated with labels that characterize them, and one may be interested in looking for isomorphism functions that satisfy additional constraints on these labels. This is the case, *e.g.*, in [Rég95] where graphs are used to represent molecules, or in computer aided design (CAD) applications where graphs are used to represent design objects [CS03].

2.3 Constraint Programming

Constraint Programming (CP) is an attractive alternative to dedicated approaches: it provides high level languages to declaratively model Constraint Satisfaction Problems (CSPs); these CSPs are solved in a generic way by embedded constraint solvers [Tsa93, LO00, ILO00, HSD92]. A *CSP* is defined by a triple (X, D, C) such that

- X is a finite set of variables,
- D is a function which maps every variable $x_i \in X$ to its domain $D(x_i)$, *i.e.*, the finite set of values that may be assigned to x_i,
- C is a set of constraints, *i.e.*, relations between some variables which restrict the set of values that can be assigned simultaneously to these variables. Constraints involving two variables are called binary constraints; we shall denote $C(x_i, x_j)$ the binary constraint holding between the two variables x_i and x_j, and we shall define this constraint by the set of couples $(v_i, v_j) \in D(x_i) \times D(x_j)$ that satisfy the constraint.

Solving a CSP (X, D, C) involves finding a complete assignment, which assigns a value $v_i \in D(x_i)$ to every variable $x_i \in X$, such that all constraints in C are satisfied.

Graph isomorphism problems may be formulated as CSPs in a very straightforward way [GJ79, Rég95]. Given two graphs $G = (V, E)$ and $G' = (V', E')$, one may define the CSP (X, D, C) such that

- a variable x_u is associated with each vertex $u \in V$, *i.e.*, $X = \{x_u \mid u \in V\}$,
- the domain of each variable x_u is the set of vertices of G' that have the same number of adjacent vertices as u, *i.e.*,
$$D(x_u) = \{u' \in V' \mid \quad \#\{(u,v) \in E\} = \#\{(u',v') \in E'\}\ \}$$
- there is a binary constraint between every pair of different variables $(x_u, x_v) \in X \times X$, denoted by $C_{edge}(x_u, x_v)$. This constraint expresses the fact that the two vertices of G' that are assigned to x_u and x_v must be connected by an edge in G' if and only if u and v are connected by an edge in G, *i.e.*,
$$\begin{array}{ll} \text{if } (u,v) \in E, & C_{edge}(x_u, x_v) = E' \\ \text{otherwise} & C_{edge}(x_u, x_v) = \{(u',v') \in V' \times V' \mid u' \neq v' \text{ and } (u',v') \notin E'\} \end{array}$$

Once a GIP has been formulated as a CSP, one can use CP to solve it in a generic way. Within this framework, additional constraints, such as constraints on vertex and edge labels, may be expressed very easily.

2.4 Global constraint for the graph isomorphism problem

The CSP formulation described in 2.3 decomposes the global semantic of the GIP into a set of binary edge constraints. Each of these edge constraints expresses the necessity either to preserve or to forbid an edge in a local way. As a consequence, using CP to solve GIPs is often less efficient than using a dedicated algorithm.

To improve the solution process of CSPs associated with GIPs, one may add an *allDiff* global constraint, in order to constrain all variables to be assigned to different vertices [Rég95]. This constraint is redundant as each binary edge constraint only contains couples of different vertices, so it is not possible to assign the same value to two different variables. This global constraint allows a constraint solver to prune the search space more efficiently, and therefore to solve GIPs quicker.

However, even with an *allDiff* global constraint, CP is still not competitive with dedicated algorithms because most of the global semantic of the problem is still lost. Hence, we have introduced in [SS04] a global constraint for the graph isomorphism problem.

Syntactically, this constraint is defined by the relation $gip(V, E, V', E', L)$ where

- V and V' are two sets of values such that $\#V = \#V'$,
- $E \subseteq V \times V$ is a set of pairs of values from V,
- $E' \subseteq V' \times V'$ is a set of pairs of values from V',
- L is a set of couples which associates a different variable of the CSP to each different value of V, *i.e.*, L is a set of $\#V$ couples of the form (x_u, u) where x_u is a variable of the CSP and u is a value of V, and such that for any pair of different couples (x_u, u) and (x_v, v) of L, $x_u \neq x_v$ and $u \neq v$.

Semantically, the global constraint $gip(V, E, V', E', L)$ is consistent if and only if there exists an isomorphism function $f : V \to V'$ such that for each couple $(x_u, u) \in L$ there exists a value $u' \in D(x_u)$ so that $u' = f(u)$.

This global constraint is not semantically global [BH03] as it can be represented by a semantically equivalent set of binary constraints as described previously. However, the *gip* constraint allows us to exploit the global semantic of GIPs to solve them more efficiently.

3 Theoretical framework

We show in this section how to build a distance-based labelling which will be used in the next section to define a filtering algorithm for the *gip* global constraint. The main idea is to label every vertex with respect to distance relationships with other vertices of the graph. This labelling is *isomorphic-consistent*, *i.e.*, two vertices that have different labels cannot be matched by an isomorphism function. Hence, this labelling can be used to filter domains by removing vertices which have different labels. Labels are built iteratively: starting from an empty label, each label is extended by considering labels of vertices within a given distance d. This labelling extension, called relabelling, is iterated until a fix-point is reached.

The distance d is a parameter of the relabelling procedure. When it is set to 1, labels are iteratively extended by considering labels of neighbors in a very similar way to the partition refinement procedure of *Nauty*. When the distance d is set to a value larger than 1, one obtains a stronger labelling than the partition refinement of Nauty.

In this section, we first show that distances are preserved by isomorphism functions. Then, we introduce labellings and relabellings. Finally, we define a distance-based relabelling. We assume $gip(V, E, V', E', L)$ to be the underlying graph isomorphism constraint to propagate, and we define $Vertices = V \cup V'$ and

$Edges = E \cup E'$. We assume without loss of generality that $V \cap V' = \emptyset$ and that each graph is connected. We restrict our attention to non directed graphs. The extension of our work to directed graphs is discussed in 7.

3.1 Distance-based invariant property

Definition. A *path* between two vertices u and v is a sequence $<v_0, v_1, v_2, ..., v_k>$ of vertices such that $v_0 = u$, $v_k = v$ and for all $i \in [1, k]$, $(v_{i-1}, v_i) \in Edges$. The *length* of a path is the number of its edges.

Definition. The *distance* between two vertices u and v, denoted by $\delta(u, v)$, is the length of the shortest path between u and v.

Definition. The *diameter of a graph* G is the largest distance between two vertices of G.

Our filtering procedure for the graph isomorphism problem is based on the following theorem which shows that distances are preserved by isomorphism functions.

Theorem 1. Given a bijective function $f : V \to V'$, the two following properties are equivalent:

1. f is an isomorphism function, *i.e.*, f is such that $\forall(u, v) \in V \times V, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$;
2. $\forall(u, v) \in V \times V, \delta(u, v) = \delta(f(u), f(v))$.

Proof. (1) $\Rightarrow$ (2): if f is an isomorphism function, then (u, v) is an edge of G iff $(f(u), f(v))$ is an edge of G' so that $< v_1, v_2, ..., v_n >$ is a path in G iff $< f(v_1), f(v_2), ..., f(v_n) >$ is a path in G', and therefore $< v_1, v_2, ..., v_n >$ is a shortest path in G iff $< f(v_1), f(v_2), ..., f(v_n) >$ is a shortest path in G', and property (2) holds.
(2) $\Rightarrow$ (1): For any pair of vertices $(u, v) \in V \times V$, if (u, v) is an edge of G, then $< u, v >$ is the shortest path between u and v so that $\delta(u, v) = 1$, and therefore $\delta(f(u), f(v)) = 1$, so that $(f(u), f(v))$ is an edge of G' (and vice versa).

3.2 Isomorphic-consistent labelling and relabelling

Before defining a labelling procedure based on Theorem 1, we introduce in this section some definitions about labellings and relabellings.

Definition. A *labelling* is a function denoted by α that associates a label $\alpha(v)$ to every vertex $v \in Vertices$. This label does not depend on vertex names but only on relations defined by edges between v and other vertices. We note $image(\alpha)$ the set of labels returned by α, *i.e.*, $image(\alpha) = \{\alpha(v) \mid v \in Vertices\}$.

Definition. A labelling α is *isomorphic-consistent* if for every isomorphism function f between (V, E) and (V', E'), vertices matched by f have identical labels, *i.e.*, $\forall v \in V, \alpha(v) = \alpha(f(v))$.

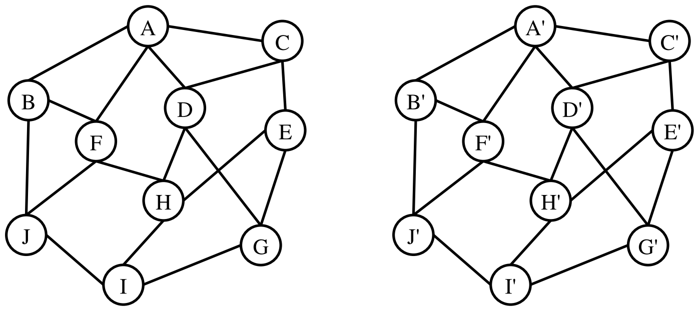

Figure 1: Definition of a $gip(V, E, V', E', L)$ constraint instance such that $V = \{A, B, C, D, E, F, G, H, I, J\}$ and $V' = \{A', B', C', D', E', F', G', H', I', J'\}$, $L = \{(x_A, A), (x_B, B), (x_C, C), (x_D, D), (x_E, E), (x_F, F), (x_G, G), (x_H, H), (x_I, I), (x_J, J)\}$, E and E' are defined as graphically displayed above.

Example 1. Let us consider the *gip* constraint instance of Figure 1 and let us define the labelling α_{deg} which labels each vertex by its degree, *i.e.*,

$$\forall v \in \mathit{Vertices}, \alpha_{deg}(v) = \#\{u \in V \mid (u, v) \in \mathit{Edges}\}$$

We have $\alpha_{deg}(A) = \alpha_{deg}(D) = \alpha_{deg}(F) = \alpha_{deg}(H) = 4$ and $\alpha_{deg}(B) = \alpha_{deg}(C) = \alpha_{deg}(E) = \alpha_{deg}(G) = \alpha_{deg}(I) = \alpha_{deg}(J) = 3$. This labelling is isomorphic-consistent as isomorphism functions only match vertices that have a same number of adjacent vertices.

An isomorphic-consistent labelling may be used to filter domains: the domain of every variable x_u associated with a vertex u may be reduced to the set of vertices that have the same label as u. Our goal is to build an isomorphic-consistent labelling that filters domains as much as possible, *i.e.*, that associates as much as possible different labels to vertices that cannot be matched.

Definition. A labelling α_1 *is at least as strong as* a labelling α_2 if

$$\forall (u, v) \in V \times V', \alpha_2(u) \neq \alpha_2(v) \Rightarrow \alpha_1(u) \neq \alpha_1(v)$$

To strengthen a labelling, we propose to iteratively apply a relabelling function.

Definition. A *relabelling* is a function denoted by β that, given a labelling α, returns a new labelling noted $\beta[\alpha]$.

Definition. A relabelling β is *isomorphic-consistent* if for any isomorphic-consistent labelling α, $\beta[\alpha]$ is also an isomorphic-consistent labelling.

Relabellings may be defined with respect to labels of adjacent vertices; as several vertices may have the same label, we introduce the following notation for multisets.

Definition. A *multiset* is a bag which may contain several occurrences of a same value. Given an underlying set S, we note a^k the fact that a value $a \in S$ occurs k times in a multiset m.

Example 2. Given the set $S = \{a, b, c, d\}$, $m = \{\{a, a, b, d, d, d\}\} = \{\{a^2, b^1, d^3\}\}$ is the multiset that contains two occurrences of a, one occurrence of b and three occurrences of d.

Example 3. Let us define the relabelling β_{adj} that relabels every vertex by the multiset composed of the labels of its neighbors, *i.e.*,

$$\forall v \in \mathit{Vertices}, \beta_{adj}[\alpha](v) = \{\{l^k \mid l \in \mathit{image}(\alpha), k = \#\{u \in \mathit{Vertices}, (v,u) \in \mathit{Edges}, l = \alpha(u)\}, k > 0\}\}$$

This relabelling β_{adj} is isomorphic-consistent because two vertices can be associated by an isomorphism function only if their neighbors can. If we consider the labelling α_{deg} of Example 1 and the *gip* constraint instance of Figure 1, we have

$$\begin{aligned}
\beta_{adj}[\alpha_{deg}](A) = \beta_{adj}[\alpha_{deg}](D) = \beta_{adj}[\alpha_{deg}](F) = \beta_H[\alpha_{deg}] &= \{\{3^2, 4^2\}\} \\
\beta_{adj}[\alpha_{deg}](B) = \beta_{adj}[\alpha_{deg}](C) &= \{\{3^1, 4^2\}\} \\
\beta_{adj}[\alpha_{deg}](E) = \beta_{adj}[\alpha_{deg}](G) = \beta_{adj}[\alpha_{deg}](I) = \beta_J[\alpha_{deg}] &= \{\{3^2, 4^1\}\}
\end{aligned}$$

Definition. A relabelling β is *strengthening* if for any labelling α, $\beta[\alpha]$ is at least as strong as α.

A very simple way to ensure that a relabelling β is strengthening is to define β in such a way that, for each vertex $v \in \mathit{Vertices}$, $\beta[\alpha](v)$ is prefixed by $\alpha(v)$.

An isomorphic-consistent relabelling function β can be used to iteratively define new isomorphic-consistent labelling functions: starting from an elementary isomorphic-consistent labelling function α, β can be iteratively applied, thus defining a sequence of labellings. We note $\beta^i[\alpha]$ the labelling obtained by iterating the relabelling β i times, starting from α. More precisely, we define:

$$\begin{aligned}
\beta^0[\alpha] &= \alpha \\
\beta^i[\alpha] &= \beta[\beta^{i-1}[\alpha]], \forall i \geq 1
\end{aligned}$$

3.3 Relabelling function based on distances

Theorem 1 shows that graph isomorphism functions preserve distances. This property can be used to define a relabelling function β_d. Basically, the idea is to extend the label of a vertex u by the labels of other vertices within a distance d from u. As several vertices may have a same label, this extension is a multiset.

Definition. Given a vertex v, a distance $d \geq 0$ and a labelling function α, we note $\Delta(v, d, \alpha)$ the multiset composed of labels of vertices at distance d from v. More formally,

$$\Delta(v,d,\alpha) \quad = \quad \{\{l^k \mid l \in \mathit{image}(\alpha), k = \#\{u \in \mathit{Vertices} \mid \delta(u,v) = d, \alpha(u) = l\}, k \geq 1\}\}$$

Example 4. Let us consider the *gip* global constraint instance of Figure 1 and the labelling function α_{deg} of Example 1. We have

$\Delta(A,0,\alpha_{deg})$	$=$	$\{\{4^1\}\}$	*(at distance 0 from A:*	*there is 1 vertex (A) labelled by* 4*)*
$\Delta(A,1,\alpha_{deg})$	$=$	$\{\{3^2, 4^2\}\}$	*(at distance 1 from A:*	*there are 2 vertices (B and C) labelled by* 3 *and 2 vertices (D and F) labelled by* 4*)*
$\Delta(A,2,\alpha_{deg})$	$=$	$\{\{3^3, 4^1\}\}$	*(at distance 2 from A:*	*there are 3 vertices (E, G and J) labelled by* 3 *and 1 vertex (H) labelled by* 4*)*
$\Delta(A,3,\alpha_{deg})$	$=$	$\{\{3^1\}\}$	*(at distance 3 from A:*	*there is 1 vertex (I) labelled by* 3*)*
$\Delta(A,4,\alpha_{deg})$	$=$	$\emptyset$	*(at distance 4 from A:*	*there is no vertex)*

Definition. Given a labelling function α and a positive integer d, the relabelling function β_d returns a new labelling function $\beta_d[\alpha]$ which labels each vertex v by a set of $d+1$ multisets, such that each multiset contains the labels of vertices at distance k from v (with $k \in [0,d]$), *i.e.*,

$$\forall v \in V, \beta_d[\alpha](v) \quad = \quad \{k : \Delta(v,k,\alpha) \mid k \in [0,d]\}$$

Example 5. Let us consider the *gip* global constraint instance of Figure 1 and the labelling function α_{deg} of Example 1. We have

$$\begin{array}{ll}\beta_1[\alpha_{deg}](A) & = \{0 : \{\{4^1\}\}, 1 : \{\{3^2, 4^2\}\}\ \} \\ \beta_2[\alpha_{deg}](A) & = \{0 : \{\{4^1\}\}, 1 : \{\{3^2, 4^2\}\}, 2 : \{\{3^3, 4^1\}\}\ \} \\ \beta_3[\alpha_{deg}](A) & = \{0 : \{\{4^1\}\}, 1 : \{\{3^2, 4^2\}\}, 2 : \{\{3^3, 4^1\}\}, 3 : \{\{3^1\}\}\ \} \\ \beta_k[\alpha_{deg}](A) & = \beta_3[\alpha_{deg}](A), \forall k > 3\end{array}$$

In other words, $\beta_1[\alpha_{deg}]$ extends α_{deg} by adding labels of neighbors; $\beta_2[\alpha_{deg}]$ extends $\beta_1[\alpha_{deg}]$ by adding labels of vertices at distance 2; and $\beta_3[\alpha_{deg}]$ extends $\beta_2[\alpha_{deg}]$ by adding labels of vertices at distance 3. As the diameter of the graph is 3, $\beta_k[\alpha_{deg}] = \beta_3[\alpha_{deg}]$ for every distance k greater than 3.

The next two theorems show that β_d is both strengthening and isomorphic-consistent, and that the larger d, the stronger β_d

Theorem 2. For every distance $d \in \mathcal{N}$, the function β_d is an isomorphic-consistent relabelling.

Proof. If α is an isomorphic-consistent labelling, then, for each isomorphism function f between G and G', $\forall u \in V, \alpha(u) = \alpha(f(u))$. Furthermore, as f is an isomorphism function and given theorem 1, $\forall(u,v) \in \mathit{Vertices}^2, \delta(u,v) = \delta(f(u), f(v))$. As a consequence, $\forall u \in V, \forall l \in \mathit{image}(\alpha), \forall j \in [0, \#V], \#\{v | v \in V \wedge \delta(u,v) = j \wedge \alpha(v) = l\} = \#\{v' | v' \in V' \wedge \delta(f(u), v') = j \wedge \alpha(v') = l\}$ (because f is a bijective application). As a consequence, $\forall u \in V, \beta_d[\alpha](u) = \beta_d[\alpha](f(u))$ and $\beta_d[\alpha]$ is an isomorphic-consistent labelling function.

Theorem 3. Given a labelling α and two integers k and l such that $1 \leq k < l$, $\beta_l[\alpha]$ is at least as strong as $\beta_k[\alpha]$ which is at least as strong as α.

Proof. $\beta_l[\alpha]$ is at least as strong as $\beta_k[\alpha]$ because, by definition, for every vertex $v \in \mathit{Vertices}$, $\beta_l[\alpha](v) = \beta_k[\alpha](v) \cup \{i : \Delta(v,i,\alpha) \mid i \in [0,d]\}$. Therefore, $\beta_k[\alpha](u) \neq \beta_k[\alpha](v) \Rightarrow \beta_l[\alpha](u) \neq \beta_l[\alpha](v)$.
$\beta_k[\alpha]$ is at least as strong as α because for every vertex $v \in \mathit{Vertices}$, $0 : \{\alpha(v)^1\}$ belongs to $\beta_k[\alpha](v)$. Therefore, $\alpha(u) \neq \alpha(v) \Rightarrow \beta_k[\alpha](u) \neq \beta_k[\alpha](v)$.

Finally, the relabelling β_d can be iteratively applied, starting from an initial labelling α, thus defining a sequence of labellings $\beta_d^0[\alpha]$, $\beta_d^1[\alpha]$, $\beta_d^2[\alpha]$, ... As each labelling $\beta_d^i[\alpha]$ is at least as strong as $\beta_d^{i-1}[\alpha]$, this sequence necessarily reaches a fix-point at some step k such that every labelling $\beta_d^{k+i}[\alpha]$ is equivalent to $\beta_d^k[\alpha]$. The next theorem shows that this fix-point is reached when at some step k the number of different labels is not increased.

Theorem 4. Given a distance d, an initial labelling α and a positive integer k, if

$$\forall(u,v) \in \mathit{Vertices} \times \mathit{Vertices}, \beta_d^k[\alpha](u) = \beta_d^k[\alpha](v) \Rightarrow \beta_d^{k+1}[\alpha](u) = \beta_d^{k+1}[\alpha](v)$$

then

$$\forall j \geq k, \forall(u,v) \in \mathit{Vertices} \times \mathit{Vertices}, \beta_d^k[\alpha] = \beta_d^k[\alpha] \Rightarrow \beta_d^j[\alpha](u) = \beta_d^j[\alpha](v)$$

Proof. Given its definition, we can see that β_d does not use the labels given by α themselves but only an equivalence relation between these labels. As a consequence, when a relabelling of the vertices does not change the equivalence relation between the vertex labels, any further relabelling cannot change this equivalence relation any more.

Roughly speaking, theorem 4 shows that, when a step of the sequence β_d^k does not increase the number of different vertex labels, a fix-point is reached and the relabelling process can be stopped. Finally, as the number of different labels is bounded by $\#V$, this fix-point is reached in at most $\#V$ steps.

4 Practical framework

4.1 IDL(d)-consistency and IDL(d)-filtering

We now propose to use the distance-based relabelling β_d to define a new partial consistency —called Iterative Distance Label (IDL) consistency— and an associated filtering algorithm for the *gip* constraint.

The relabelling β_d is iterated starting from an initial labelling α. We first define this initial labelling to be the labelling $\alpha_\emptyset$ which associates the same label $\emptyset$ to every vertex, *i.e.*, for every vertex $v \in \mathit{Vertices}, \alpha_\emptyset(v) = \emptyset$. We shall introduce other initial labellings in section 4.2.

Definition. Given a distance $d \geq 1$, a $gip(V, E, V', E', L)$ global constraint is IDL(d)-consistent if for every value v in the domain of a variable associated by L to a vertex u, the vertices u and v are associated with a same label by any labelling $\beta_d^k[\alpha_\emptyset]$, *i.e.*,

$$\forall (x_u, u) \in L,\ \forall v \in D(x_u),\ \forall k \geq 0, \beta_d^k[\alpha_\emptyset](u) = \beta_d^k[\alpha_\emptyset](v)$$

Algorithm 1 describes a filtering procedure that ensures IDL(d)-consistency. Starting from an initial labelling β^0 that associates the same label $\emptyset$ to every vertex (lines 1–2), this procedure iteratively computes β^i from β^{i-1} (lines 5–12), renames the labels of β^i (line 13) and filters domains with respect to β^i (lines 14–15) until either a domain becomes empty —thus proving inconsistency— or the number of labels has not increased —thus reaching a fix-point.

The time and space complexities of relabelling, renaming and filtering are studied below. We define $n = \#\mathit{Vertices}$ and $p = \#\mathit{Edges}$.

Computation of β^i from β^{i-1} (lines 5–12). This step basically implies n breadth first searches bounded by d: starting from every vertex v, we iteratively compute the sets δ_k of vertices at distance k from v, for each distance $k \in [1, d]$.

The time complexity of this step depends on the d parameter.

- In the worst case, *i.e.*, if d is greater than or equal to the diameter of the graph, it corresponds to n full breadth first searches so that it is in $\mathcal{O}(np)$.

  This complexity could be reduced to $\mathcal{O}(n^2)$ by memorizing, for every vertex $v \in \mathit{Vertices}$ and every distance $k \in [1, d]$ a list $\delta_k(v)$ of vertices at distance k from v. However, experiments have shown us that this implementation actually spends more CPU time. The reason is that adjacency lists are often already stored in the CPU cache memory, so that accessing to the neighbors of a vertex is often very quickly done, whereas $\delta_k(v)$ lists are too big to stay in the cache memory, so that the processor often has to restore these lists from the RAM to its cache, which is more time consuming.

Algorithm 1: IDL(d) Filtering procedure

```
Input: a constraint $gip(V, E, V', E', L)$,
       the domain $D(x_u)$ of every variable $x_u$ occurring in $L$,
       a distance $d \geq 1$
Output: filtered domains $D$ such that $gip(V, E, V', E', L)$ is IDL($d$)-consistent
1  foreach $v \in$ Vertices do
2      $\beta^0(v) \leftarrow \emptyset$
3  $i \leftarrow 1$
4  repeat
       /* Computation of labelling $\beta^i$ from labelling $\beta^{i-1}$ */
5      foreach $v \in$ Vertices do
6          $\delta_0 \leftarrow \{v\}$
7          $marked \leftarrow \{v\}$
8          for $k$ in $1..d$ and while $\delta_{k-1} \neq \emptyset$ do
               /* Invariant: $\delta_{k-1}$ = set of vertices at distance $k-1$ from $v$ */
               /* and marked = set of vertices at distance $j \leq k-1$ from $v$ */
9              $\delta_k \leftarrow \{u \mid \exists u' \in \delta_{k-1}, (u', u) \in Edges, u \notin marked\}$
10             compute the multiset $m_k$ which contains an occurrence of $\beta^{i-1}(u)$ for each vertex $u \in \delta_k$
11             $marked \leftarrow marked \cup \delta_k$
12         $\beta^i(v) \leftarrow \{0 : \{\beta^{i-1}(v)\} \cup \{k : m_k \mid k \in 1..d\}$
13     rename labels of $\beta^i$
       /* Filtering with respect to the new labelling $\beta^i$ */
14     foreach $(x_u, u) \in L$ do
15         $D(x_u) \leftarrow D(x_u) \cap \{v \in Vertices, \beta^i(u) = \beta^i(v)\}$
16     $i \leftarrow i + 1$
17 until $\exists(x_u, u) \in L, D(x_u) = \emptyset$ or $\#\{\beta^{i-1}(u), u \in \text{Vertices}\} = \#\{\beta^i(u), u \in \text{Vertices}\}$ ;
```

- In the best case, *i.e.*, if $d = 1$, one only has to compute for each vertex v the multiset m_1 containing the labels of the direct neighbors of v. In this case, the time complexity for computing β^i from β^{i-1} is $\mathcal{O}(p)$.

The space complexity of this step does not depend on d, as we do not memorize lists of vertices at a given distance. It is in $\mathcal{O}(n^2)$ as there are n labels, and the size of each label is bounded by n (remember that labels are renamed at each iteration).

Renaming step (line 13). This step is introduced in order to allow us to manage vertex labels in constant time and memory. Indeed, at each relabelling iteration, labels become larger. However, one can easily show that these labels can be renamed after each relabelling step, provided that the equivalence relation defined by labels is preserved by the renaming.

To rename labels, they must be sorted. The size of each label is bounded by n, and there are n labels. Therefore, sorting all labels is done in $\mathcal{O}(n^2 \cdot \log n)$. We use a Hash table to compare and rename vertex labels. With such a table, the time complexity for renaming a sorted label is linear with respect to the size of the label, *i.e.*, in $\mathcal{O}(n)$, provided that the table is large enough to limit the number of collisions.

The space complexity of this step is in $\mathcal{O}(n^2)$ as the Hash table has $\mathcal{O}(n)$ entries and the size of each entry is in $\mathcal{O}(n)$.

Filtering step (lines 12–13). This step is done in $\mathcal{O}(n^2)$.

Complexity of IDL(d)-filtering. In the worst case, the fix-point is reached after $\mathcal{O}(n)$ iterations of the repeat until loop. Therefore, the time complexity of IDL(d)-filtering is $\mathcal{O}(n^2(p + n \log n))$ if d is greater or equal to the diameter of the graph; it is $\mathcal{O}(n^3 \log n)$ if $d = 1$. The space complexity is $\mathcal{O}(n^2)$.

4.2 Initial labelling

Algorithm 1 starts the relabelling process from an initial labelling $\alpha_\emptyset$ which labels all vertices with $\emptyset$ so that every vertex of V may be matched with every vertex of V'. However, it may happen that the domain of some variables associated with vertices of V have been reduced, either by the propagation of other constraints of the CSP, or when by a domain splitting during a branch and propagate search.

In this case, the relabelling process may be started from an initial labelling which integrates as much as possible these domain reductions, *i.e.*, such that if a vertex u does not belong to the domain of a variable x_v, then u and v are associated with different labels, thus indicating that it is not possible to match these two vertices. However, the initial labelling must not remove solutions, *i.e.*, if a vertex u belongs to the domain of a variable x_v associated with a vertex v, then u and v must be associated with a same label.

More formally, let us define the bipartite graph $G_{cc} = (\mathit{Vertices}, E_{cc})$ such that $E_{cc} = \{(u, u') \in V \times V' \mid u' \in D(x_u)\}$. If two vertices are connected by a path in G_{cc}, then they must have the same label, otherwise they can have different labels. Hence, the initial labelling may be built by computing the set of connected components of G_{cc}, and assigning the same label to all vertices within a same connected component. The set of connected components of G_{cc} may be computed by a simple search in $\mathcal{O}(\#E_{cc})$.

Also, each time the domain of a variable x_u is reduced to a singleton $\{v\}$, we can remove v from the domains of all other variables and then relabel both u and v with a new label, different from all other labels. In this case, it is no longer necessary to iterate the relabelling process on these two vertices.

5 Illustration of IDL(d)-filtering for $d = \infty$ and $d = 1$

We now illustrate IDL(d)-filtering on the *gip* global constraint instance of Figure 1 for $d = \infty$ and $d = 1$. As the two graphs are isomorphic, we only display labels computed for vertices of V. Also, for reasons of space, when several vertices have a same label l, we group all these vertices within a same set S and denote by $\alpha(S) = l$ the fact that every vertex of the set S is labelled by l.

5.1 Illustration of IDL(∞)-filtering

At step 0, each vertex is labelled by $\emptyset$:

$$\alpha_0(\{A, B, C, D, E, F, G, H, I, J\}) = \quad \emptyset$$

After the first relabelling step, we have

$$\begin{array}{lcllll}
\beta^1_{+\infty}[\alpha_\emptyset](\{A, D, F\}) & = & \{ & 0 : \{\{\emptyset^1\}\}, 1 : \{\{\emptyset^4\}\}, 2 : \{\{\emptyset^4\}\}, 3 : \{\{\emptyset^1\}\} & \} & \text{renamed to} \quad a \\
\beta^1_{+\infty}[\alpha_\emptyset](\{B, C, E, G, I\}) & = & \{ & 0 : \{\{\emptyset^1\}\}, 1 : \{\{\emptyset^3\}\}, 2 : \{\{\emptyset^4\}\}, 3 : \{\{\emptyset^2\}\} & \} & \text{renamed to} \quad b \\
\beta^1_{+\infty}[\alpha_\emptyset](\{H\}) & = & \{ & 0 : \{\{\emptyset^1\}\}, 1 : \{\{\emptyset^4\}\}, 2 : \{\{\emptyset^5\}\} & \} & \text{renamed to} \quad c \\
\beta^1_{+\infty}[\alpha_\emptyset](\{J\}) & = & \{ & 0 : \{\{\emptyset^1\}\}, 1 : \{\{\emptyset^3\}\}, 2 : \{\{\emptyset^3\}\}, 3 : \{\{\emptyset^3\}\} & \} & \text{renamed to} \quad d
\end{array}$$

The domain of x_H is reduced to $\{H'\}$ and the domain of x_J is reduced to $\{J'\}$ so that H, H', J, and J' are no longer relabelled. Then after the second relabelling step, we have

$$\begin{array}{lclllllr}
\beta^2_{+\infty}[\alpha_\emptyset](\{A\}) & = & \{ & 0:\{\{a^1\}\}, & 1:\{\{a^2,b^2\}\}, & 2:\{\{b^2,c^1,d^1\}\}, & 3:\{\{b^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{B\}) & = & \{ & 0:\{\{b^1\}\}, & 1:\{\{a^2,d^1\}\}, & 2:\{\{a^1,b^2,c^1\}\}, & 3:\{\{b^2\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{C\}) & = & \{ & 0:\{\{b^1\}\}, & 1:\{\{a^2,b^1\}\}, & 2:\{\{a^1,b^2,c^1\}\}, & 3:\{b^1,d^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{D\}) & = & \{ & 0:\{\{a^1\}\}, & 1:\{\{a^1,b^2,c^1\}\}, & 2:\{\{b^4\}\}, & 3:\{\{d^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{E\}) & = & \{ & 0:\{\{b^1\}\}, & 1:\{\{b^2,c^1\}\}, & 2:\{\{a^3,b^1\}\}, & 3:\{\{b^1,d^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{F\}) & = & \{ & 0:\{\{a^1\}\}, & 1:\{\{a^1,b^1,c^1,d^1\}\}, & 2:\{\{a^1,b^3\}\}, & 3:\{\{b^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{G\}) & = & \{ & 0:\{\{b^1\}\}, & 1:\{\{a^1,b^2\}\}, & 2:\{\{a^1,b^1,c^1,d^1\}\}, & 3:\{\{a^1,b^1\}\} & \} \\
\beta^2_{+\infty}[\alpha_\emptyset](\{I\}) & = & \{ & 0:\{\{b^1\}\}, & 1:\{\{b^1,c^1,d^1\}\}, & 2:\{\{a^2,b^2\}\}, & 3:\{\{a^1,b^1\}\} & \}
\end{array}$$

Every vertex has a different label so that the domain of every variable has been reduced to a singleton and relabelling can be stopped.

5.2 Illustration of IDL(1)-filtering

At step 0, each vertex is labelled by $\emptyset$:

$$\alpha_\emptyset(\{A,B,C,D,E,F,G,H,I,J\}) = \emptyset$$

After the first relabelling step, we have

$$\begin{array}{lclll}
\beta^1_1[\alpha_\emptyset](\{A,D,F,H\}) & = & \{ \; 0:\{\{\emptyset^1\}\}, & 1:\{\{\emptyset^4\}\} \; \} \text{ renamed to} & a \\
\beta^1_1[\alpha_\emptyset](\{B,C,E,G,I,J\}) & = & \{ \; 0:\{\{\emptyset^1\}\}, & 1:\{\{\emptyset^3\}\} \; \} \text{ renamed to} & b
\end{array}$$

After the second relabelling step, we have

$$\begin{array}{lclll}
\beta^2_1[\alpha_\emptyset](\{A,D,F,H\}) & = & \{ \; 0:\{\{a^1\}\}, & 1:\{\{a^2,b^2\}\} \; \} \text{ renamed to} & c \\
\beta^2_1[\alpha_\emptyset](\{B,C\}) & = & \{ \; 0:\{\{b^1\}\}, & 1:\{\{b^1,a^2\}\} \; \} \text{ renamed to} & d \\
\beta^2_1[\alpha_\emptyset](\{E,G,I,J\}) & = & \{ \; 0:\{\{b^1\}\}, & 1:\{\{a^1,b^2\}\} \; \} \text{ renamed to} & e
\end{array}$$

After the third relabelling step, we have

$$\begin{array}{lclll}
\beta^3_1[\alpha_\emptyset](\{A\}) & = & \{ \; 0:\{\{c^1\}\}, & 1:\{\{c^2,d^2\}\} \; \} \text{ renamed to} & f \\
\beta^3_1[\alpha_\emptyset](\{B,C\}) & = & \{ \; 0:\{\{d^1\}\}, & 1:\{\{c^2,e^1\}\} \; \} \text{ renamed to} & g \\
\beta^3_1[\alpha_\emptyset](\{D,F\}) & = & \{ \; 0:\{\{c^1\}\}, & 1:\{\{c^2,d^1,e^1\}\} \; \} \text{ renamed to} & h \\
\beta^3_1[\alpha_\emptyset](\{E,J\}) & = & \{ \; 0:\{\{e^1\}\}, & 1:\{\{c^1,d^1,e^1\}\} \; \} \text{ renamed to} & i \\
\beta^3_1[\alpha_\emptyset](\{G,I\}) & = & \{ \; 0:\{\{e^1\}\}, & 1:\{\{c^1,e^2\}\} \; \} \text{ renamed to} & j \\
\beta^3_1[\alpha_\emptyset](\{H\}) & = & \{ \; 0:\{\{c^1\}\}, & 1:\{\{c^2,e^2\}\} \; \} \text{ renamed to} & k
\end{array}$$

The domain of x_A is reduced to $\{A'\}$ and the domain of x_H is reduced to $\{H'\}$ so that A, A', H, and H' are no longer relabelled. Then, after the fourth relabelling step, we have

$$\begin{array}{lclll}
\beta^4_1[\alpha_\emptyset](\{B,C\}) & = & \{ \; 0:\{\{g^1\}\}, & 1:\{\{f^1,h^1,i^1\}\} \; \} \text{ renamed to} & l \\
\beta^4_1[\alpha_\emptyset](\{D\}) & = & \{ \; 0:\{\{h^1\}\}, & 1:\{\{f^1,g^1,j^1,k^1\}\} \; \} \text{ renamed to} & m \\
\beta^4_1[\alpha_\emptyset](\{E\}) & = & \{ \; 0:\{\{i^1\}\}, & 1:\{\{g^1,j^1,k^1\}\} \; \} \text{ renamed to} & n \\
\beta^4_1[\alpha_\emptyset](\{F\}) & = & \{ \; 0:\{\{h^1\}\}, & 1:\{\{f^1,g^1,i^1,k^1\}\} \; \} \text{ renamed to} & o \\
\beta^4_1[\alpha_\emptyset](\{G\}) & = & \{ \; 0:\{\{j^1\}\}, & 1:\{\{h^1,i^1,j^1\}\} \; \} \text{ renamed to} & p \\
\beta^4_1[\alpha_\emptyset](\{I\}) & = & \{ \; 0:\{\{j^1\}\}, & 1:\{\{i^1,j^1,k^1\}\} \; \} \text{ renamed to} & q \\
\beta^4_1[\alpha_\emptyset](\{J\}) & = & \{ \; 0:\{\{i^1\}\}, & 1:\{\{g^1,h^1,j^1\}\} \; \} \text{ renamed to} & r
\end{array}$$

The domains of all variables, except x_B and x_C are reduced to singletons so that only B, B', C and C' are relabelled. After the fifth relabelling step, we have

$$\begin{array}{rcl} \beta_1^5[\alpha_\emptyset](\{B\}) & = & \{ \quad 0 : \{\{l^1\}\}, \quad 1 : \{\{f^1, o^1, r^1\}\} \quad \} \\ \beta_1^5[\alpha_\emptyset](\{C\}) & = & \{ \quad 0 : \{\{l^1\}\}, \quad 1 : \{\{f^1, m^1, o^1\}\} \quad \} \end{array}$$

The domain of every variable is reduced to a singleton and relabelling can be stopped.

5.3 Discussion

On this example, both IDL(1) and IDL(∞) reduce all domains to singletons, but they perform a different number of iterations: two for IDL(∞) and five for IDL(1). However, as pointed out in section 4, the complexity of one relabelling step of IDL(∞) is an order higher than the complexity of one relabelling step of IDL(1) ($\mathcal{O}(n \cdot p)$ instead of $\mathcal{O}(p)$).

The label-filtering introduced in [SS04] corresponds to the first two iterations of IDL(∞)-filtering whereas the ILL-filtering introduced in [SS07] exactly corresponds to IDL(1)-filtering. We have experimentally compared these two filtering algorithms in [SS07] and we have shown that both ILL-filtering and Label-filtering are nearly always able to reduce all domains to singleton (for non automorphic graphs), but that Label-filtering is an order slower than ILL-filtering.

6 Experimental results

In this section, we compare different instantiations of IDL(d)-filtering with state-of-the-art algorithms, *i.e.*, Nauty, Saucy and Sparetest (an improved version of Nauty for sparse graphs which has been sent to us by B. McKay in a personal communication). We do not include tree search based algorithms neither CP approaches based on the CSP model introduced in 2.3 as they are not competitive.

Nauty, Saucy and Sparetest have been designed for finding automorphisms in a graph and compactly generate the whole group of automorphisms. They can be used to solve the GIP as the compact representation generated by these algorithms defines a signature such that two graphs have the same signature if and only if they are isomorphic. However, one should keep in mind that these algorithms actually solve a more difficult problem than the GIP.

IDL(d)-filtering only ensures a partial consistency. Hence, IDL(d)-filtering has been integrated within a branch and propagate tree search. In this section, IDL(d) refers to a branch and propagate tree search which performs IDL(d)-filtering at each node of the search.

We only consider feasible GIP instances, such that the two graphs are isomorphic, as non feasible instances are usually more easily solved. For each considered approach, we measure the CPU time spent to solve the problem. We have considered three kinds of graphs: randomly generated graphs, sparse graphs with bounded degrees, and regular sparse graphs. We have also made experiments on graphs that are randomly generated using a power law distribution of degrees $P(d = k) = k^{-\lambda}$: this distribution corresponds to scale-free networks which model a wide range of real networks, such as social, Internet, or neural networks [Bar03]. We obtained very similar results on these graphs so that we do not report these results in this paper.

All results have been obtained on a 1.6Ghz Pentium M with 512Mb of RAM.

6.1 Results on randomly generated graphs

We have randomly generated graphs with a Nauty tool called genrang. We have considered graphs with different sizes (from 1000 to 6500 vertices), and graphs with different edge densities (from 1% to 50%).

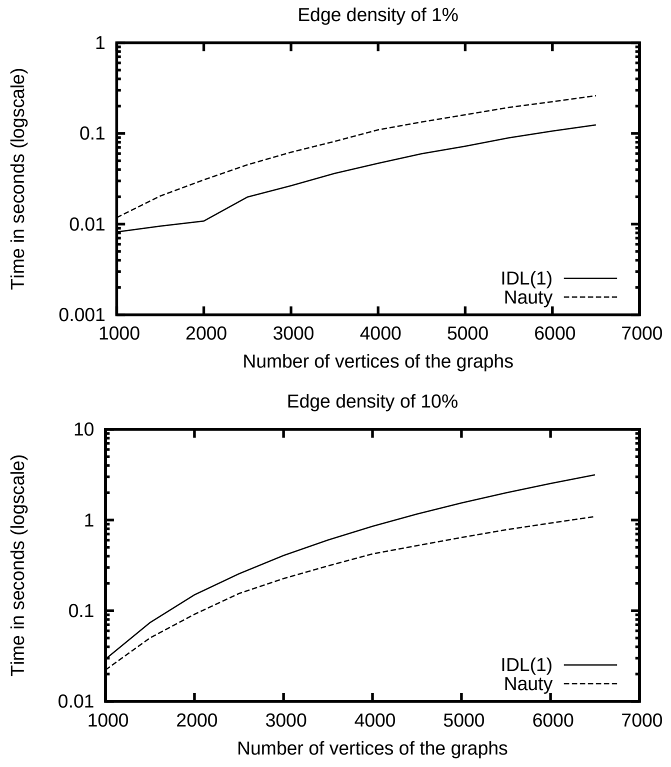

Figure 2: Run time w.r.t. the number of vertices for two different edge densities: 1% (top) and 10% (bottom). Average results on 100 graphs for each size of graphs and edge density.

We first compare the best two approaches for these graphs, *i.e.*, Nauty and IDL(1), and then discuss performances of other approaches, *i.e.*, IDL(k), Saucy, and Sparetest.

Comparison of Nauty and IDL(1). Figure 2 compares Nauty and IDL(1) when varying the number of vertices from 1000 to 6500 to study scale-up properties, for two different edge densities: 1% and 10%. It shows that, IDL(1) is better than Nauty when the edge density is 1%, whereas Nauty is better than IDL(1) when the edge density is 10%.

Figure 3 compares Nauty and IDL(1) on graphs having 1000 vertices, when varying the edge density from 1% to 50% (we do not report experimental results on graphs with higher densities as, in this case, one has better consider complementary graphs). It shows that IDL(1) is slightly better than Nauty on low density graphs, but that Nauty clearly becomes better than IDL(1) when increasing the density. Finally, when the density is 50%, Nauty is twice as fast as IDL(1).

Note that run time differences between Nauty and IDL(1) mainly come from data structures and implementation issues as both approaches are based on a very similar iterative relabelling based on labels of direct neighbors.

Comparison of different instantiations of IDL(d). On all these instances, IDL(1)-filtering is strong enough to always reduce all domains to singletons so it is never necessary to develop a search tree.

The average number of relabelling steps performed by IDL(1) before reaching the fix-point depends on the number of vertices of the graphs: the larger the size of the graphs, the lower the number of relabelling steps. For example, the average number of relabelling steps is 3.4 (resp. 2.9 and 2.1) on graphs with 200

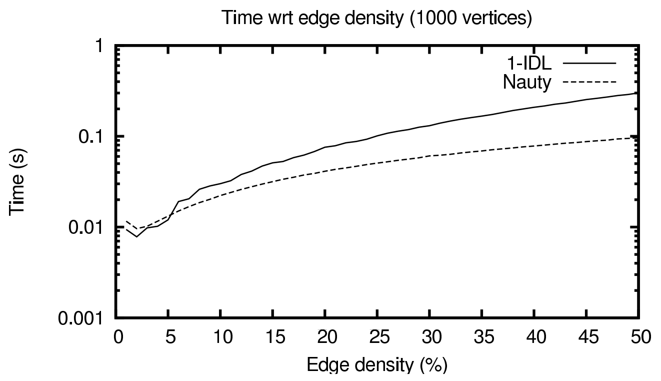

Figure 3: Run time w.r.t. edge density for graphs having 1000 vertices. Average results on 100 graphs for each edge density. Note the log scale on the y-axis.

(resp. 400 and 600) vertices; for graphs with more than 800 vertices, the number of relabelling steps is always equal to 2.

Hence, on these graphs, increasing the value of the d parameter increases CPU-times so that IDL(d) with $d > 1$ is not competitive with IDL(1).

Comparison with Saucy and Sparetest. Saucy and Sparetest have been designed to handle sparse graphs with very low edge densities. Hence, on this first set of randomly generated graphs they are not competitive with Nauty and IDL(1).

6.2 Sparse graphs with bounded degrees

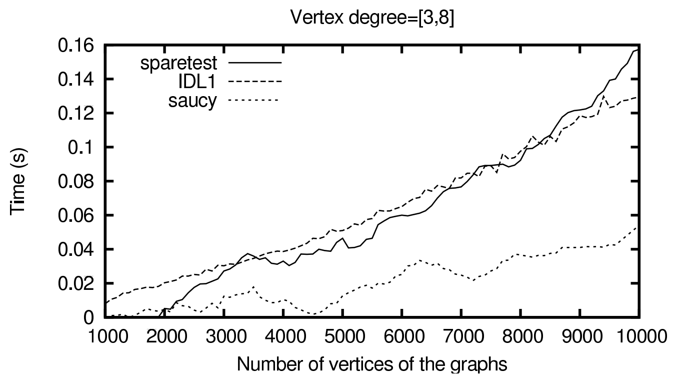

Figure 4: Run time w.r.t. the number of vertices on sparse graphs with degrees bounded between 3 and 8.

We now consider sparse graphs such that vertex degrees are bounded between 3 and 8. The density of these graphs is equal to 1% for graphs with 1000 vertices and 0.1% for graphs with 10000 vertices. On these sparse graphs, Nauty is not competitive with Saucy and Sparetest (even when using the best invariant for this kind of graphs, *i.e.*, *twopaths*). For example, on graphs with 2000 vertices, Nauty is more than 4 times as slow as other approaches; on graphs with 3000 vertices, it is more than 10 times as slow.

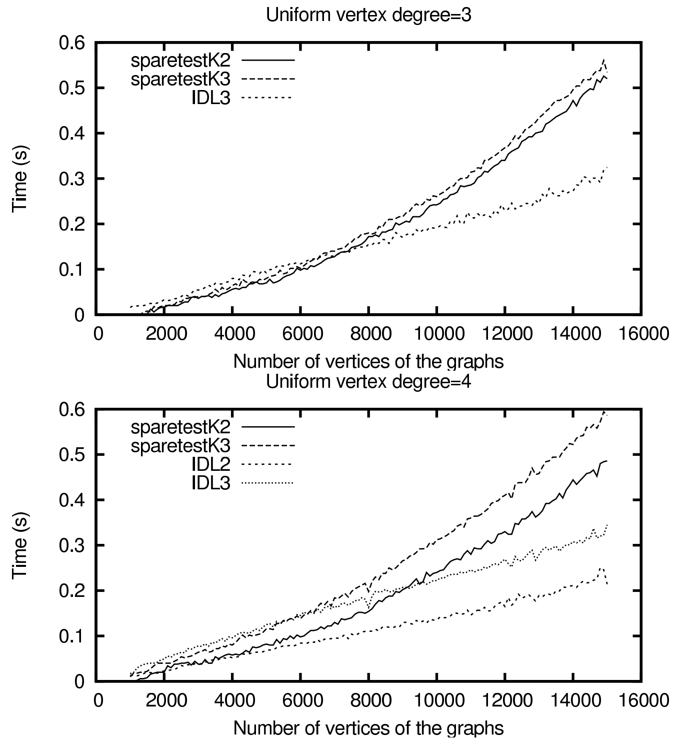

Figure 5: Results for regular graphs for two different vertex degrees: 3 (top) and 4 (bottom).

Comparison of Saucy, Sparetest, and IDL(1). Figure 4 shows that on these graphs Sparetest and IDL(1) have very similar performances, whereas Saucy is better. The difference between Saucy and both Sparetest and IDL(1) increases when considering larger (and sparser) graphs so that for graphs with 10000 vertices (the edge density of which is 0.1%), Saucy is more than twice as fast.

Comparison of different instantiations of IDL(d). Like for randomly generated graphs, IDL(1)-filtering is strong enough to reduce all domains to singletons so that it is never necessary to develop a search tree. Hence, on these graphs, increasing the value of the d parameter increases CPU-times so that IDL(d) with $d > 1$ is not competitive with IDL(1).

6.3 Results on regular sparse graphs

We now compare approaches on regular sparse graphs, *i.e.*, graphs such that all vertices have the same degree. We have considered two degree values, *i.e.*, 3 and 4, and we have generated regular graphs which have from 1000 to 15000 vertices.

Performances of Nauty, Saucy, and IDL(1). Both Nauty and Saucy are not competitive to solve these regular graphs because all vertices have the same number of neighbors. For example, Nauty (resp. Saucy) spends more than one second (resp. more than three seconds) to solve instances with 1000 vertices and a degree of 3.

On these instances, IDL(1) behaves like Nauty and Saucy: each vertex has exactly the same number of neighbors so that IDL(1)-filtering is not able to reduce any domain without developing a search tree.

Comparison of Sparetest with IDL(d). The basic version of Sparetest is not competitive for solving regular graphs. However, an option of Sparetest can be used to add a vertex invariant which improves its performances on regular graphs. The idea is to start the iterative vertex partition refinement from an initial partition which groups together all vertices that have the same number of vertices at a distance smaller or equal to a given parameter k (this roughly corresponds to applying once the relabelling β_k, but then iteratively applying the relabelling β_1). When solving regular sparse graphs with Sparetest, the best results are obtained with $k = 2$ and $k = 3$ (options "-k2" and "-k3").

Figure 5 compares the two variants of Sparetest, with $k = 2$ and $k = 3$, with IDL(2) and IDL(3) on regular graphs. It shows us that when the degree of the vertices is set to 3, the best results are obtained by IDL(3) (results obtained by IDL(2) are not displayed because they are not competitive); when it is set to 4, the best results are obtained by IDL(2). We have also performed experiments with regular graphs with higher degrees than 4, and note very similar results to those obtained when degree=4, *i.e.*, IDL(2) is the best performing approach on these graphs.

7 Conclusion

We have introduced IDL(d)-filtering, a new parametric filtering algorithm dedicated to the graph isomorphism problem. The d parameter determines the strength of the filtering: the larger d, the stronger the filtering.

When $d = 1$, this algorithm basically follows the same partition refinement procedure as the one introduced in Nauty and used in Saucy and Sparetest, which are improved versions of Nauty's dedicated to sparse graphs. Experimental results have shown us that IDL(1) exhibits nice properties with respect to edge density variations:

- on dense graphs, the best performing approach is Nauty; on these graphs, IDL(1) is competitive with Nauty, even though it is slower on the densest graphs, whereas neither Saucy nor Sparetest are competitive;
- on sparse graphs, the best performing approaches are Saucy and Sparetest; on these graphs, IDL(1) is competitive with Saucy and Sparetest, even though it is slower on the sparsest graphs, whereas Nauty is not competitive.

On randomly generated graphs, where the vertices have different degrees, IDL(1)-filtering is strong enough to reduce all domains to singletons. Therefore, on these graphs, using stronger filterings, such as IDL(2) or IDL(3), only increases CPU time.

However, on regular graphs, approaches using filterings based on the direct neighborhood of vertices —such as IDL(1), Nauty, or Saucy— are not efficient as all vertices have the same degree. On these instances, using stronger filterings, such as IDL(2) or IDL(3), actually improves the solution process.

IDL(d)-filtering has been defined for non directed graphs. However, it could be easily extended to directed graphs. A first possibility is to exploit the fact that two directed graphs are isomorphic only if their non-directed counterparts also are isomorphic. However, this may lead to poor quality filterings as edge orientations are not taken into account. Another possibility to extend our work to directed graphs consists in adapting the β_d relabelling function to take into account edge orientations. For example, the β_1 relabelling function can be adapted by computing separately the multiset of successor labels and the multiset of predecessor labels.

IDL(d)-filtering is based on isomorphic-consistent labellings and relabellings, that exploit distance-based invariant properties. This idea has been extended to the subgraph isomorphism problem in [ZDS+07]: like in IDL(1)-filtering, nodes are labelled by some invariant property, and labels are iteratively extended by considering labels of adjacent nodes; however, in the case of subgraph isomorphism, label compatibilities are expressed with respect to a partial order instead of an equivalence relation.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison Wesley, 1974.

[Bar03] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, 2003.

[BH03] Christian Bessière and Pascal Van Hentenryck. To be or not to be... a global constraint. *CP'03, Kinsale, Ireland*, pages 789–794, 2003.

[CFSV04] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[CS03] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In *5th International Conference on Case-Based Reasoning (ICCBR 2003)*, volume Lecture Notes in Artificial Intelligence Nu. 2689 - Springer-Verlag, pages 80–95, 2003.

[DLSM04] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. *DAC*, pages 530–554, 2004.

[For96] Scott Fortin. The graph isomorphism problem. Technical report, Dept of Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1996.

[FSV01] Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199. Cuen, 2001.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to The Theory of NP-Completness.* W.H. Freeman, San Francisco, 1979.

[HSD92] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3):113–159, 1992.

[HW74] John E. Hopcroft and Jin-Kue Wong. Linear time algorithm for isomorphism of planar graphs. 6^{th} *Annu. ACM Symp. theory of Comput.*, pages 172–184, 1974.

[ILO00] ILOG,S.A. *ILOG Solver 5.0 User's Manual and Reference Manual.* 2000.

[LO00] François Laburthe and OCRE. CHOCO: implementing a CP kernel. In *Proc. of the CP'2000 workshop on techniques for implementing constraint programming systems, Singapore*, 2000.

[Luk82] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer System Science*, pages 42–65, 1982.

[McG79] James J. McGregor. Relational consistency algorithms and their applications in finding subgraph and graph isomorphisms. *Information Science*, 19:229–250, 1979.

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[Pug05] Jean-François Puget. Automatic detection of variable and value symmetries. In *Principles and Practice of Constraint Programming - CP 2005*, volume 3709, pages 475–489, 2005.

[Rég95] Jean-Charles Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique.* PhD thesis, Univ. Montpellier II, 1995.

[SD76] Douglas Schmidt and Larry Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM (JACM)*, 23(3):433–445, July 1976.

[SS04] Sébastien Sorlin and Christine Solnon. A global constraint for graph isomorphism problems. In *the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004)*, pages 287–301. Springer-Verlag, Avril 2004.

[SS07] Sébastien Sorlin and Christine Solnon. A new filtering algorithm for the graph isomorphism problem, April 2007. Contribution to the chapter "Constraint Propagation and Implementation" of the book "Trends in Constraint Programming" edited by Frédéric Benhamou, Narendra Jussien and Barry O'Sullivan, pages 103-107, ISTE Publisher.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Ull76] Jeffrey D. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association of Computing Machinery*, 23(1):31–42, 1976.

[ZDD06] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Elimination des symétries pour l'appariement de graphes. In Laurent Henocque, editor, *JFPC'06, Deuxièmes Journées Francophones de Programmation par Contraintes*, pages 357–367, 2006.

[ZDS$^+$07] S. Zampelli, Y. Deville, C. Solnon, S. Sorlin, and P. Dupont. Filtering for subgraph isomorphism. In *Principles and Practice of Constraint Programming (CP'2007)*, number 4741 in LNCS, pages 728–742. Springer, 2007.