Artificial Intelligence 174 (2010) 850–864

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

AllDifferent-based filtering for subgraph isomorphism

Christine Solnon

Université de Lyon, Université Lyon 1, LIRIS, UMR5205 CNRS, F-69622, France

ARTICLE INFO

Article history:
Received 24 December 2009
Received in revised form 3 May 2010
Accepted 3 May 2010
Available online 6 May 2010

Keywords:
Subgraph isomorphism
Constraint programming
All different constraint

ABSTRACT

The subgraph isomorphism problem involves deciding if there exists a copy of a pattern graph in a target graph. This problem may be solved by a complete tree search combined with filtering techniques that aim at pruning branches that do not contain solutions. We introduce a new filtering algorithm based on local *all different* constraints. We show that this filtering is stronger than other existing filterings — *i.e.*, it prunes more branches — and that it is also more efficient — *i.e.*, it allows one to solve more instances quicker.

1. Introduction

Graphs are widely used in real-life applications to represent structured objects such as, for example, molecules, images, or biological networks. In many of these applications, one looks for a copy of a pattern graph into a target graph [4]. This problem, known as subgraph isomorphism, is NP-complete in the general case [6].

Subgraph isomorphism problems may be solved by a systematic exploration of the search space composed of all possible injective matchings from the set of pattern nodes to the set of target nodes: starting from an empty matching, one incrementally extends a partial matching by matching a non-matched pattern node to a non-matched target node until either some edges are not matched by the current matching (the search must backtrack to a previous choice point and go on with another extension) or all pattern nodes have been matched (a solution has been found). To reduce the search space, this exhaustive exploration is combined with filtering techniques that aim at removing candidate couples of non-matched pattern-target nodes. Different levels of filtering may be considered; some are stronger than others (they remove more nodes), but also have higher time complexities.

In this paper, we describe and compare existing filtering algorithms for the subgraph isomorphism problem, and we introduce a new filtering algorithm which is stronger. We experimentally evaluate this new filtering algorithm on a wide benchmark of instances, and we show that it is much more efficient on many instances.

2. Definitions and notations

A *graph* $G = (N, E)$ consists of a *node set* N and an *edge set* $E \subseteq N \times N$, where an edge (u, u') is a couple of nodes. The set of neighbors of a node u is denoted $adj(u)$ and is defined by $adj(u) = \{u' \mid (u, u') \in E\}$. In this paper, we implicitly consider non-directed graphs, such that $(u, u') \in E \Leftrightarrow (u', u) \in E$. The extension of our work to directed graphs is discussed in Section 5.

A *subgraph isomorphism problem* between a pattern graph $G_p = (N_p, E_p)$ and a target graph $G_t = (N_t, E_t)$ consists in deciding whether G_p is isomorphic to some subgraph of G_t. More precisely, one should find an injective matching $f : N_p \rightarrow N_t$, that associates a different target node to each pattern node, and that preserves pattern edges, *i.e.*,

E-mail address: christine.solnon@liris.cnrs.fr.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2010.05.002

$$\forall(u,u') \in E_p, \quad (f(u), f(u')) \in E_t$$

The function f is called a *subisomorphism function*.

Note that the subgraph is not necessarily induced so that two pattern nodes that are not linked by an edge may be matched to two target nodes which are linked by an edge. This problem is also called subgraph monomorphism or subgraph matching in the literature.

In the following, we assume $G_p = (N_p, E_p)$ and $G_t = (N_t, E_t)$ to be the underlying instance of subgraph isomorphism problem, and we assume without loss of generality that $N_p \cap N_t = \emptyset$. We usually denote u or u' (resp. v or v') nodes of G_p (resp. G_t).

We denote $\#S$ the cardinality of a set S. We also define $N = N_p \cup N_t$, $E = E_p \cup E_t$, $n_p = \#N_p$, $n_t = \#N_t$, $e_p = \#E_p$, $e_t = \#E_t$, and d_p and d_t the maximal degrees of the graphs G_p and G_t.

3. Filtering for subgraph isomorphism

Subgraph isomorphism problems may be modeled as constraint satisfaction problems in a very straightforward way. In this section, we first show how to model and solve subgraph isomorphism problems within a constraint satisfaction framework. Then, we describe different filtering algorithms for subgraph isomorphism in Sections 3.3 to 3.6, and we compare them in Section 3.7.

3.1. Modeling and solving subgraph isomorphism by means of constraints

A constraint satisfaction problem (CSP) is defined by a set of variables, such that each variable is associated with a domain (*i.e.*, the set of values that it may be assigned to), and a set of constraints (*i.e.*, relations that restrict the set of values that may be assigned to some variables simultaneously). Solving a CSP involves finding an assignment of values to all variables such that all constraints are satisfied.

A subgraph isomorphism problem may be modeled as a CSP by associating a variable (denoted x_u) with every pattern node u. The domain of a variable x_u (denoted D_u) contains the set of target nodes that may be matched to u. Intuitively, assigning a variable x_u to a value v corresponds to matching the pattern node u to the target node v. The domain D_u is usually reduced to the set of target nodes the degree of which is higher or equal to the degree of u as node u may be matched to node v only if $\#adj(u) \leqslant \#adj(v)$.

Constraints ensure that the assignment of variables to values corresponds to a subisomorphism function. There are two kinds of constraints:

- edge constraints ensure that pattern edges are preserved, *i.e.*,

$$\forall(u,u') \in E_p, \quad (x_u, x_{u'}) \in E_t$$

- difference constraints ensure that the assignment corresponds to an injective function, *i.e.*,

$$\forall(u,u') \in N_p^2, \quad u \neq u' \quad \Rightarrow \quad x_u \neq x_{u'}$$

Within this framework, solving a subgraph isomorphism problem involves finding an assignment of the variables that satisfies all constraints. We shall consider that a variable is assigned whenever its domain is reduced to a singleton, *i.e.*, $D_u = \{v\} \Leftrightarrow x_u = v$.

Subgraph isomorphism problems modeled as CSPs may be solved by building a search tree that explores all possible variable assignments until finding a solution. The size of this search tree may be reduced by using filtering techniques which propagate constraints to remove values from domains.

We briefly recall some basic principles of constraint propagation in Section 3.2. Then, we describe different filtering techniques that may be used to solve subgraph isomorphism problems in Sections 3.3 to 3.6. Note that some of these filterings (*i.e., FC*(*Diff*), *GAC*(*AllDiff*), *FC*(*Edges*), and *AC*(*Edges*)) are generic constraint propagation techniques that may be used to solve any CSP whereas some others (*i.e., LV2002* and *ILF*(k)) are dedicated to the subgraph isomorphism problem.

3.2. Recalls on constraint propagation

Constraint propagation aims at filtering variable domains by removing inconsistent values, that is, values that do not belong to any solution. This constraint propagation step may be done at each choice point of the search. If it removes all values in the domain of a variable, then the search can backtrack to a previous choice.

A pioneering work for constraint propagation has been done in 1972 by Waltz for a scene drawing application [19]. Since then, many different constraint propagation algorithms have been proposed. These algorithms achieve different partial consistencies and also have different time and space complexities. In this section, we do not aim at describing all existing propagation algorithms. We only briefly describe two basic and well-known generic techniques, that is, *forward-checking* and *maintaining arc-consistency*. The reader may refer to [17,10] for more information.

Forward-checking The basic idea of forward-checking is to propagate all constraints involving a variable just after its assignment in order to remove from the domains of the non-assigned variables any value which is not consistent with this assignment. More precisely, after the assignment of x_i to v_i, one propagates binary constraints between x_i and any non-assigned variable x_j by removing from the domain of x_j any value v_j such that the assignment $\{(x_i, v_i), (x_j, v_j)\}$ violates the constraint holding between x_i and x_j. When constraints have arities greater than two, one may propagate constraints such that all variables but one are assigned.

Maintaining arc-consistency A stronger filtering, but also a more expensive one, is obtained by maintaining *arc-consistency*, also called 2-consistency. Roughly speaking, a binary CSP is arc-consistent if each value v_i in the domain of a variable x_i has at least one *support* in the domain of every other variable, thus ensuring that if x_i is assigned to v_i then each other variable still has at least one consistent value in its domain. More precisely, given a variable $x_i \in X$ and a value $v_i \in D(x_i)$, a support of (x_i, v_i) for a variable x_j is a value $v_j \in D(x_j)$ such that the partial assignment $\{(x_i, v_i), (x_j, v_j)\}$ is consistent. A binary CSP (X, D, C) is arc-consistent if every value in every domain has at least one support in the domain of each other variable.

To maintain arc-consistency while constructing a partial assignment A, we filter variable domains after each variable assignment by removing non-supported values. Such a filtering must be repeated until no more domain is reduced: as soon as a value is removed, we must check that this value is not the only support of some other values. There exist many different algorithms for ensuring arc-consistency, which exhibit different time and space complexities. For instance, a widely used algorithm for achieving arc consistency of a set of binary constraints is AC4 [14] whose time and space complexities are $\mathcal{O}(ck^2)$, where c is the number of constraints and k the maximum domain size. Although AC4 is worst-case optimal in time, it always reaches this worst case because of its expensive initialisation phase; many improvements have been proposed since AC4, leading for example to AC6, AC7 and AC2001 (see [17] for more details). Arc consistency may also be generalized to non-binary CSPs. In this case, it is called *generalized arc consistency*.

3.3. Propagation of difference constraints (FC(Diff) and GAC(AllDiff))

Difference constraints may be propagated by forward-checking (denoted *FC*(*Diff*)): each time a pattern node u is matched to a target node v, *FC*(*Diff*) removes v from the domains of all non-matched nodes. This may be done in $\mathcal{O}(n_p)$.

FC(*Diff*) propagates each binary difference constraint separately. A stronger filtering may be obtained by propagating the whole set of difference constraints in order to ensure that all pattern nodes can be assigned to different target nodes. More precisely, achieving the generalized arc consistency of a global *AllDifferent* constraint (denoted *GAC*(*AllDiff*)) removes from the domain of every pattern node u every target node v such that, when u is matched to v, the other pattern nodes cannot be matched to all different target nodes. In [16], Régin has shown how to use the matching algorithm of Hopcroft and Karp for achieving *GAC*(*allDiff*). The time complexity of this algorithm is $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Example 1. Let us consider four variables x_1, x_2, x_3, and x_4 such that $D_1 = \{a\}$, $D_2 = D_3 = \{a, b, c\}$, and $D_4 = \{a, b, c, d\}$.

FC(*Diff*) removes a from the domains of x_2, x_3, and x_4.

GAC(*AllDiff*) also removes a from the domains of x_2, x_3, and x_4. It further removes b and c from the domain of x_4 as if x_4 is assigned to b or c, then x_2 cannot be assigned to a value different from both x_3 and x_4.

3.4. Propagation of edge constraints (FC(Edges) and AC(Edges))

Edge constraints may be propagated by forward checking (denoted *FC*(*Edges*)): each time a pattern node u is matched to a target node v, *FC*(*Edges*) removes from the domain of every node adjacent to u any target node that is not adjacent to v. This may be done in $\mathcal{O}(d_p \cdot n_t)$.

One may go one step further and maintain the arc consistency of edges constraints (denoted *AC*(*Edges*)) so that

$$\forall (u, u') \in E_p, \quad \forall v \in D_u,\ \exists v' \in D_{u'},\ (v, v') \in E_t$$

As a CSP modeling a subgraph isomorphism problem has e_p edge constraints and the maximum domain size is n_t, the time complexity of *AC*(*Edges*) is $\mathcal{O}(e_p \cdot n_t^2)$ when using AC4.

Example 2. Let us consider the subgraph isomorphism problem displayed in Fig. 1. Note that this instance has no solution as G_p cannot be mapped into a subgraph of G_t. Let us suppose that node 3 has been matched to node E so that $D_3 = \{E\}$, and that E has been removed from the domains of all other pattern nodes (*e.g.*, by *FC*(*Diff*) or *GAC*(*AllDiff*)).

FC(*Edges*) removes B, C, and F from the domains of nodes 1, 2, and 4 because B, C, and F are not adjacent to E whereas 1, 2, and 4 are adjacent to 3.

Like *FC*(*Edges*), *AC*(*Edges*) removes B, C, and F from the domains of nodes 1, 2, and 4. It is also able to remove G from the domain of 1 as the matching $(1, G)$ has no support for the edge $(1, 4)$. Indeed, none of the adjacent nodes of G (*i.e.*, B, F, and E) belongs to the domain of 4. For the same reasons, *AC*(*Edges*) also removes G from the domains of 2 and 4.

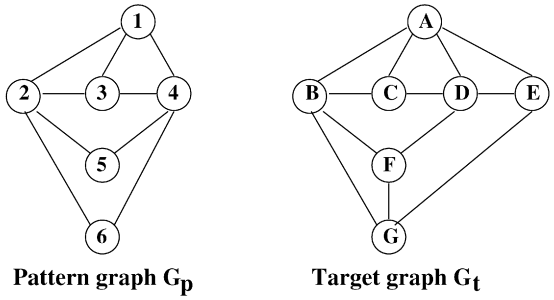

Fig. 1. Instance of subgraph isomorphism problem.

3.5. Propagation of a set of edge constraints (LV2002)

Both *FC*(*Edges*) and *AC*(*Edges*) propagate each edge constraint separately. A stronger filtering is obtained by propagating edge constraints in a more global way, *i.e.*, by propagating the fact that a whole set of nodes must be adjacent to a given node. Indeed, a pattern node u may be matched to a target node v only if the number of nodes adjacent to u is smaller or equal to the number of target nodes that are both adjacent to v and belong to domains of nodes adjacent to u (otherwise some nodes adjacent to u cannot be matched to nodes adjacent to v). Hence, Larrosa and Valiente have proposed in [12] a filtering algorithm (denoted *LV2002*) which propagates this constraint. More precisely, they define the set

$$\mathcal{F}(u,v) = \bigcup_{u' \in adj(u)} \left(D_{u'} \cap adj(v)\right)$$

$\mathcal{F}(u,v)$ is a superset of the set of nodes that may be matched to nodes that are adjacent to u if u is matched to v. Therefore, one can remove v from D_u whenever $\#\mathcal{F}(u,v) < \#adj(u)$. One can also remove v from D_u whenever there exists a pattern node $u' \in adj(u)$ such that $D_{u'} \cap adj(v) = \emptyset$, thus enforcing arc consistency of edge constraints. The *LV2002* filtering algorithm has a time complexity of $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Example 3. Let us consider the subgraph isomorphism problem displayed in Fig. 1. Let us suppose that node 3 has been matched to node E so that $D_3 = \{E\}$, and that E has been removed from the domains of all other pattern nodes (*e.g.*, by *FC*(*Diff*) or *GAC*(*AllDiff*)).

Like *AC*(*Edges*), *LV2002* removes nodes B, C, F, and G from the domains of nodes 1, 2, and 4. It is also able to remove values A and D from the domain of 1. Indeed,

$$\mathcal{F}(1,A) = (D_2 \cup D_3 \cup D_4) \cap adj(A) = \{D, E\}$$
$$\mathcal{F}(1,D) = (D_2 \cup D_3 \cup D_4) \cap adj(D) = \{A, E\}$$

As, $\#\mathcal{F}(1,A) < \#adj(1)$ and $\#\mathcal{F}(1,D) < \#adj(1)$, both A and D are removed from D_1 so that the domain of 1 becomes empty and an inconsistency is detected.

3.6. Iterated labeling filtering (ILF(k))

Zampelli et al. have proposed in [20] a filtering algorithm (called *ILF*(k)) which exploits the graph structure in a global way to compute labels that are associated with nodes and that are used to filter domains. More precisely, a compatibility relationship is defined over the set of node labels. This compatibility relationship is used to remove from the domain of a pattern node u every target node v such that the label of u is not compatible with the label of v.

ILF(k) is an iterative procedure that starts from an initial labeling. This initial labeling may be defined by node degrees. In this case, the compatibility relationship is the classical $\leqslant$ order. This labeling is used to remove from the domain of a pattern node u every target node v such that $\#adj(u) \not\leqslant \#adj(v)$ as u cannot be matched to v if u has more adjacent nodes than v.

This initial labeling is extended to filter more values. Given a labeling l and a compatibility relationship $\preccurlyeq$ between labels of l, one defines a new labeling l' such that the new label $l'(u)$ of a node u is the multiset which contains all labels of nodes adjacent to u. The compatibility relationship $\preccurlyeq'$ is such that $l'(u) \preccurlyeq' l'(v)$ if for every occurrence x of a label in $l'(u)$ there exists a different occurrence y of a label in $l'(v)$ such that $x \preccurlyeq y$. The key point relies on the computation of the new compatibility relationship $\preccurlyeq'$, which is done in $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$ thanks to the matching algorithm of Hopcroft and Karp (see [20] for more details).

Such labeling extensions are iterated. A parameter k is introduced, that determines the number of labeling extensions. Note that iterated labeling extensions may be stopped before reaching this bound k if some domain has been reduced to an

empty set, or if a fixpoint is reached — such that no more value may be filtered. The *ILF*(k) procedure has a time complexity of $\mathcal{O}(min(k, n_p \cdot n_t) \cdot n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

[20] also introduces a weaker filtering, called *ILF**(k). The idea is to approximate, at each iteration, the label compatibility relationship by a total order so that the next compatibility relation may be computed by sorting the multisets and sequentially comparing them. The time complexity of this weaker filtering is $\mathcal{O}(min(k, n_p \cdot n_t) \cdot n_p \cdot n_t \cdot d_t)$.

Example 4. Let us consider the subgraph isomorphism problem displayed in Fig. 1. The initial degree-based labeling is the labeling l such that

- $l(5) = l(6) = 2$,
- $l(1) = l(3) = l(C) = l(E) = l(F) = l(G) = 3$,
- $l(2) = l(4) = l(A) = l(B) = l(D) = 4$

and the order over this set of labels is such that

- 2 is compatible with 2, 3, and 4,
- 3 is compatible with 3 and 4,
- 4 is compatible with 4.

Hence, one can remove the target nodes C, E, F, and G from the domains of the pattern nodes 2 and 4.

The extension of this initial degree-based labeling is the labeling l' such that

- $l'(1) = l'(3) = l'(E) = l'(F) = \{\{3, 4, 4\}\}$,
- $l'(2) = l'(4) = \{\{2, 2, 3, 3\}\}$,
- $l'(5) = l'(6) = \{\{4, 4\}\}$,
- $l'(A) = \{\{3, 3, 4, 4\}\}$,
- $l'(B) = l'(D) = \{\{3, 3, 3, 4\}\}$,
- $l'(C) = \{\{4, 4, 4\}\}$,
- $l'(G) = \{\{3, 3, 4\}\}$

and the order over this set of labels is such that

- $\{\{3, 4, 4\}\}$ is compatible with $\{\{3, 3, 4, 4\}\}$ and $\{\{3, 4, 4\}\}$,
- $\{\{2, 2, 3, 3\}\}$ is compatible with $\{\{3, 3, 4, 4\}\}$ and $\{\{3, 3, 3, 4\}\}$,
- $\{\{4, 4\}\}$ is compatible with $\{\{3, 3, 4, 4\}\}$, $\{\{4, 4, 4\}\}$ and $\{\{3, 4, 4\}\}$.

As $l'(1)$ is not compatible with $l'(B)$, B is removed from D_1. For the same reasons, B, D and G are removed from the domains of nodes 1, 3, 5 and 6.

This new labeling l' can be further extended, thus removing more values, and finally proving the inconsistency of this instance.

3.7. Discussion

Most of the algorithms that have been proposed for solving the subgraph isomorphism problem may be described by means of the filtering algorithms described in Sections 3.3 to 3.6. In particular:

- McGregor [13] combines *FC*(*Diff*) and *FC*(*Edges*);
- Ullmann [18] combines *FC*(*Diff*) and *AC*(*Edges*);
- Régin [15] combines *GAC*(*AllDiff*) and *AC*(*Edges*);
- Larrosa and Valiente [12] combine *GAC*(*AllDiff*) and *LV2002*;
- Zampelli et al. combine *GAC*(*AllDiff*), *AC*(*Edges*), and *ILF*(k).

These different filterings achieve different consistencies. Some of them are stronger than others. In particular,

- *GAC*(*AllDiff*) is stronger than *FC*(*Diff*);
- *LV2002* is stronger than *AC*(*Edges*) which is stronger than *FC*(*Edges*).

However, *GAC*(*AllDiff*) and *FC*(*Diff*) are not comparable with *FC*(*Edges*), *AC*(*Edges*), *LV2002*, and *ILF*(k) as they do not propagate the same constraints.

The relations between *ILF*(k) and other filterings that propagate edge constraints (*i.e.*, *LV2002*, *AC*(*Edges*), and *FC*(*Edges*)) depend on initial domains: if the initial domain of every variable contains all target nodes, then *ILF*(k) is stronger than

LV2002, provided that the number of labeling extensions k is greater or equal to 2.[1] However, if some domains have been reduced (which is usually the case when the filtering is done at a node which is not at the root of the search tree), then *ILF*(k) is not comparable with *LV2002* and *AC*(*Edges*).

Indeed, *ILF*(k) does not exploit domains to filter values as labelings and compatibility relationships that are iteratively computed do not depend at all on domains. To allow *ILF*(k) to propagate some domain reductions, the iterative labeling extension process has been combined, before each labeling extension, with the two following steps:

- Reduction of the target graph with respect to domains: if a target node v does not belong to any domain, then this node and its incident edges are discarded from the target graph.
- Strengthening of a labeling with respect to singleton domains: if a domain D_u is reduced to a singleton $\{v\}$, then nodes u and v are labeled with a new label which is not compatible with any other label, except itself, thus preventing other pattern nodes from being matched with v.

When adding these two steps, *ILF*(k) is stronger than *FC*(*Edges*). However, it is still not comparable with *LV2002* and *AC*(*Edges*).

To propagate more domain reductions, one may start the iterative labeling extension process from an initial labeling which fully integrates domain reductions in the compatibility relation, so that if a target node v does not belong to the domain of a pattern node u, then the label associated with v is not compatible with the label associated with u. More formally, Zampelli et al. have defined in [20] such an initial labeling, denoted l_{dom}, as follows:

- a different unique label l_x is associated with every different (pattern or target) node $x \in N_p \cup N_t$;
- $\forall (u, v) \in N_p \times N_t$, l_u is compatible with l_v iff $v \in D_u$ and $\#adj(u) \leqslant \#adj(v)$.

They have shown that, in this case, *ILF*(k) is stronger than *LV2002* provided that $k \geqslant 2$. However, if this filtering is stronger, it is also very expensive to achieve as the complexity of *ILF*(k) highly depends on the number of different labels. Indeed, the theoretical complexity of one iteration of *ILF*(k) (*i.e.*, $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$) corresponds to the worst case where all nodes have different labels. If the number of different pattern and target labels respectively are l_p and l_t, then the complexity of one iteration of *ILF*(k) is $\mathcal{O}(e_p + l_p \cdot l_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

4. Global neighborhood constraints and LAD-filtering

We introduce a global neighborhood constraint in Section 4.1, and we describe a propagation algorithm which achieves the generalized arc consistency of this constraint in Section 4.2. We compare this consistency with other partial consistencies in Section 4.3.

4.1. Global neighborhood constraints

For each subisomorphism function $f : N_p \to N_t$ and for each pattern node $u \in N_p$, we have:

1. $\forall u' \in adj(u)$, $f(u') \in adj(f(u))$,
2. $\forall (u', u'') \in adj(u) \times adj(u)$, $u' \neq u'' \Rightarrow f(u') \neq f(u'')$.

The first property is a direct consequence of the fact that edges are preserved by subisomorphism functions whereas the second property is a direct consequence of the fact that subisomorphism functions are injections.

When considering the CSP associated with a subgraph isomorphism problem, these two properties may be expressed by the following constraint on the neighborhood of u:

$$x_u = v \quad \Rightarrow \quad \forall u' \in adj(u), \quad x_{u'} \in adj(v) \\ \wedge\, allDiff\left(\left\{x_{u'} \;\middle|\; u' \in adj(u)\right\}\right)$$

Note that the filtering algorithm *LV2002* introduced by Larrosa and Valiente in [12] actually propagates this neighborhood constraint (although it has not been explicitly introduced in [12]). However, *LV2002* only ensures a partial consistency: it basically ensures that the number of nodes adjacent to u is smaller or equal to the number of target nodes that are both adjacent to v and belong to domains of nodes adjacent to u. In Section 4.2, we describe a filtering algorithm which ensures the generalized arc consistency of neighborhood constraints.

[1] k must be greater or equal to 2 if the initial labeling from which the iterative labeling extension process is started is the empty labeling, that associates the same label to all nodes. If the initial labeling is defined by node degrees, then one iteration is enough to obtain a stronger consistency (see [20] for more details).

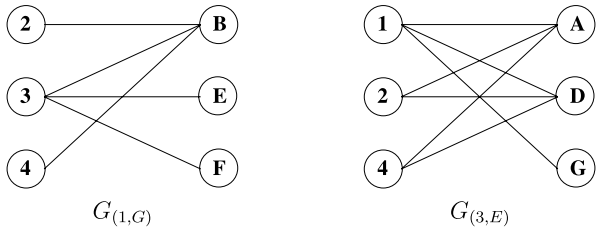

Fig. 2. Bipartite graphs associated with $(1, G)$ and $(3, E)$.

Example 5. Let us consider the subgraph isomorphism problem displayed in Fig. 1, and let us define initial domains with respect to node degrees, *i.e.*

$$D_1 = D_3 = D_5 = D_6 = \{A, B, C, D, E, F, G\}$$
$$D_2 = D_4 = \{A, B, D\}$$

The neighborhood constraint for the couple of nodes $(1, G)$ is

$$x_1 = G \quad \Rightarrow \quad x_2 \in \{B, F, E\} \wedge x_3 \in \{B, F, E\} \wedge x_4 \in \{B, F, E\} \\ \wedge \mathit{allDiff}\big(\{x_2, x_3, x_4\}\big)$$

Achieving the generalized arc consistency of this constraint allows us to remove G from D_1: if $x_1 = G$ then both x_2 and x_4 must belong to the singleton $\{B\}$ (corresponding to the intersection of their domains with $\{B, F, E\}$) so that x_2 and x_4 cannot be assigned to different values.

Note that on this example, the filtering *LV2002* cannot remove G from D_1 as $\mathcal{F}(1, G) = (D_2 \cup D_3 \cup D_4) \cap \mathit{adj}(G) = \{B, E, F\}$ so that $\#\mathcal{F}(1, G) \geqslant \#\mathit{adj}(1)$. Note also that a simple *allDiff* constraint on the set of variables $\{x_2, x_3, x_4\}$ cannot be used to remove G from D_1: one has to combine this *allDiff* constraint with the fact that, if 1 is matched to G, then 2, 3, and 4 must be matched to nodes that are adjacent to G.

4.2. A filtering algorithm for propagating global neighborhood constraints

The generalized arc consistency of a neighborhood constraint may be ensured by looking for a covering matching in a bipartite graph, as proposed by Régin in [16] for the *AllDifferent* global constraint. Let us recall that a matching of a graph $G = (N, E)$ is a subset of edges $m \subseteq E$ such that no two edges of m share a same endpoint. A matching $m \subseteq E$ covers a set of nodes N_i if every node of N_i is an endpoint of an edge of m. In this case, we shall say that m is a N_i-covering matching of G.

For every couple of nodes (u, v) such that $v \in D_u$, we define a bipartite graph that associates a node with every node adjacent to u or v and an edge with every couple (u', v') such that $v' \in D_{u'}$.

Definition 1. Given two nodes $(u, v) \in N_p \times N_t$ such that $v \in D_u$, we define the bipartite graph $G_{(u,v)} = (N_{(u,v)}, E_{(u,v)})$ such that

- $N_{(u,v)} = \mathit{adj}(u) \cup \mathit{adj}(v)$;
- $E_{(u,v)} = \{(u', v') \in \mathit{adj}(u) \times \mathit{adj}(v) \mid v' \in D_{u'}\}$.

If there does not exist a matching of the bipartite graph $G_{(u,v)}$ that covers $\mathit{adj}(u)$, then the nodes adjacent to u cannot be matched to all different nodes, and therefore v can be removed from D_u.

This filtering must be iterated. Indeed, when v is removed from D_u, the edge (u, v) is removed from other bipartite graphs so that some bipartite graphs may no longer have covering matchings. A key point for an incremental implementation of this filtering lies in the fact that the edge (u, v) only belongs to bipartite graphs $G_{(u',v')}$ such that $u' \in \mathit{adj}(u)$ and $v' \in \mathit{adj}(v) \cap D(u')$. Filtering is iterated until either a domain becomes empty — thus detecting an inconsistency — or reaching a fixpoint such that generalized arc consistency has been enforced, *i.e.*, such that for every couple (u, v) there exists a $\mathit{adj}(u)$-covering matching of $G_{(u,v)}$.

Example 6. The bipartite graph $G_{(1,G)}$ used to propagate the neighborhood constraint of Example 5 is displayed in the left part of Fig. 2. There does not exist a matching of this graph that covers $\mathit{adj}(1)$ because both 2 and 4 can only be matched to B. As a consequence, one can remove G from D_1.

The bipartite graph $G_{(3,E)}$ used to propagate the neighborhood constraint associated with the couple $(3, E)$ is displayed in the right part of Fig. 2. There exists a matching of this graph that covers $\mathit{adj}(3)$ (*e.g.*, $m = \{(1, G), (2, A), (4, D)\}$) so that E

Algorithm 1 LAD-filtering

Input: A set S of couples of pattern/target nodes to be filtered
Output: failure (if an inconsistency is detected) or success.
In case of success, domains are filtered so that $\forall u \in N_p, \forall v \in D_u$, there exists a matching of $G_{(u,v)}$ that covers $adj(u)$.

```
begin
    while S ≠ ∅ do
        Remove a couple of pattern/target nodes (u, v) from S
        if there does not exist a matching of G_(u,v) that covers adj(u) then
            Remove v from D_u
            if D_u = ∅ then return failure
            S ← S ∪ {(u', v') | u' ∈ adj(u), v' ∈ adj(v) ∩ D_u'}
        end
    end
    return success
end
```

is not removed from D_3. However, once G has been removed from D_1, the edge $(1, G)$ is removed from $G_{(3,E)}$ and there no longer exists a matching that covers $adj(3)$ (as both 1, 2, and 3 can only be matched to A and D). Hence, E is also removed from D_3.

Algorithm 1 describes the resulting filtering procedure, called LAD (Local All Different) filtering. This procedure takes in input a set S of couples of pattern/target nodes to be filtered. At the root of the search tree, this set should contain all couples of pattern/target nodes, *i.e.*, $S = \{(u, v) \mid u \in N_p, v \in D_u\}$. Then, at each choice point of the search tree, S should be initialized with the set of all couples (u, v) such that $v \in D_u$ and a node adjacent to v has been removed from the domain of a node adjacent to u since the last call to LAD-filtering.

For each couple of nodes (u, v) that belongs to the set S, LAD-filtering checks if there exists a matching of $G_{(u,v)}$ that covers $adj(u)$. If this is not the case, then v is removed from D_u, and all couples (u', v') such that u' is adjacent to u, and v' is adjacent to v and belongs to $D_{u'}$ are added to S.

The key point is to efficiently implement the procedure that checks if there exists a covering matching of $G_{(u,v)}$. Régin has shown in [16] that one can use the algorithm of Hopcroft and Karp [7] to find such a matching. The time complexity of this algorithm is $\mathcal{O}(a\sqrt{b})$ where a and b respectively are the number of edges and nodes in the bipartite graph. As the bipartite graph $G_{(u,v)}$ has $\#adj(u) + \#adj(v)$ nodes and, in the worst case (if no domain has been reduced), $\#adj(u) \cdot \#adj(v)$ edges, and as $d_t \geqslant d_p$ (otherwise the subgraph isomorphism problem instance is trivially inconsistent), the complexity of checking if there exists a covering matching of $G_{(u,v)}$ is $\mathcal{O}(d_p \cdot d_t \cdot \sqrt{d_t})$.

This complexity may be improved by exploiting the fact that the algorithm of Hopcroft and Karp is incremental: starting from an empty matching, it iteratively computes new matchings that contain more edges than the previous matching, until the matching is maximal. Each iteration basically consists in a breadth first search and is in $\mathcal{O}(d_p \cdot d_t)$ whereas the number of iterations is bounded by $2 \cdot \sqrt{d_t + d_p}$. However, if one starts the algorithm from a matching that already contains k edges, and if the maximal matching has l edges, then the number of iterations is also bounded by $l - k$ (as the size of the matching increases of at least one at each iteration).

We use this property to improve the time complexity of LAD-filtering. More precisely, for each pattern node $u \in N_p$ and each target node $v \in D_u$, we memorize the last computed matching of $G_{(u,v)}$. The space complexity of memorizing the covering matchings of all bipartite graphs is $\mathcal{O}(n_p \cdot n_t \cdot d_p)$ (there are at most $n_p \cdot n_t$ bipartite graphs, and the covering matching of $G_{(u,v)}$ is composed of $\#adj(u)$ edges). As it would be very expensive, both in time and memory, to create a copy of all covering matchings at each choice point, we simply update these covering matchings whenever this is necessary. More precisely, each time we need to check if there exists a covering matching of a bipartite graph $G_{(u,v)}$, we proceed as follows:

1. we scan the last recorded matching of $adj(u)$ and remove every couple (u', v') such that v' no longer belongs to $D(u')$;
2. if one or more couples have been removed, then we call Hopcroft Karp to complete it;
3. if Hopcroft Karp actually succeeds in completing it, then we record the computed complete matching.

Theorem 1. *The time complexity of LAD-filtering is* $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$.

Proof.

- The complexity for computing a first covering matching for all bipartite graphs is $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$; this step is performed once, at the beginning of the search process.
- Each time a value v is removed from a domain D_u, one has to update the matchings of all bipartite graphs $G_{(u',v')}$ such that $u' \in adj(u)$ and $v' \in D_{u'} \cap adj(v)$, *i.e.*, of $d_p \cdot d_t$ bipartite graphs in the worst case, and each update is done incrementally in $\mathcal{O}(d_p \cdot d_t)$.
- In the worst case, only one value is removed when updating the covering matchings of all neighbors and there are $n_p \cdot n_t$ values to remove. □

4.3. Comparison of LAD-filtering with other filterings

In this section, we compare the consistency ensured by LAD-filtering with other partial consistencies.

Theorem 2. *LAD-filtering* (*Algorithm* 1 *with S initialized to all couples* (u, v) *such that* $u \in N_p$ *and* $v \in D_u$) *ensures the Generalized Arc Consistency of neighborhood constraints, denoted* GAC(Neighborhood).

Proof. If there exists a pattern node $u \in N_p$ such that for every target node $v \in D_u$, it is not possible to assign every different pattern node $u' \in adj(u)$ to a different target node which is adjacent to v and belongs to $D_{u'}$, then LAD-filtering removes every value $v \in D_u$ (because every bipartite graph $G_{(u,v)}$ does not have a $adj(u)$-covering matching), and returns failure. Otherwise, it returns success and filters domains so that for every pattern node $u \in N_p$ and every target node $v \in D_u$, every different pattern node $u' \in adj(u)$ can be assigned to a different target node which is adjacent to v and belongs to $D_{u'}$ (as every bipartite graph $G_{(u,v)}$ has an $adj(u)$-covering matching). □

Theorem 3. GAC(Neighborhood) *is stronger than* LV2002.

Proof. *GAC*(*Neighborhood*) is at least as strong as *LV2002* because, for each pattern node $u \in N_p$ and each target node $v \in D_u$, if there exists a $adj(u)$-covering matching of $G_{(u,v)}$, then all target nodes of this covering matching belong to the set $\mathcal{F}(u, v)$ and therefore $\#\mathcal{F}(u, v) \geqslant \#adj(u)$. It is actually strictly stronger: for example, it is able to detect the inconsistency of the instance displayed in Fig. 1 whereas *LV2002* is only able to reduce the domains of the variables associated with nodes 2 and 4 to $\{A, B, D\}$ whereas the domains of the other variables are not reduced. □

Theorem 4. GAC(Neighborhood) *is as strong as* ILF(k) *when labeling extensions are started from the initial labeling* l_{dom} *and when they are iterated until reaching a fixpoint, i.e.,* $k = \infty$.

Proof. The initial labeling l_{dom} associates a unique different label with every node, and the label of a pattern node u is compatible with the label of a target node v iff $\#adj(u) \leqslant \#adj(v)$ and $v \in D_u$. With such an initial compatibility relationship, the multiset m_u that contains all labels of nodes adjacent to u is compatible with the multiset m_v that contains all labels of nodes adjacent to v iff there exists a covering matching of $G_{(u,v)}$ (as a label of m_u is compatible with a label of m_v iff there is an edge between the corresponding nodes in $G_{(u,v)}$). When a node v is removed from a domain D_u, both $ILF(\infty)$ and *LAD* check, for every couple $(u', v') \in adj(u) \times adj(v) \cap D_{u'}$, that every node adjacent to u' may still be matched to a different node adjacent to v'. In both cases, this is done in an iterative process, until a fixpoint is reached. The difference between $ILF(\infty)$ and *LAD* is that $ILF(\infty)$ recomputes all matchings, for all possible pattern/target couples, at each iteration, whereas *LAD* only updates matchings that have actually been impacted by domain reductions. Hence, *LAD* has a lower time complexity. □

Actually, $ILF(k)$ performs very poorly when it is started from the initial labeling l_{dom}. It performs much better when it is started from an initial labeling defined with respect to node degrees: with such an initial labeling, the number of different labels is usually strongly reduced and, therefore, the number of compatibility relationships to compute is also strongly reduced.

Theorem 5. GAC(Neighborhood) *is weaker than* Singleton Arc Consistency *of Edge and* AllDifferent *constraints* (*denoted* SAC(Edges + AllDiff)).

Proof. Let us first recall that singleton arc consistency ensures that we can enforce arc consistency without failure after any assignment of a value to a variable [1]. Hence, $SAC(Edges + AllDiff)$ ensures that, $\forall u \in N_p, \forall v \in D_u$, if D_u is reduced to the singleton $\{v\}$, then $AC(Edges)$ combined with $GAC(AllDiff)$ does not detect an inconsistency.

- $SAC(Edges + AllDiff)$ is at least as strong as $GAC(Neighborhood)$: when reducing a domain D_u to a singleton $\{v\}$, if $AC(Edges)$ combined with $GAC(AllDiff)$ does not detect an inconsistency, then there exists a $adj(u)$-covering matching of the bipartite graph $G_{(u,v)}$. Indeed, $AC(Edges)$ will reduce domains of nodes adjacent to u to nodes which are adjacent to v, while $GAC(AllDiff)$ will ensure that all nodes adjacent to u can be assigned to all different values.
- $SAC(Edges + AllDiff)$ is actually stronger than $GAC(Neighborhood)$ as it is able to detect the inconsistency of the subgraph isomorphism problem instance displayed in Fig. 3 whereas $GAC(Neighborhood)$ does not reduce any domain. □

However, the optimal worst-case time complexity of enforcing singleton arc consistency of a binary CSP is $\mathcal{O}(nd^3e)$ where e is the number of constraints, n is the number of variables and d is the domain size [1]. For our subgraph isomorphism CSP, if we only consider the binary edge constraints, we have $n = n_p$, $d = n_t$, and $e = e_p$ so that enforcing $SAC(Edges)$ is in $\mathcal{O}(n_p \cdot n_t^3 \cdot e_p)$. Let us consider the case of fixed-degree graphs such that $e_p = (n_p \cdot d_p)/2$. In this case, the time

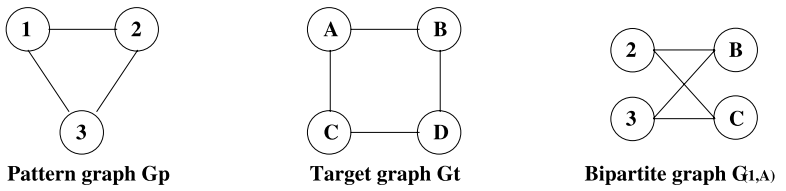

Fig. 3. Instance of subgraph isomorphism problem. Let us suppose that the initial domains are $D_1 = D_2 = D_3 = \{A, B, C, D\}$. *GAC*(*Neighborhood*) does not reduce any domain as every bipartite graph $G_{(u,v)}$ has an *adj*(u)-covering matching (see, *e.g.*, the bipartite graph $G_{(1,a)}$ displayed on the right part of the figure). However, *SAC*(*Edges* + *AllDiff*) detects an inconsistency: if D_1 is reduced to $\{A\}$, then *AC*(*Edges*) reduces D_2 and D_3 to nodes that are adjacent to A (*i.e.*, to $\{C, B\}$) and the edge $(3, 2)$ is no longer supported (as G_t has no edge between C and B) so that *AC*(*Edges*) detects an inconsistency.

Algorithm 2 LAD-filtering for directed graphs

Input: A set S of triples (u, v, x) such that $x \in \{pred, succ\}$
Output: failure (if an inconsistency is detected) or success.
In case of success, domains are filtered so that $\forall u \in N_p$, $\forall v \in D_u$, there exist a matching of $G^{pred}_{(u,v)}$ that covers $pred(u)$ and a matching of $G^{succ}_{(u,v)}$ that covers $succ(u)$.

```
begin
    while S ≠ ∅ do
        Remove a triple (u, v, x) from S
        if there does not exist a matching of G^x_(u,v) that covers x(u) then
            Remove v from D_u
            if D_u = ∅ then return failure
            S ← S ∪ {(u', v', succ) | u' ∈ succ(u), v' ∈ succ(v) ∩ D_u'} ∪ {(u', v', pred) | u' ∈ pred(u), v' ∈ pred(v) ∩ D_u'}
        end
    end
    return success
end
```

complexity of enforcing *SAC*(*Edges*) is $\mathcal{O}(n_p^2 \cdot n_t^3 \cdot d_p)$, which should be compared to the time complexity of LAD-filtering, *i.e.*, $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$. In the worst case, *i.e.*, if both G_p and G_t are complete graphs so that $d_p = n_p - 1$ and $d_t = n_t - 1$, enforcing *SAC*(*Edges*) and LAD-filtering have the same time complexity. However, for sparser graphs, LAD-filtering has a lower time complexity.

5. Extension to directed graphs

LAD-filtering may be extended to directed graphs in a rather straightforward way. In directed graphs, edges are ordered couples of nodes and, for each node u, one distinguishes the set of successor nodes $succ(u)$ that may be reached by an outgoing edge (*i.e.*, $succ(u) = \{u' \in N \mid (u, u') \in E\}$), from the set of predecessor nodes $pred(u)$ that may be reached from an ingoing edge (*i.e.*, $pred(u) = \{u' \in N \mid (u', u) \in E\}$).

To extend LAD-filtering to directed graphs, one has to associate two bipartite graphs with every couple (u, v) such that $u \in N_p$ and $v \in D_u$:

- the bipartite graph used to check that each successor of u may be matched to a different successor of v, *i.e.*, $G^{succ}_{(u,v)} = (N^{succ}_{(u,v)}, E^{succ}_{(u,v)})$ with $N^{succ}_{(u,v)} = succ(u) \cup succ(v)$ and $E^{succ}_{(u,v)} = \{(u', v') \in succ(u) \times succ(v) \mid v' \in D_{u'}\}$,
- the bipartite graph used to check that each predecessor of u may be matched to a different predecessor of v, *i.e.*, $G^{pred}_{(u,v)} = (N^{pred}_{(u,v)}, E^{pred}_{(u,v)})$ with $N^{pred}_{(u,v)} = pred(u) \cup pred(v)$ and $E^{pred}_{(u,v)} = \{(u', v') \in pred(u) \times pred(v) \mid v' \in D_{u'}\}$.

Algorithm 2 extends Algorithm 1 to directed graphs. The main difference is that it maintains a set of triples (u, v, x) such that $x \in \{pred, succ\}$ instead of a set of couples (u, v). At each iteration, a triple (u, v, x) is removed from the set, and if the graph $G^x_{(u,v)}$ does not have a covering matching, then v is removed from D_u and S is updated by adding all triples (u', v', x') such that an edge has been removed from the bipartite graph $G^{x'}_{(u',v')}$.

6. Experimental results

6.1. Test suite

We consider 1993 subgraph isomorphism instances that come from three different databases.

Scale-free database (classes sf-d-D-n and si-d-D-n) This database has been used in [20] to evaluate *ILF*(k). Graphs of these instances are scale-free networks that have been randomly generated using a power law distribution of degrees $P(d = k) = k^{-\lambda}$ with $\lambda = 2.5$ (see [20] for more details). There are 5 classes. Each of the first four classes, denoted *sf-d-D-n*, contains 20 feasible instances such that the target graph has n nodes which degrees are bounded between d and D, and the pattern graph is extracted from the target graph by randomly selecting 90% of nodes and edges from the target graph in such a

way that the pattern graph is still connected. The fifth class, denoted *si-d-D-n*, contains 20 non-feasible instances that have been generated like instances of the first four classes, excepted that 10% of new edges have been added in pattern graphs in order to obtain infeasible instances.

GraphBase database (class LV) This database has been used in [12] to evaluate *LV2002*. It contains 113 undirected graphs with different properties, *i.e.*, simple, acyclic, connected, biconnected, triconnected, bipartite and planar. We have considered the 50 first graphs. This set contains graphs ranging from 10 to 128 nodes. Using these graphs, we have generated 793 instances of the subgraph isomorphism problem by considering all couples of graphs (G_p, G_t) that are not trivially solved, *i.e.*, such that $e_p > 0$, $n_p \leqslant n_t$ and $d_p \leqslant d_t$.

Vflib database (classes bvg-n, bvgm-n, m4D-n, m4Dr-n, and r-d-n) This database has been used in [2] to evaluate Vflib, a program dedicated to graph and subgraph isomorphism problems. It contains 63 classes of instances, and each class contains instances such that the target graph has from 20 to 1000 nodes. For each class, we have only considered 4 sizes and, for each size, we have only considered the first 10 instances. We have grouped classes as follows (see [5] for more details on the original classes):

- *bvg-n* (where $n \in \{100, 200, 400, 800\}$ corresponds to the number of nodes of the target graphs);
  These classes contain fixed-valence graphs and are composed of the first 10 instances of the original classes si_x-b_y-*n* where $x \in \{2, 4, 6\}$ corresponds to the size of the pattern graph with respect to the target graph (*i.e.*, 20%, 40%, or 60%) and $y \in \{3, 6, 9\}$ corresponds to the valence. Hence, each class *bvg-n* contains 90 instances.
- *bvgm-n* (where $n \in \{100, 200, 400, 800\}$ corresponds to the number of nodes of the target graphs);
  These classes contain modified bounded-valence graphs and are composed of the first 10 instances of the original classes si_x-b_y*m-n* where $x \in \{2, 4, 6\}$ corresponds to the size of the pattern graph with respect to the target graph (*i.e.*, 20%, 40%, or 60%) and $y \in \{3, 6, 9\}$ corresponds to the valence. Hence, each class *bvgm-n* contains 90 instances.
- *m4D-n* (where $n \in \{81, 256, 526, 1296\}$ corresponds to the number of nodes of the target graphs);
  These classes contain graphs that correspond to 4D regular meshes and are composed of the first 10 instances of the original classes si_x-*m4D-n* where $x \in \{2, 4, 6\}$ corresponds to the size of the pattern graph with respect to the target graph (*i.e.*, 20%, 40%, or 60%). Hence, each class *m4D-n* contains 30 instances.
- *m4Dr-n* (where $n \in \{81, 256, 526, 1296\}$ corresponds to the number of nodes of the target graphs):
  These classes contain graphs that correspond to 4D irregular meshes and are composed of the first 10 instances of the original classes si_x-$m4Dr_r$-*n* where $x \in \{2, 4, 6\}$ corresponds to the size of the pattern graph with respect to the target graph (*i.e.*, 20%, 40%, or 60%) and $r \in \{2, 4, 6\}$ corresponds to the degree of irregularity. Hence, each class *m4Dr-n* contains 90 instances.
- *r-p-n* (where $n \in \{100, 200, 400, 600\}$ corresponds to the number of nodes and $p \in \{0.01, 0.05, 0.1\}$ corresponds to the probability of adding an edge between two nodes).
  These classes contain graphs that have been randomly generated and are composed of the first 10 instances of the original classes si_x-*rand*-r_p-*n* where $x \in \{2, 4, 6\}$ corresponds to the size of the pattern graph with respect to the target graph (*i.e.*, 20%, 40%, or 60%). Hence, each class *r-p-n* contains 30 instances.

6.2. Considered solvers

LAD *LAD*-filtering has been implemented in C and has been integrated in a complete tree search. At each node of the search tree, the next pattern node to be assigned is chosen with respect to the *minDom* heuristic, *i.e.*, we choose the non-assigned pattern node that has the smallest number of target nodes in its domain. A choice point is created for each target node that belongs to the domain of the variable to be assigned, and these different choice points are explored by increasing order of values. At each node of the search tree, *LAD*-filtering is combined with *GAC*(*AllDiff*). This search procedure is called *LAD*.

LAD is compared with *ILF*(k), with $k \in \{1, 2, 4\}$, *Abscon*(*GAC*), *Abscon*(*FC*), and *Vflib*.

ILF(k) The original implementation of *ILF*(k) was in Gecode. We consider here a new implementation in C which uses the same data structures and the same ordering heuristics as *LAD*, and which is also combined with *GAC*(*AllDiff*). This new implementation is much more efficient than the original one. For example, instances of class *sf5-8-1000* are solved in 0.19 seconds with the new implementation of *ILF*(*1*) whereas they were solved in 11.2 seconds with the old implementation.

We compare results obtained with different numbers of labeling extension iterations, *i.e.*, with $k \in \{1, 2, 4\}$. We do not report results with $k > 4$ as this never improves the solution process.

Abscon *Abscon* is a generic CSP solver written in Java by Lecoutre and Tabary (see [11] for more details). The fact that Abscon is implemented in Java whereas all other approaches are implemented in C or C++ must be taken into account since Java programs have running times several time larger than their C/C++ counterparts. We consider two variants of this solver:

- *Abscon*(*FC*) performs a forward checking propagation of the constraints, *i.e.*, *FC*(*Edges*) and *FC*(*Diff*).

Table 1
Finding all solutions: for each class, the first line gives the number of solved instances (in less than one hour on a 2.26 GHz Intel Xeon E5520), and the second line gives the CPU time (average on the completed runs).

Class	*Vflib*	*Abscon*		*ILF*			*LAD*
		FC	*AC*	$k=1$	$k=2$	$k=4$	
sf5-8-200	16	**20**	**20**	**20**	**20**	**20**	**20**
	72.45	2.04	1.75	**0.00**	0.02	0.03	0.02
sf5-8-600	0	**20**	**20**	**20**	**20**	**20**	**20**
	–	138.10	135.01	**0.07**	0.15	0.15	0.29
sf5-8-1000	0	**20**	**20**	**20**	**20**	**20**	**20**
	–	1651.11	1631.88	**0.19**	0.55	0.59	0.83
sf20-300-300	0	16	16	**20**	**20**	**20**	**20**
	–	386.87	474.11	**0.35**	5.95	8.24	2.56
si20-300-300	0	6	5	**20**	19	19	**20**
	–	823.20	1046.91	132.33	30.42	48.75	**27.77**
bvg-100	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.02	1.99	2.78	0.04	0.07	0.13	0.75
bvgm-100	89	89	**90**	**90**	**90**	**90**	**90**
	6.55	11.06	16.57	**0.48**	0.49	**0.48**	0.53
m4D-81	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	0.09	1.08	1.04	0.03	0.05	0.05	**0.02**
m4Dr-81	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	1.65	3.70	2.40	**0.18**	0.19	0.20	**0.18**
r0.01-100	21	23	28	**29**	**29**	**29**	**29**
	83.60	121.98	322.67	**158.35**	170.63	170.53	180.24
r0.05-100	2	22	**23**	**23**	22	22	**23**
	513.01	28.60	56.78	135.81	107.18	108.99	**19.73**
r0.1-100	0	25	28	28	28	28	**29**
	–	64.91	218.38	217.17	242.00	243.12	**148.38**
All instances	338	451	460	480	478	478	**481**
	13.83	118.72	144.86	34.41	31.09	32.07	**22.34**

- *Abscon*(*AC*) maintains Arc Consistency of edge constraints. For the difference constraints, it maintains a consistency that is stronger than *AC(Diff)* but weaker than *GAC*(*AllDiff*). It also uses symmetry breaking techniques.

Both variants consider the *minDom* ordering heuristic for choosing the next variable to assign.

Vflib *Vflib* [2,3] is a solver dedicated to graph and subgraph isomorphism problems, and it is considered as the state-of-the-art for subgraph isomorphism. It basically performs a forward checking propagation of edge and difference constraints, but this propagation is limited to nodes that are adjacent to already matched nodes for difference constraints. It uses specific variable and value ordering heuristics: variable and values are chosen so that the subgraph induced by the matched nodes is connected (except when the pattern or the target graphs are composed of different connected components).

6.3. Experimental comparison on the problem of finding all solutions

Let us first consider the problem of finding all solutions to an instance, thus allowing a comparison that is less dependent on value ordering heuristics. For this first experiment, we have discarded instances that have too many solutions. Hence, we have only considered classes from the *scale-free* database, and the smallest classes of the *vflib* database (such that the target graph has 100 or 81 nodes).

Table 1 displays, for each class and each considered approach, the number of instances for which all solutions have been found in less than one hour on a 2.26 GHz Intel Xeon E5520, and the average corresponding CPU time. On these classes, *LAD* has solved 1 (resp. 3, 3, 23, 29, and 143) more instances than *ILF*(*1*) (resp. *ILF*(*2*), *ILF*(*4*), *Abscon*(*AC*), *Abscon*(*FC*), and *Vflib*). When comparing CPU times, we note that *LAD* is slower than the three variants of *ILF* on classes *sf-5-8-** and *bvg*, but these instances are easy ones and *LAD* solves them in less than one second. However, on harder classes such as *si20-300-300*, *r0.05-100*, and *r0.1-100*, *LAD* is significantly quicker than *ILF*. On all classes, *LAD* and *ILF* are an order quicker than *Abscon*. *Vflib* is competitive on classes *bvg-100*, *m4D-81*, and *m4Dr-81*, but it is not competitive at all on all other classes.

Table 2 displays the average number of fail nodes (*i.e.*, the number of times an inconsistency is detected), for each class and each approach except *Vflib* (because this information is not available in *Vflib*). On some classes, such as *sf5-8-**, *LAD* and *ILF* have comparable numbers of failed nodes, and this corresponds to the classes that are more quickly solved by *ILF* than by *LAD*. However, on some other instances, such as *r*-100*, *LAD* explores many fewer nodes than *ILF*. The number of fail

Table 2
Number of fail nodes (average on the completed runs); numbers in brackets after class names give the average number of solutions of the instances of the class.

Class (#solutions)	*Abscon*		*ILF*			*LAD*
	FC	*AC*	$k=1$	$k=2$	$k=4$	
sf5-8-200 (1.10)	3076	108	5	0	0	0
sf5-8-600 (1.00)	418	418	4	0	0	0
sf5-8-1000 (1.05)	557	557	7	0	0	0
sf20-300-300 (4.45)	397,844	29,338	38	13	7	0
si20-300-300 (0.00)	913,730	61,191	15,342	62	22	27
bvg-100 (218)	10,037	2862	461	391	391	0
bvgm-100 (145,855)	8977	95,618	641	379	222	1
m4D-81 (1253)	1904	327	701	669	652	23
m4Dr-81 (30,642)	22,920	23,592	1356	1304	1300	12
r0.01-100 (57,291,325)	38,853	6,749,220	10,621	6717	6175	60
r0.05-100 (6,062,230)	233,044	615,882	2,857,279	539,522	539,167	5243
r0.1-100 (30,501,838)	819,714	1,985,225	2,227,792	2,224,579	2,224,408	320,067

Table 3
Finding the first solution of instances of the *LV* class: #solved is the number of solved instances (in less than one hour on a 2.26 GHz Intel Xeon E5520), Time and #fail respectively give the CPU time and the number of fail nodes (average on the completed runs).

	Vflib	*Abscon*		*ILF*			*LAD*
		FC	*AC*	$k=1$	$k=2$	$k=4$	
#solved	468	647	662	698	699	699	**728**
Time	73.72	72.51	54.25	30.85	31.12	30.77	**14.57**
#fail	–	1,202,372	324,075	297,107	182,588	159,493	13,560

nodes of both *ILF* and *LAD* is an order smaller than *Abscon*. On some classes, *Abscon*(*AC*) has more fail nodes than *Abscon*(*FC*), but this corresponds to the fact that *Abscon*(*AC*) solves more instances than *Abscon*(*FC*) and, for these harder instances, the number of fail nodes is significantly higher than for the instances that are solved by both approaches.

6.4. Experimental comparison on the problem of finding the first solution

To illustrate scale-up properties of the different approaches and compare them on a larger set of instances, we now consider the problem of finding the first solution (or proving inconsistency). For this comparison, we consider instances of the *LV* class and the larger classes of the *vflib* database (such that the target graph has more than 100 nodes).

Table 3 displays the number of solved instances for the *LV* class, which contains 793 instances with many different features (graphs have different properties and sizes; some instances are feasible and have many solutions, some others are inconsistent). For this class, *LAD* has solved 29 (resp. 29, 30, 66, 81, and 260) more instances than *ILF*(*4*) (resp. *ILF*(*2*), *ILF*(*1*), *Abscon*(*AC*), *Abscon*(*FC*), and *Vflib*). This table also shows us that *Abscon*(*AC*) and *ILF*(*1*) have comparable number of fail nodes, and nearly four times as less as *Abscon*(*FC*). *ILF*(*2*) and *ILF*(*4*) have smaller number of fail nodes but the reduction of the search space is not enough to allow *ILF*(*2*) and *ILF*(*4*) to become competitive. The number of fail nodes of *LAD* is much smaller (more than 20 times as small as *Abscon*(*AC*) and *ILF*(*1*)).

Tables 4 and 5 allow us to compare scale-up properties of the different considered approaches. Table 4 displays results on rather easy classes of the *Vflib* database. *LAD* is able to solve the 900 instances of these classes in less than 1.5 seconds on average, and it is nearly 4 times as fast as *ILF*(k). It is also significantly faster than *Abscon*. Interestingly, *Vflib* is very efficient and exhibits very good scale-up properties on some classes such as *bvg-**, *bvgm-**, and *m4Dr-**. Actually, *Vflib* uses variable and value ordering heuristics that are not used by the other approaches: at each iteration, it chooses the next couple (u, v) of nodes to match so that both u and v are adjacent to some nodes that have already been matched (whenever this is possible). These ordering heuristics may explain the very good behavior of *Vflib* on some instances when the goal is to find only one solution. It may also explain the fact that it is able to solve 29 instances of the *m4D-256* class in less than 0.01 second, whereas it is not able to solve the last instance of this class in one hour.

However, Table 5 shows us that the different approaches exhibit different scale-up properties on the random classes *r-p-n*. Indeed, when the probability p of adding an edge is 0.01, *LAD* is better than *Abscon* which is better than *ILF*, whereas when this probability increases, *Abscon* is better than *ILF* which is better than *LAD*. Actually, the denser and the larger the graphs, and the worse *LAD*. This may come from the fact that the complexity of LAD-filtering is $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$: the degree is 10 times bigger (on average) for the graphs of classes *r0.1-** than for those of classes *r0.01-**. Therefore, when graphs are rather sparse, it is worth filtering with *LAD* whereas when graphs are denser, one has better consider a simpler filtering procedure such as *AC*(*Edges*).

Table 4
Finding the first solution of easy instances of the *vflib* base: for each class, the first line displays the number of solved instances (in less than one hour on a 2.26 GHz Intel Xeon E5520) and the second line the CPU time (average on completed runs).

Class	*Vflib*	*Abscon*		*ILF*			*LAD*
		FC	*AC*	$k=1$	$k=2$	$k=4$	
bvg-200	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.00	0.68	0.78	**0.00**	**0.00**	**0.00**	0.14
bvg-400	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.00	2.85	2.99	0.01	0.01	0.01	1.06
bvg-800	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.02	54.13	54.86	0.03	0.04	0.05	8.41
bvgm-200	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.00	0.95	0.73	**0.00**	**0.00**	**0.00**	0.01
bvgm-400	**90**	89	89	**90**	**90**	**90**	**90**
	0.01	3.20	1.66	1.55	**0.01**	**0.01**	0.04
bvgm-800	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.03	12.02	12.07	0.06	0.04	**0.03**	0.19
m4D-256	29	**30**	**30**	**30**	**30**	**30**	**30**
	0.00	2.88	1.73	**0.01**	**0.01**	**0.01**	0.04
m4D-526	23	**30**	**30**	29	**30**	29	**30**
	4.11	159.76	164.90	9.61	32.93	30.72	**1.71**
m4D-1296	20	23	23	29	29	29	**30**
	0.05	252.73	242.47	52.93	61.21	73.33	**5.67**
m4Dr-256	**90**	**90**	**90**	**90**	**90**	**90**	**90**
	0.00	7.91	1.44	0.23	1.05	2.24	0.06
m4Dr-526	**90**	89	89	89	89	89	**90**
	0.01	23.47	23.35	14.02	18.31	19.08	0.33
m4Dr-1296	**90**	89	89	**90**	**90**	**90**	**90**
	0.06	193.27	188.44	6.41	5.46	5.43	1.63
All instances	882	890	890	897	898	897	**900**
	0.12	41.95	40.60	4.25	5.55	6.04	**1.43**

Table 5
Finding the first solution of hard instances of the *Vflib* base: for each class, the first line displays the number of solved instances (in less than one hour on a 2.26 GHz Intel Xeon E5520) and the second line the CPU time (average on completed runs).

Class	*Vflib*	*Abscon*		*ILF*			*LAD*
		FC	*AC*	$k=1$	$k=2$	$k=4$	
r0.01-200	3	28	**30**	28	28	28	**30**
	1735.93	192.14	0.99	27.48	44.09	46.49	**0.04**
r0.01-400	0	20	29	14	14	14	**30**
	–	33.14	69.68	175.83	228.78	214.85	**45.58**
r0.01-600	0	23	23	12	9	7	**29**
	–	226.38	236.14	428.14	1069.96	806.96	**113.51**
r0.05-200	0	25	28	28	28	28	**30**
	–	266.77	142.80	125.57	198.68	198.66	**38.28**
r0.05-400	0	22	24	**25**	**25**	**25**	17
	–	632.84	647.48	519.04	500.54	**494.12**	1190.88
r0.05-600	0	**14**	**14**	13	5	5	1
	–	**1915.65**	1936.98	1505.51	2319.68	2304.85	2100.61
r0.1-200	0	27	**29**	26	26	26	21
	–	143.36	**309.54**	320.52	357.70	351.10	646.31
r0.1-400	0	**6**	**6**	5	5	5	1
	–	**1764.63**	1972.18	1950.67	1917.68	2070.81	961.35
r0.1-600	0	0	0	0	0	0	0
	–	–	–	–	–	–	–
All instances	3	165	**183**	151	140	138	159
	1735.93	443.14	**409.55**	414.03	447.36	426.67	268.48

7. Conclusion

We have introduced a new filtering algorithm for subgraph isomorphism that basically ensures that all nodes adjacent to a same pattern node may be matched to nodes that are all different and that are all adjacent to a same target node. This filtering is stronger than *LV2002*. Actually, it achieves the same consistency as the strongest variant of *ILF*(k), *i.e.*, when the initial labeling fully integrates domain reductions and when labeling extensions are iterated until reaching a fixpoint. However, this consistency is achieved at a lower cost by updating matchings incrementally instead of recomputing them from scratch at each iteration, and by updating only the matchings that are impacted by a domain reduction instead of recomputing all matchings.

We have experimentally shown on a wide benchmark of 2000 or so instances that this new filtering is able to solve more instances quicker, and that it drastically reduces the search space so that many instances are solved without backtracking. However, this filtering is outperformed by arc consistency on the densest random graphs, such that edge density is greater or equal to 10%.

This filtering procedure could be easily integrated within a constraint programming language. In particular, we plan to integrate it in our constraint-based graph matching system [9] that is built on top of *Comet* [8].

We also plan to improve LAD-filtering by studying different strategies for choosing, at each iteration, the next couple (u, v) that is removed from S. In the results reported in this paper, we have considered a basic *last in first out* strategy as S is implemented with a stack. However, we could use a priority queue that orders couples with respect to the number of edges that have been removed from the corresponding bipartite graph.

Further work will also concern the extension of this filtering procedure to the maximum common subgraph problem, which involves finding the largest graph that is subisomorphic to two given graphs. Indeed, the algorithm of Hopcroft and Karp may be used to compute the maximal matching of bipartite graphs $G_{(u,v)}$, thus giving a bound on the largest number of edges that may be matched when u is matched to v.

Acknowledgements

Many thanks to Yves Deville for enriching discussions, to Jean-Christophe Luquet for having implemented a first version of ILF, to Christophe Lecoutre for guiding me in the use of Abscon, and to anonymous reviewers for their valuable remarks who helped me improving this paper. This work was done in the context of project SATTIC (ANR grant BLANC07-1_184534).

References

[1] Christian Bessiere, Romuald Debruyne, Theoretical analysis of singleton arc consistency and its extensions, Artif. Intell. 172 (1) (2008) 29–41.
[2] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Performance evaluation of the vf graph matching algorithm, in: ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing, IEEE Computer Society, Washington, DC, USA, 1999, p. 1172.
[3] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento, An improved algorithm for matching large graphs, in: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, 2001, pp. 149–159.
[4] Donatello Conte, Pasquale Foggia, Carlo Sansone, Mario Vento, Thirty years of graph matching in pattern recognition, IJPRAI 18 (3) (2004) 265–298.
[5] Pasquale Foggia, Carlo Sansone, Mario Vento, A database of graphs for isomorphism and sub-graph isomorphism benchmarking, in: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, 2001.
[6] M. Garey, D. Johnson, Computers and Intractability, Freeman and Co., New York, 1979.
[7] John E. Hopcroft, Richard M. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM J. Comput. 2 (4) (1973) 225–231.
[8] P. Van Hentenryck, L. Michel, Constraint-Based Local Search, The MIT Press, 2005.
[9] V. le Clément, Y. Deville, C. Solnon, Constraint-based graph matching, in: 15th Conference on Principles and Practice of Constraint Programming (CP), in: LNCS, vol. 5732, Springer, 2009, pp. 274–288.
[10] C. Lecoutre, Constraint Networks: Techniques and Algorithms, ISTE/Wiley, 2009.
[11] C. Lecoutre, S. Tabary, Abscon 112: towards more robustness, in: 3rd International Constraint Solver Competition (CSC'08), 2008, pp. 41–48.
[12] Javier Larrosa, Gabriel Valiente, Constraint satisfaction algorithms for graph pattern matching, Mathematical Structures in Comp. Sci. 12 (4) (2002) 403–422.
[13] J.J. McGregor, Relational consistency algorithms and their application in finding subgraph and graph isomorphisms, Inf. Sci. 19 (3) (1979) 229–250.
[14] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artif. Intell. 28 (1986) 225–233.
[15] Jean-Charles Régin, Développement d'outils algorithmiques pour l'intelligence artificielle. Application à la chimie organique, PhD thesis, 1995.
[16] J.-C. Régin, A filtering algorithm for constraints of difference in CSPs, in: Proc. 12th Conf. American Assoc. Artificial Intelligence, vol. 1, Amer. Assoc. Artificial Intelligence, 1994, pp. 362–367.
[17] F. Rossi, P. van Beek, T. Walsh, Handbook of Constraint Programming (Foundations of Artificial Intelligence), Elsevier Science Inc., New York, NY, USA, 2006.
[18] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (January 1976) 31–42.
[19] David L. Waltz, Generating semantic descriptions from drawings of scenes with shadows, Technical Report AI-TR-271, MIT Artificial Intelligence Laboratory, 1972.
[20] S. Zampelli, Y. Deville, C. Solnon, Solving subgraph isomorphism problems with constraint programming, Constraints (2010), in press, doi:10.1007/s10601-009-9074-3.