Constraint-Based Graph Matching

Vianney le Clément[1], Yves Deville[1], Christine Solnon[2,3]

[1] Université catholique de Louvain, Department of Computing Science and Engineering, Place Sainte-Barbe 2, 1348 Louvain-la-Neuve (Belgium)
vianney.leclement@student.uclouvain.be
Yves.Deville@uclouvain.be
[2] Université de Lyon
[3] Université Lyon 1, LIRIS, CNRS UMR5205, 69622 Villeurbanne Cedex (France)
christine.solnon@liris.cnrs.fr

Abstract. Measuring graph similarity is a key issue in many applications. We propose a new constraint-based modeling language for defining graph similarity measures by means of constraints. It covers measures based on univalent matchings, such that each node is matched with at most one node, as well as multivalent matchings, such that a node may be matched with a set of nodes. This language is designed on top of Comet, a programming language supporting both Constraint Programming (CP) and Constraint-Based Local Search (CBLS). Starting from the constraint-based description of the measure, we automatically generate a Comet program for computing the measure. Depending on the measure characteristics, this program either uses CP —which is better suited for computing exact measures such as (sub)graph isomorphism— or CBLS —which is better suited for computing error-tolerant measures such as graph edit distances. First experimental results show the feasibility of our approach.

1 Introduction

In many applications graphs are used to model structured objects such as, e.g., images, design objects, molecules or proteins. In these applications, measuring graph similarity is a key issue for classification, pattern recognition or information retrieval. Measuring the similarity of two graphs involves finding a best matching between their nodes. Hence, graph similarity measures are closely related to graph matching problems. There exist many different kinds of graph matchings, ranging from exact matchings such as (sub)graph isomorphism to error-tolerant matchings such as (extended) edit distances.

Exact matchings may be solved by complete approaches such as, e.g., Nauty [1] for graph isomorphism and Vflib [2] for subgraph isomorphism. These approaches exploit invariant properties such as node degrees to prune the search space and they are rather efficient on this kind of problems. However, in many real world applications one looks for (sub)graph isomorphisms which satisfy additional constraints such as, e.g., label compatibility between the matched nodes. Dedicated approaches such as Vflib can only handle equality constraints between matched labels; other constraints cannot be used during the search process to reduce the search space.

Error-tolerant matchings involve finding a best matching, that optimizes some given objective function which evaluates the similarity induced by the matching. They are

usually solved by numerical methods [3,4,5], or by heuristic approaches which explore the search space of all possible matchings in an incomplete way, using heuristics to guide the search such as, e.g., genetic algorithms [6], greedy algorithms [7], reactive tabu search [8], and Ant Colony Optimization [9]. These algorithms do not guarantee to find optimal solutions; as a counterpart, they usually find rather good solutions within reasonable CPU times. Algorithms dedicated to error-tolerant matchings may be used to solve exact matching problems by defining appropriate edit costs. However, they may be less efficient than dedicated approaches such as Nauty or Vflib.

Contribution. Graph matching problems may be defined by means of constraints on the cardinality of the matching, and on edges and labels. Hence, we introduce a modeling language for defining graph matchings by means of constraints. This language allows one to define new graph matching problems in a declarative way, by a simple enumeration of constraints. It covers both exact and error-tolerant matchings. We show that this language can be used to define existing matching thus giving a uniform framework to these different matching problems.

Graph matching problems defined by means of constraints are solved by constraint solvers that are embedded into the language. We more particularly consider two different kinds of solving approaches: a complete approach, which combines a tree search with filtering techniques, and an incomplete approach, based on local search. Our system is designed on top of Comet, a constraint-based modeling language supporting both tree search and local search. Starting from the constraint-based description of the matching, we automatically generate a Comet program for computing it. Depending on the constraints, this program either uses tree search or local search, thus choosing the most efficient approach for solving the considered matching problem. A similar approach is taken in [10] for solving scheduling problems.

Outline of the Paper. Section 2 gives an overview of existing graph matching problems. Section 3 briefly describes the constraint programming paradigm on which our approach is based. Section 4 introduces a set of constraints that may be used to define graph matching problems. Section 5 shows how to use these constraints to define existing graph matching problems. Section 6 discusses implementation issues and Section 7 gives some preliminary experimental results.

2 Graph Matching Problems

We consider labeled directed graphs defined by $G = (N, E, L)$ such that N is a set of nodes, $E \subseteq N \times N$ is a set of directed edges, and $L : N \cup E \to \mathbb{N}$ is a function that associates labels to nodes and edges. Throughout this paper, we assume that the labeled graphs to be matched are $G_1 = (N_1, E_1, L_1)$ and $G_2 = (N_2, E_2, L_2)$ such that $N_1 \cap N_2 = \varnothing$.

Functional Matchings. A (total) functional matching between G_1 and G_2 is a function $f : N_1 \to N_2$ which matches each node of G_1 with a node of G_2. When the matching preserves all edges of G_1, i.e., $\forall (u_1, v_1) \in E_1, (f(u_1), f(v_1)) \in E_2$, it is called a *graph homomorphism* [11].

In many cases, f is injective so that each node of G_2 is matched to at most one node of G_1, i.e., $\forall u_1, v_1 \in N_1, u_1 \neq v_1 \Rightarrow f(u_1) \neq f(v_1)$. In this case the matching is said to be *univalent*. Particular cases are *subgraph isomorphism*, when f is a homomorphism,

and *graph isomorphism*, when f is bijective and f^{-1} is also a homomorphism. Note that while subgraph isomorphism is NP-complete, graph isomorphism is not known to be nor NP-complete nor in P.

These different problems can be extended to the case where f is a partial function such that some nodes of N_1 are not matched to a node of N_2. In particular, the *maximum common subgraph* corresponds to the partial injective matching which preserves edges and maximizes the number of matched nodes or edges.

These different problems can also be extended to take into account node and edge labels, thus leading to the *graph edit distance* [12] or the *graph matching problem* of [13]. For example, the graph edit distance involves finding a partial injective matching which minimizes the sum of deletion (resp. addition) costs associated with the labels of the nodes and edges of G_1 (resp. G_2) that are not matched, and substitution costs associated with labels of nodes and edges that are matched but that have different labels.

Relational Matchings. Many real-world applications involve comparing objects described at different granularity levels and, therefore, require *multivalent* matchings, such that each node may be matched with a (possibly empty) set of nodes. In particular, in the field of image analysis, some images may be over-segmented whereas some others may be under-segmented so that several regions of one image correspond to a single region of another image. In this case, the matching is no longer a function, but becomes a *relation* $M \subseteq N_1 \times N_2$, and the goal is to find the best matching, i.e., the matching which maximizes node, edge and label matchings while minimizing the number of split nodes (that are matched with more than one node). Graph similarity measures based on multivalent matchings have been proposed, e.g., in [7,14].

All these problems, ranging from maximum common subgraph to similarity measures based on multivalent matchings, are NP-hard.

3 Constraint-Based Modeling

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 [15].

Many embedded constraint solvers are based on a complete tree search combined with filtering techniques which reduce the search space. We have proposed in [16] and [17] filtering algorithms that are respectively dedicated to graph and subgraph isomorphism problems. These filtering algorithms exploit the global structure of the graphs to drastically reduce the search space. We have experimentally shown that they allow CP to be competitive, and in some cases to outperform, dedicated approaches such as Nauty or Vflib.

Embedded constraint solvers may also be based on local search. In this case, the search space is explored in an incomplete way by iteratively performing local modifications, using some metaheuristics such as tabu search or simulated annealing to escape from locally optimal solutions. We have introduced in [8] a reactive tabu search approach for solving multivalent graph matching problems.

Comet [18] is a constraint-based modeling language which supports both complete tree search and local search. A Comet program is composed of two parts: (1) a high-

level model describing the problem by means of constraints, constraint combinators, and objective functions; (2) a search procedure expressed at a high abstraction level.

Our system for modeling and solving graph matching problems is designed on top of Comet. An example of program for modeling and solving a subgraph isomorphism problem is given in Fig. 1.

```
1  include "matching";
2
3  bool[,] adj1 = ...
4  bool[,] adj2 = ...
5  SimpleGraph<Mod> g1(adj1);
6  SimpleGraph<Mod> g2(adj2);
7
8  Matching<Mod> m(g1,g2);
9  m.post(cardMatch(g1.getAllNodes(), 1, 1));
10 m.post(injective(g1.getAllNodes()));
11 m.post(matchedToSomeEdges(g1.getAllEdges()));
12 m.close();
13
14 DefaultGMSynthesizer synth();
15 GMSolution<Mod> sol = synth.solveMatching(m);
16 print(sol);
```

Fig. 1. Example of $\mathcal{SI}$ matching problem solved with our Comet prototype

As for every Comet program, this program consists in two parts. In the first part (lines 9–13), the problem is modeled by means of high-level constraints. The first two constraints specify the cardinality of the matching to search, which must match each node of G_1 with exactly one node of G_2 (line 10), and which must be injective (line 11). The last constraint specifies that the matching must preserve edges (line 12). Note that problem-dependent constraints may be very easily added. We introduce in section 4 the different constraints that may be used to model graph matching problems, and we show in section 5 how to use these constraints to define existing graph matching problems.

In the second part (lines 15–16), a synthesizer is called to automatically generate a Comet program for computing the solution. Depending on the constraints, this program either uses tree search or local search, thus choosing the most appropriate approach. This synthesizer is described in Section 6.

4 Constraints for Modeling Graph Matching Problems

A graph matching problem between two directed graphs $G_1 = (N_1, E_1, L_1)$ and $G_2 = (N_2, E_2, L_2)$ involves finding a matching $M \subseteq N_1 \times N_2$ which satisfies some given constraints. These constraints actually specify the considered matching problem. In this section, we introduce different constraints that may be used to define graph matching

Let $M \subseteq N_1 \times N_2, u, v \in N_1 \cup N_2, U \subseteq N_1 \cup N_2, ub, lb \in \mathbb{N}, L = L_1 \cup L_2$,
$D \subseteq (N_1 \cup N_2) \times (N_1 \cup N_2)$.

$$
\begin{array}{rl}
\text{MinMatch}(M,u,lb) & \equiv lb \leq \#M(u) \\
\text{MinMatch}(M,U,lb) & \equiv \forall u \in U : \text{MinMatch}(M,u,lb) \\
\text{MaxMatch}(M,u,ub) & \equiv \#M(u) \leq ub \\
\text{MaxMatch}(M,U,ub) & \equiv \forall u \in U : \text{MaxMatch}(M,u,ub) \\
\text{CardMatch}(M,u,lb,ub) & \equiv \text{MinMatch}(M,u,lb) \wedge \text{MaxMatch}(M,u,ub) \\
\text{CardMatch}(M,U,lb,ub) & \equiv \forall u \in U : \text{CardMatch}(M,u,lb,ub) \\
\hline
\text{Injective}(M,U) & \equiv \forall u,v \in U, u \neq v : M(u) \cap M(v) = \varnothing \\
\hline
\text{MatchedToSomeEdges}(M,u,v) & \equiv \exists u' \in M(u), \exists v' \in M(v) : (u',v') \in E_1 \cup E_2 \\
\text{MatchedToSomeEdges}(M,D) & \equiv \forall (u,v) \in D : \text{MatchedToSomeEdges}(M,u,v) \\
\text{MatchedToAllEdges}(M,u,v) & \equiv \forall u' \in M(u), \forall v' \in M(v) : (u',v') \in E_1 \cup E_2 \\
\text{MatchedToAllEdges}(M,D) & \equiv \forall (u,v) \in D : \text{MatchedToAllEdges}(M,u,v) \\
\hline
\text{MatchSomeNodeLabels}(M,u) & \equiv \exists v \in M(u) : L(u) = L(v) \\
\text{MatchSomeEdgeLabels}(M,u,v) & \equiv \exists u' \in M(u), \exists v' \in M(v) : (u',v') \in E_1 \cup E_2 \\
 & \quad \wedge L(u,v) = L(u',v') \\
\text{MatchAllNodeLabels}(M,u) & \equiv \forall v \in M(u) : L(u) = L(v) \\
\text{MatchAllEdgeLabels}(M,u,v) & \equiv \forall u' \in M(u), \forall v' \in M(v) : (u',v') \in E_1 \cup E_2 \\
 & \quad \wedge L(u,v) = L(u',v')
\end{array}
$$

Fig. 2. Basic constraints for modeling graph matching problems.

problems. To make the following easier to read, we denote by $M(u)$ the set of nodes that are matched to a node u by M, i.e., $\forall u \in N_1, M(u) = \{v \in N_2 | (u,v) \in M\}$ and $\forall u \in N_2, M(u) = \{v \in N_1 | (v,u) \in M\}$.

Fig. 2 describes the basic constraints for modeling graph matching problems.

The first set of constraints enables to specify the minimum and maximum number of nodes a node is matched to. For ease of use, these constraints are also defined for a set U of nodes, to constrain the number of nodes matched to every node in U.

The second set of constraints enables to state that a set U of nodes is *injective*, i.e., that the nodes of this set are matched to different nodes. There is a simple relationship between the maximum numbers of matched nodes and injective nodes when U is equal to N_1 (resp. N_2): in this case, every node of N_2 (resp. N_1) must be matched to at most one node of N_1 (resp. N_2), i.e.,

$$
\begin{aligned}
&\text{Injective}(M,N_1) \Leftrightarrow \text{MaxMatch}(M,N_2,1) \\
&\text{Injective}(M,N_2) \Leftrightarrow \text{MaxMatch}(M,N_1,1)\ .
\end{aligned}
$$

The third set of constraints allows one to specify that a couple of nodes must be matched to a couple of nodes related by an edge. MatchedToSomeEdges ensures that there exists at least one couple of matched nodes which is related by an edge whereas MatchedToAllEdges ensures that all couples of matched nodes are related by edges. Note that, when $M(u) = \varnothing$ or $M(v) = \varnothing$, the MatchedToSomeEdges(M, u,v) constraint is violated whereas MatchedToAllEdges(M,u,v) is satisfied. Note also that when $\#M(u) = \#M(v) = 1$, the two constraints MatchedToSomeEdges(M,u,v) and MatchedToAllEdges(M,u,v) are equivalent. Note finally that these constraints are meaningful only when u and v belong to the same graph. For ease of use, these con-

straints are also defined for a set D of couples of nodes to constrain every couple of D to be matched to edges.

The last set of constraints enables to specify that labels of matched nodes or edges must be equal. On the one hand MatchSomeNodeLabels (resp. MatchSomeEdgeLabels) ensures that there is at least one matched node (resp. edge) with the same label. On the other hand MatchAllNodeLabels (resp. MatchAllEdgeLabels) ensures that all matched nodes (resp. edges) have the same label.

All these constraints may either be posted as hard ones, so that they cannot be violated, or as soft ones, so that they may be violated at some given cost. Soft constraints are posted by using a specific method which has two arguments: the constraint and the cost associated with its violation.

5 Modeling Graph Matching Problems by Means of Constraints

We now show how to model classical graph matching problems with the constraints introduced in the previous section. Note that different (equivalent) formulations of these problems are possible.

Exact matching problems are modeled with hard constraints. For graph homomorphism ($\mathcal{GH}$), the matching must be a total function which preserves edges, i.e.,
$\mathcal{GH}(M,G_1,G_2) \equiv \text{CardMatch}(M,N_1,1,1) \wedge \text{MatchedToSomeEdges}(M,E_1)$.

For subgraph isomorphism ($\mathcal{SI}$), we add an injective constraint to $\mathcal{GH}$, i.e.,
$\mathcal{SI}(M,G_1,G_2) \equiv \mathcal{GH}(M,G_1,G_2) \wedge \text{Injective}(M,N_1)$.

For induced subgraph isomorphism ($\mathcal{ISI}$), we add a MatchedToAllEdges constraint to $\mathcal{SI}$ in order to ensure that when the two nodes of an edge of G_2 are matched to nodes of G_1, these matched nodes are related by an edge in G_1, i.e.,
$\mathcal{ISI}(M,G_1,G_2) \equiv \mathcal{SI}(M,G_1,G_2) \wedge \text{MatchedToAllEdges}(M,E_2)$.

For graph isomorphism ($\mathcal{GI}$), we check that the matching is a bijective total function which preserves edges, i.e.,
$\mathcal{GI}(M,G_1,G_2) \equiv \text{CardMatch}(M,N_1 \cup N_2,1,1) \wedge \text{MatchedToSomeEdges}(M,E_1 \cup E_2)$.

Constraints can also be used for modeling approximate matching problems such as the maximum common subgraph ($\mathcal{MCS}$). In this case, one has to combine hard constraints (for ensuring that the matching is a partial function) with soft constraints (for maximizing the number of edges of G_1 which are matched), i.e.,

$$\begin{aligned}\mathcal{MCS}(M,G_1,G_2) \equiv\ & \text{MaxMatch}(M,N_1 \cup N_2,1) \\ & \wedge\ \forall (u,v) \in E_1, \text{soft}(\text{MatchedToSomeEdges}(M,u,v),1)\ .\end{aligned}$$

By defining the cost of violation of each MatchedToSomeEdges soft constraint to 1, we ensure that the optimal solution will have a cost equal to the number of edges of G_1 which are not in the common subgraph. Hence, the number of edges in the common subgraph is equal to the number of edges of G_1 minus the cost of the optimal solution.

For maximum common induced subgraph ($\mathcal{MCIS}$), one has to replace the soft MatchedToSomeEdges constraint by a hard MatchedToAllEdges constraint as edges between matched nodes must be preserved. To maximize the number of nodes that are

matched, we add a soft MinMatch constraint. More precisely,

$$\begin{aligned}\mathcal{MCIS}(M,G_1,G_2) \equiv\ & \text{MaxMatch}(M,N_1\cup N_2,1)\\ & \wedge\ \text{MatchedToAllEdges}(M,E_1\cup E_2)\\ & \wedge\ \forall u\in N_1,\text{soft}(\text{MinMatch}(M,u,1),1)\ .\end{aligned}$$

By defining the cost of violation of each MinMatch soft constraint to 1, we ensure that the optimal solution will have a cost equal to the number of nodes of G_1 which are not in the common subgraph. Hence, the number of nodes in the common subgraph is equal to the number of nodes of G_1 minus the cost of the optimal solution.

The graph edit distance ($\mathcal{GED}$) generalizes $\mathcal{MCS}$ by taking into account edge and node labels. This distance is computed with respect to some edit costs which are given by the user. Let us note $c_d(l)$ (resp. $c_a(l)$) the edit cost associated with the deletion (resp. addition) of the label l, and $c_s(l_1,l_2)$ the edit cost associated with the substitution of label l_1 by label l_2. The graph edit distance may be defined by $\mathcal{GED}(M,G_1,G_2) \equiv$

$$\begin{aligned}&\text{MaxMatch}(M,N_1\cup N_2,1)\\ &\wedge\ \forall u\in N_1,\text{soft}(\text{MinMatch}(M,u,1),c_d(L_1(u)))\\ &\wedge\ \forall u\in N_2,\text{soft}(\text{MinMatch}(M,u,1),c_a(L_2(u)))\\ &\wedge\ \forall u\in N_1,\text{soft}(\text{MatchAllNodeLabels}(M,u),c_s(L_1(u),L_2(M(u))))\\ &\wedge\ \forall (u,v)\in E_1,\text{soft}(\text{MatchedToSomeEdges}(M,u,v),c_d(L_1(u,v)))\\ &\wedge\ \forall (u,v)\in E_2,\text{soft}(\text{MatchedToSomeEdges}(M,u,v),c_a(L_2(u,v)))\\ &\wedge\ \forall (u,v)\in E_1,\text{soft}(\text{MatchAllEdgeLabels}(M,u,v),c_s(L_1(u,v),L_2(M(u),M(v))))\ .\end{aligned}$$

In this case, violation costs of soft constraints are defined by edit costs. For the nodes of G_1 (resp. G_2), the cost of violation of the MinMatch constraint is equal to the edit cost of the deletion (resp. addition) of node labels as this constraint is violated when a node of G_1 (resp. G_2) is not matched to a node of G_2 (resp. G_1), thus indicating that this node must be deleted (resp. added). The cost of the soft MatchAllNodeLabels constraint is equal to the edit cost of substituting the label of u by the label of the node it is matched to in G_2. Similar soft constraints are posted on edges to define deletion, addition and substitution costs.

Finally, one may also define multivalent graph matching problems, such that one node may be matched to a set of nodes. Let us consider for example the extended graph edit distance ($\mathcal{EGED}$) defined in [14]. This distance extends $\mathcal{GED}$ by adding two edit operations for splitting and merging nodes. Let us note $c_p(u,U)$ (resp. $c_m(U,u)$) the edit cost associated with the splitting of the node $u\in N_1$ into the set of nodes $U\subseteq N_2$ (resp. the merging of the set of nodes $U\subseteq N_1$ into the node $u\in N_2$). $\mathcal{EGED}$ may be defined by replacing the hard MaxMatch constraint of $\mathcal{GED}$ by two soft MaxMatch constraints which respectively evaluate split and merged nodes, i.e.,

$$\begin{aligned}&\forall u\in N_1,\text{soft}(\text{MaxMatch}(M,u,1),c_p(u,M(u))\\ &\wedge\ \forall u\in N_2,\text{soft}(\text{MaxMatch}(M,u,1),c_m(u,M(u)))\ .\end{aligned}$$

6 Comet Prototype

As illustrated in Fig. 1, constraint-based graph matching in our Comet prototype is done in two parts. First, high level constraints modeling the problem are posted. Second, a synthesizer is called to solve the problem by means of CP and/or CBLS techniques.

Canonical Form of Modeling Constraints High-level constraints, called modeling constraints, implement the `GMConstraint<Mod>` interface. Such constraints, like those described in section 4, are stated on the nodes, edges and labels of the graph and are posted by the model, implemented by `Matching<Mod>`, as hard or soft constraints. For soft constraints, an additional parameter specifies the cost of a violation.

To easily state characteristics of the matching, we introduce *canonical* constraints aggregating all the modeling constraints of a type. For example, all MinMatch and MaxMatch constraints will be aggregated into a single cardinality canonical constraint knowing the lower and upper bounds of the matchings of each node. The model maintains two constraint stores, for hard and soft canonical constraints respectively. When a hard (resp. soft) modeling constraint is posted to the model, its `postCanonical` (resp. `postSoftCanonical`) method is called with the hard (resp. soft) constraint store. This method can add or modify canonical constraints within the store. Typically a modeling constraint will only modify its associated canonical constraint. The key concept is that there must not exist more than one canonical constraint of each type, identified by a unique string, in a constraint store. Apart for stating characteristics, this has another benefit: global constraints can be generated by the synthesizer.

Once the model is closed, i.e. no more constraints may be added, each canonical constraint gets a chance to modify itself or other constraints, in order to make the whole model canonical, through the `canonify` method. In a canonical model, we cannot add any modeling constraint without altering the described matching problem. For example, the Injective constraint adjusts the cardinality constraint if every node of a graph belongs to an injective set. The `canonify` method should report the canonical constraints it has modified, so that the model can iterate the procedure, using a static dependency graph of the canonical constraints, untill it reaches a fix-point. Methods of the different interfaces are depicted below.

```
1  class Matching<Mod> { ...
2    void post(GMConstraint<Mod> constraint);
3    void postSoft(GMConstraint<Mod> constraint, int cost);
4  }
5  interface GMConstraint<Mod> {
6    void postCanonical(GMConstraintStore<Mod> store);
7    void postSoftCanonical(GMConstraintStore<Mod> store, int cost);
8  }
9  interface CanonicalGMConstraint<Mod> {
10   string getId();
11   set{string} canonify(GMConstraintStore<Mod> store);
12   void postCP(Solver<CP> cp, GMVarStore<CP> vars);
13   void postLS(SetConstraintSystem<LS> S, GMVarStore<LS> vars);
14 }
```

Synthesizer Once the model is closed, a synthesizer is called to effectively solve the problem. The canonical representation of the model allows to compute various kinds of characteristics such as whether the matching is functional, univalent or mul-

tivalent, or even the class of the problem (such as those defined in section 4). The `GMCharacteristic<Mod>` interface allows to define such characteristics.

This synthesizer has three tasks, i.e., create variables, post the CP and/or CBLS constraints, and perform the search.

Creating Variables. To represent the matching, a variable x_u is associated to each node $u \in N$; the value of x_u denotes the matching of node u, that is $M(u)$. We assume that nodes are represented by positive integers. The type and domain of these variables depend on the MinMatch and MaxMatch constraints, as described below:

MinMatch	MaxMatch	Type	Domain
1	1	`int`	N
0	1	`int`	$N \cup \{\perp\}$
Otherwise		`set{int}`	2^N

The $\perp$ value denotes that $M(u) = \varnothing$. It is implemented as the negative node number, i.e., $-u$, ensuring a unique $\perp$ value for each node of a graph. Of course, for nodes in G_1, the domain is restricted to N_2, and similarly for nodes in G_2.

Depending on the constraint solver chosen by the synthesizer (complete incremental search (CP) or incomplete local search (LS)), the variables are declared as CP variables or LS variables in the Comet language. As set variables do not yet exist in Comet CP, we have implemented these with a simple boolean array. A similar limitation led us to reimplement the constraint interface in Comet LS.

Since a variable is declared for nodes in G_1 as well as for nodes in G_2, the matching M is redundantly represented. Depending on the chosen solver, channeling constraints (CP) or Comet invariants (LS) are added to relate these two sets of variables. This redundant representation allows the solver to choose the best variables to construct a solution and to perform the search.

The association between a node and its variable is available to the various constraints through the `GMVarStore<CP>` and `GMVarStore<LS>` interfaces. As the creation of variables is likely to be the same for every synthesizer, the default implementations of these interfaces handle the variable creation in their constructor.

Posting the Constraints and Performing the Search. Once the variables are created, the synthesizer asks the canonical (hard and soft) constraints to post themselves with the `postCP` or `postLS` methods. These methods take two arguments: the solver and the `GMVarStore` containing the associations between nodes and variables. In CP, soft constraints are implemented by an objective function. In LS, hard constraints are either handled by a neighborhood ensuring that they cannot be violated, or they are posted as soft constraints with much higher violation costs.

A synthesizer solves a problem either in CP or in LS. The default choice of CP or LS, as implemented by `DefaultGMSynthesizer`, depends on the constraints: if the maximum number of matched nodes is 1 for every node of one graph and if all constraints are hard ones, then the synthesizer chooses CP, and the variables associated with these nodes are used as choice variables in the tree search; otherwise, the synthesizer chooses LS and the variables associated with the graph with the fewest set variables are used for defining neighborhoods.

Implementing the Constraints We now focus on how constraints for modeling matching problems are implemented in our framework.

Node Cardinality. For univalent matchings, the cardinality constraints (MinMatch, MaxMatch, and CardMatch) are already implemented by variable domains. For multivalent matchings, nodes are associated with set variables and we post inequality constraints on the cardinality of these sets.

If the matching is a surjective function from N_1 to N_2, i.e. MinMatch$(M,N_2,1)$ and MaxMatch$(M,N_1,1)$, a redundant constraint is added to the CP solver. This is a particular case of the global cardinality constraint `cardinality`$(N_2,1,[x_u|u \in N_1],\#N_1)$ [19] which here holds when at least 1 variable (and at most $\#N_1$) is assigned to each value of N_2. A similar constraint is posted if the matching is surjective from N_2 to N_1.

Injective Set. For univalent matchings, Injective(M,U) constraints are implemented by `alldifferent`$([x_u|u \in U])$ constraints. Note that we associate a different $\perp$ value to every different node u (defined by $-u$) so that `alldifferent` is not violated when several nodes are matched to $\perp$.

For multivalent matchings such that some variables in the set U are implemented as `set{int}` variables, additional constraints of the form $x_{u1} \cap x_{u2} = \varnothing$ (with x_{u1} and x_{u2} set variables) and $x_v \notin x_u$ (with x_u a set variable and x_v an integer variable) are posted.

Edges. If we consider univalent matchings, MatchedToSomeEdges(M,u,v) (resp. MatchedToAllEdges(M,u,v)) is easily implemented as $(x_u,x_v) \in E_1 \cup E_2$ (resp. $x_u \neq \perp \wedge x_v \neq \perp \Rightarrow (x_u,x_v) \in E_1 \cup E_2$). When the matching is multivalent, LS constraints are generated.

Additional Constraints. The system is open and modular so that new constraints may be defined. For instance, we introduced the constraint CommonNeighbor(u,v) which holds if $M(u)$ and $M(v)$ share at least one neighbor.

Current Limitations of the System The prototype is about 4,200 lines of Comet code. In the current implementation, the cost of soft constraints must be a fixed value; hence $\mathcal{GED}$ and $\mathcal{EGED}$ are not yet supported. The CP part of the current prototype does not support soft constraints and MatchedToEdges constraints for multivalent matching. This is not very limitative as CP is not really adapted for these matching problems.

In the current implementation, a matching problem is solved either with CP or with LS. In the future, we plan to allow the solver to combine CP with LS.

The analysis of the characteristics of the matching problem is rather limited. It can however detect the standard matching problems. We plan to add additional global constraints in the CP part in order to speed up the search process. In particular, we plan to integrate the redundant constraints of [20], the filtering algorithm of [17] for the subgraph isomorphism problem, and the filtering algorithm of [16] for the graph isomorphism problem.

The metaheuristics in the LS part are still basic (tabu search); this will be extended and adapted to matching problems. In particular, we plan to implement the reactive tabu search algorithm of [8].

7 Experimental Results

We report experiments on the subgraph isomorphism problem and on a pattern recognition problem using CP, and on the maximum common subgraph using LS.

7.1 Subgraph Isomorphism Using a CP Solver

We evaluate our system on the subgraph isomorphism problem as modeled in Fig. 1. Given the characteristics of the problem, the default synthesizer uses CP. Our model is compared with the state of the art Vflib C++ library [2].

The benchmark contains two families of randomly generated graphs. In the first family (P*), graphs 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 [21]. We only report experiments on graphs generated with the standard value $\lambda = 2.5$. Each class contains 20 different instances. For each instance, we first generate a connected target graph which node degrees are bounded between 5 and 8. Then, a connected pattern graph is extracted from the target graph by randomly selecting 90% of nodes and edges. All instances of classes P200, P600 and P1000 are feasible instances that respectively have 200, 600, and 1000 nodes.

The second family (V*) is taken from the Vflib benchmarks [22]. Class V200 contains the 100 instances from the class called si2_r001_m200 in Vflib. These instances were randomly generated using a uniform distribution, the target graph has 200 nodes, and the source graph has 40 nodes, i.e. 20% of the target (see [22] for details). In order to assess the modularity of our approach, we generated the classes V200+k (with $k \in \{1,2,3,4,5\}$) by adding k additional CommonNeighbor(u,v) constraints in the Subgraph Isomorphism model. In order to ensure the existence of a solution (instances without solution are all solved by the CP solver in about a second), we have randomly chosen (u,v) from the pairs of nodes satisfying the additional constraint from the solutions found. Such side constraints cannot be directly handled by Vflib; so we added to Vflib an algorithm to filter the solutions satisfying these additional constraints.

Table 1 gives for every class the percentage of solved instances within a CPU time limit of 600s on a Core 2 Quad (only one core used) 2,4 Ghz with 2Go of RAM. It also gives the execution time of the solved instances (mean, standard deviation, minimum and maximum times in seconds).

On the P* instances, the synthesized CP algorithm solves much more instances and is also much more efficient than the standard Vflib. On the V200 class, Vflib solves more instances. However, as Vflib is not able to actively exploit additional constraints to prune the tree during the search, results of Vflib on V200+k classes are almost identical than for V200; there is a slight overhead for the postprocess filtering the solutions. On the contrary, the synthesized CP solver can exploit these additional constraints during the search, increasing the number of solved instances and reducing the computation time. The more additional constraints, the more solved instances and the less computation time. From V200+4, the CP solver outperforms Vflib. This clearly shows the interest of a constraint-based approach compared to specialized algorithms.

Table 1. Comparison of synthesizer/CP and Vflib on $\mathcal{SI}$.

	Vflib					Synthesizer/CP				
class	solv.%	mean	std	min	max	solv.%	mean	std	min	max
P200	75	61.8	93.0	0.5	309.2	100	6.1	1.9	2.6	9.5
P600	0	-	-	-	-	100	234.3	70.5	105.8	402.6
P1000	0	-	-	-	-	15	522.8	107.8	370.3	599.1
V200	82	63.2	112.9	0.0	574.7	62	84.3	132.2	0.8	530.9
V200+1	82	63.8	114.2	0.0	583.3	71	69.5	108.4	0.8	524.7
V200+2	82	64.4	115.1	0.0	582.9	77	63.5	117.1	0.7	540.2
V200+3	82	64.9	115.8	0.0	582.7	79	50.9	119.0	0.7	531.9
V200+4	82	65.5	116.8	0.0	584.8	85	40.0	93.8	0.7	510.3
V200+5	81	59.4	102.8	0.0	459.6	87	31.5	84.5	0.7	507.2

It should be noticed that the CP approach could be made more efficient by a full implementation of the iterative filtering described in [17].

7.2 Pattern Recognition Using a CP Solver

We now illustrate our solver on a pattern recognition problem which involves finding patterns in images. Graphs are generated from images by extracting interest points (corresponding to salient points) and computing a Delaunay triangulation on them. Finding patterns in images amounts to finding connected sets of faces in graphs modeling images. This problem lies part-way between subgraph isomorphism and induced subgraph isomorphism as some edges in the target graph are mandatory and some others are optional. This problem may be solved with subgraph isomorphism, assuming post-processing is done on the found solutions to check that all mandatory edges are matched. In a pattern recognition context, it may be meaningful to introduce an additional constraint stating that the distance between two nodes in the pattern should be similar to the distance of the corresponding target nodes (up to a small delta value). We have implemented this constraint as a CP propagator with a forward-checking consistency level. Note that such constraints cannot be handled with Vflib as it can only handle equality constraints between matched labels.

Table 2 gives the results for target graphs with 100, 500 and 1000 nodes. Five pattern graphs are extracted from each target graphs by selecting a connected subset of faces which respectively contains 5%, 10%, 20%, 33%, and 50% of the target faces. Table 2 shows us that Vflib is better on small instances, but is outperformed by our system on larger ones. Using the additional constraint globally improves the performances, even though only forward checking has been used. This shows the interest of the more flexible constraint-based approach in real-world applications.

7.3 Maximum Common Subgraph Using an LS Solver

We now evaluate the feasibility of our approach on the maximum common subgraph ($\mathcal{MCS}$) problem as described in Section 5, using an LS solver. Existing solvers like vflib cannot handle the MCS problem. Also, [3,4,5,7] do not handle the MCS. Different

Table 2. Execution time in seconds of subgraph isomorphism for pattern recognition without ($\mathcal{SI}$) and with ($\mathcal{SI}+$) additional constraint on distances. Rows represent the number of vertices of the target graphs, columns show the sizes of the pattern graphs as a percentage of the target.

		Synthesizer/CP					Vflib				
		5%	10%	20%	33%	50%	5%	10%	20%	33%	50%
$\mathcal{SI}$	100	0.8	0.5	0.7	**0.1**	0.2	**0.0**	**0.0**	**0.0**	2.0	**0.0**
	500	19.3	4.7	**10.5**	**15.8**	**30.7**	**0.1**	**0.1**	246.7	192.3	–
	1000	**30.6**	**595.8**	**119.0**	**152.3**	–	86.7	–	–	–	–
$\mathcal{SI}+$	100	0.3	0.1	0.1	0.1	0.2					
	500	3.0	4.4	9.5	16.9	28.9					
	1000	16.1	47.8	82.5	148.0	–					

Table 3. Maximum common subgraph with LS. Average (standard deviation) of execution time, number of iterations, and percentage of edges in the best found solution. The model has been executed 10 times per instance.

class	time	iterations	edges%
M25	8.5 (2.5)	7768.1 (2301.3)	48.3 (1.1)
M50	33.9 (10.7)	8023.8 (2543.3)	40.2 (0.5)
M100	141.5 (46.4)	8398.4 (2755.0)	34.5 (0.2)

complete approaches have been compared on the MCS in [23] but experimental results are limited to graphs up to 30 nodes so that we have not compared our approach on these benchmarks.

The benchmarks are also taken from the Vflib benchmarks [2]. The classes M25, M50 and M100 contains 20 instances from the classes called mcs50_r02 in Vflib. The graphs have respectively 25, 50 and 100 nodes and have been randomly generated (see [22] for details). It is known that these graphs have a common *induced* subgraph with 50% of the nodes, but this lower bound does not provide any information on the size of maximum common partial subgraph in terms of edges.

A basic tabu search is generated by the synthesizer. The MaxMatch(M,N_2) hard constraint is maintained through the neighborhood, by either swapping the value of two matched nodes in G_1, by matching a node in G_1 to an unmatched node of G_2, or by removing a matching in G_1. The time limit is fixed at 20,000 iterations, but the search stops after 5,000 iterations without global improvement. A random new solution is also generated after 1,000 iterations without global improvement. The results are reported in Table 3.

It is difficult to compare an LS approach with complete algorithms for maximum common subgraph as these algorithms cannot handle graphs with 100 nodes [24]. This also justifies the choice of an LS solver by our default synthesizer. The generated LS solver could be improved in many ways. These benchmarks are presented to assess the feasibility of generating an LS solver by the synthesizer.

8 Conclusion

Measuring graph similarity is a key issue in many applications. We proposed a new constraint-based modeling language for defining graph matching problems by means of constraints. It covers both univalent and multivalent matchings, and we have shown that it may be used to define many different existing matching problems in a very declarative way. Such a constraint-based formulation of the different matching problems actually stressed out their shared features and differences in a very concise way.

We have built a synthesizer which is able to automatically generate a Comet program to solve matching problems modeled with our language. Depending on the characteristics of the matching, the synthesizer either uses a branch and propagate approach —which is better suited for computing exact matchings such as (sub)graph isomorphism— or a local search approach —which is better suited for computing error-tolerant matching such as graph edit distances. First experimental results showed the feasibility of our approach on subgraph isomorphism and maximum common subgraph problems.

As future work, we will extend the prototype to lift the current limitations : handling soft constraints in CP, integrating new filtering algorithms in CP, improving the analysis of the matching characteristics, integrating several LS metaheuristics, and combining CP and LS solvers.

Acknowledgments The authors want to thank the anonymous reviewers for their helpful comments. Christine Solnon acknowledges an ANR grant BLANC 07-1_184534: this work was done in the context of project SATTIC. This research is also partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy).

References

1. McKay, B.D.: Practical graph isomorphism. Congressus Numerantium **30** (1981) 45–87
2. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the vf graph matching algorithm. In: ICIAP. (1999) 1172–1177
3. Umeyama, S.: An eigendecomposition approach to weighted graph matching problems. IEEE Transactions on Pattern Analysis and Machine Intelligence **10**(5) (1988) 695–703
4. Almohamad, H., Duffuaa, S.: A linear programming approach for the weighted graph matching problem. IEEE Transactions on Pattern Analysis and Machine Intelligence **15**(5) (1993) 522–525
5. Zaslavskiy, M., Bach, F., Vert, J.: A path following algorithm for the graph matching problem. In: Image and Signal Processing. LNCS 5099, Springer (2008) 329–337
6. Cross, A., Wilson, R., Hancock, E.: Inexact graph matching using genetic search. Pattern Recognition **30** (1997) 953–970
7. Champin, P.A., Solnon, C.: Measuring the similarity of labeled graphs. In: ICCBR 2003. LNAI 2689, Springer (2003) 80–95
8. Sorlin, S., Solnon, C.: Reactive tabu search for measuring graph similarity. In: workshop on Graph-based Representations in Pattern Recognition. LNCS 3434, Springer (2005) 172–182
9. Sammoud, O., Solnon, C., Ghedira, K.: Ant Algorithm for the Graph Matching Problem. In: EvoCOP 2005. LNCS 3448, Springer (2005) 213–223

10. Monette, J.N., Deville, Y., Hentenryck, P.V.: AEON: Synthesizing scheduling algorithms from high-level models. In: Proceedings of 2009 INFORMS Computing Society Conference. (2009)
11. Vosselman, G.: Relational Matching. Springer , LNCS 628 (1992)
12. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recognition Letters **18** (1997) 689–694
13. Zaslavskiy, M., Bach, F., Vert, J.P.: A path following algorithm for graph matching. In: Image and Signal Processing. Volume 5099 of LNCS., Springer (2008) 329–337
14. Ambauen, R., Fischer, S., Bunke, H.: Graph Edit Distance with Node Splitting and Merging. In: IAPR Workshop on Graph-based Representation in Pattern Recognition. LNCS 2726, Springer (2003) 95–106
15. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)
16. Sorlin, S., Solnon, C.: A parametric filtering algorithm for the graph isomorphism problem. Constraints **13**(4) (2008) 518–537
17. Zampelli, S., Deville, Y., Solnon, C., Sorlin, S., Dupont, P.: Filtering for Subgraph Isomorphism. In: CP 2007. LNCS 4741, Springer (2007) 728–742
18. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
19. Quimper, C., Golynski, A., Lopez-Ortiz, A., van Beek, P.: An efficient bounds consistency algorithm for the global cardinality constraint. Constraints **10**(1) (2005) 115–135
20. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical. Structures in Comp. Sci. **12**(4) (2002) 403–422
21. Barabasi, A.L.: Linked: How Everything Is Connected to Everything Else and What It Means. Plume (2003)
22. De Santo, M., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recogn. Lett. **24**(8) (2003) 1067–1079
23. Bunke, H., Foggia, P., Guidobaldi, C., Sansone, C., Vento, M.: A comparison of algorithms for maximum common subgraph on randomly connected graphs. In: Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition, Springer-Verlag, LNCS2396 (2002) 123–132
24. Sorlin, S.: Mesurer la similarité de graphes. PhD thesis, UniversitClaude Bernard, Lyon I, France (2006)