A Constraint Programming Approach to Subgraph Isomorphism

Stéphane Zampelli

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences de l'Ingénieur

June 2008

Ecole polytechnique de Louvain
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis committee:

Yves Deville (director)	INGI, UCLouvain, Belgium
Jean-François Puget	Ilog, France
Christine Solnon	LIRIS, University of Lyon, France
Peter Van Roy	INGI, UCLouvain, Belgium
Marc Lobelle (president)	INGI, UCLouvain, Belgium

Acknowledgments

This work could not have been achieved without the following persons.

I would like to thank my supervisor Yves Deville for his continuous support during this research, and all the members of the thesis committee: Christine Solnon, Peter Van Roy, Jean-François Puget and Marc Lobelle. I would also like to thank Pierre Dupont for supporting this research at earlier steps of this thesis.

I would like to thank the constraint group at UCLouvain, for the constructive discussions, their help in learning constraint programming, and the friendly atmosphere during those research years. Thank you Grégoire Dooms, Nguyen Tran Sy, Jean-Noël Monette, Sébastien Mouthuy, Pierre Schaus, Pham Quang Dung, and Ho Trong Viet. I would like to thank Jérome Callut for his friendship and numerous interesting discussions.

I would like to thank Raphaël Collet for his disinterested help at the beginning of my thesis, as he did for many young researchers.

I would like to thank all the people I had the chance to have a collaboration with. Belaïd Benhamou and Mohammed Réda Saïdi helped me in understanding and exploring symmetries. Martin Mann and Rolf Backofen welcomed me at their lab in Freiburg where we did an enthusiastic job about decomposition. I thank Christine Solnon for her interest about my works and her continuous helpful comments and insights.

I would like to thank everyone at the INGI department, including the technical staff, who provided a great place to study and research.

Part of my time was spent at the cafeteria of the department. I would like to thank all the folks that hanged out there during noons and nights, and who started interesting discussions about a large number of subjects.

I would like to thank all my friends that recognize themselves in this sentence.

I would like to thank Quentin Delval for his friendship during those years in Louvain-la-Neuve.

I would like to thank Yasmine Jouhari for her love and support during this thesis.

I would like to thank my whole family, especially my mother, my father, and Adriana Rossi. My gratitude goes far beyond any word.

This research was supported by the Walloon Region, project BioMaze (WIST 315432) and project Transmaze (WIST516207), and by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy).

Contents

1

Introduction

The objective of this dissertation is to provide an *expressive and efficient declarative framework* for subgraph matching problems using constraint programming.

Subgraph Matching A graph is a set of points where two points are joined by a line if they are related. Graphs can model, for example, the representation of the relationships inside a community where points are persons, and there is a line between two persons if they know each other. Subgraph matching is the identification of a subgraph inside a target graph that is exactly the same as a given graph. For example, one may want to extract a subgraph of an initial community graph that is exactly the same as a given graph of interest, identifying subgroups of people that share exactly the same structural relationship than another given group.

Challenges Subgraph matching is a challenging problem. The first challenge is that this problem is NP-Complete. This means that there is little hope that a polynomial time algorithm can be found to solve it. This calls for practically efficient algorithms. The second challenge is expressiveness. The user may want to state a property about the extracted subgraph. For instance, the user may want to extract two subgraphs that are related by a path inside the initial graph. Expressiveness is challenging because the matching algorithm must usually be redesigned and reimplemented for each new feature.

Existing Approaches Existing approaches consist in dedicated algorithms. It was previously argued that dedicated algorithm such as `vflib` [LV02] represents the state-of-art in subgraph matching. Moreover, many works in subgraph matching use their own dedicated implementation, based on techniques presented in the Ullmann algorithm [Ull76] (see [Wer06] for an example). Netmatch [FGP+07] is an instance of the expressiveness approach, where for instance path properties between two input graphs can be stated.

Constraint Programming Constraint Programming (CP) solves constraint satisfaction problems (CSP) by combining constraint propagation with search. A CSP is specified by a set of variables that range over a set of possible values called their domain and a set of constraints to be satisfied. An assignment of each variable to a value in its domain is a solution to the CSP if it satisfies all constraints.

The set of constraints is used to filter the domains of the variables. This filtering is called propagation. Propagation is usually not sufficient to find a solution. Instead, the CSP is simplified by restricting the domain of variables. This restriction is called branching. The propagation can then remove further values. The interleaving of branching and propagation is called search. Branching usually ensures that all possible restrictions of the domain of the variables are explored so that the search is complete. This search can be viewed as a tree. The branching determines the form of the search tree, while the exploration is the way the tree is traversed.

Most modern CP systems offer high level abstractions in order to be expressive. Global constraints are constraints aggregating a common pattern of constraints and using an efficient filtering algorithm for this conjunction of simpler constraints. They allow the user to declare problem specific constraints. Model systems also offer an abstracted search, as the branching and the exploration can be declared. The user just ignores the search mechanism and is faced only with modelling choices. Apart from their declarive feature, modern CP systems have also proved their efficiency for NP-Complete problems.

Subgraph Matching in CP Regarding graph matching, the constraint programming approach has already given some efficient practical results for graph isomorphism [SS08] ; for maximum common subgraph, it was claimed that constraint programming is the most ef-

fective approach on a DIMACS benchmark [Rég03]. Several works [McG79, Rég95, Rud98a, LV02] have already tackled the subgraph matching problem using constraint programming. However the subgraph matching problem was used as a convenient benchmark for CP techniques. Several CP techniques were tested, such as path consistency [McG79], optimal AC filtering [Rég95], or the comparison of different level of consistencies [LV02]. On the contrary, our goal is to establish CP as the state-of-the-art approach for subgraph matching, replacing dedicated algorithms. [Rud98b] describes a modeling of subgraph matching in CP arguing that it could be expressive and efficient, but remains at a pure modeling level.

The CP framework is thus attractive because it can face the computational difficulty of subgraph matching, and include subgraph matching inside a declarative framework.

Contributions The contributions of this thesis are the followings:

- We propose an efficient and expressive declarative framework for graph matching. It uses constraint programming and exploits graph and map variables, instead of ground objects. This allows to handle various graph matching problems, instead of the traditional development of various dedicated and specific algorithms.

- We develop a novel subgraph isomorphism global constraint. This global constraint is able to use the semantic of the subgraph isomorphism as well as the global structure of the two input graphs. It is shown that it is the state-of-the-art filtering algorithm, compared to dedicated algorithms and other CP approaches.

- We show how existing CP approaches can be integrated and generalized in a declarative framework using graph and map variables.

- We show that the proposed constraint programming framework is performant compared to the state-of-the-art dedicated algorithms for graph matching.

- It is shown how symmetries can be used in the subgraph isomorphism problem, taking into account the pattern graph and the target graph.

- State-of-the-art decomposition techniques are unable to deal with subgraph isomorphism. We therefore adapt decomposition techniques to the subgraph isomorphism problem.
- When the number of solutions is high, we show that our decomposition approach outperforms all other approaches, including CP.

Outline Chapter 2 introduces the common ideas behind existing dedicated approaches, and details their algorithm. The existing CP modelings are also presented, and their associated propagators are detailed.

Chapter 3 faces the challenge of designing and implementing a declarative and efficient CP framework for subgraph matching. Map variables are introduced; they represent the matching function. The originality of map variables lies in the fact that the domain and codomain sets are not ground.

Associated propagation rules are developed, and an original algorithm pruning the Map variables is presented. Afterwards it is shown how graph variables can be used to model a wide variety of graph matching problems through a *single* matching constraint. The associated matching propagator is also presented.

Chapter 4 faces the challenge of developing a global propagator that takes into account both pattern and target graphs and the current state of the domains, instead of the conjunction of locally arc-consistent constraints. A novel propagator is presented, based on the labelling of the nodes that iteratively recompute the labels based on their neighboors' labels. We show experimentally that for difficult problems, our novel propagator beat all previous dedicated and CP approaches.

Chapter 5 deals with symmetries. It is shown how *all* global variable and value symmetries can be detected by computing the set of automorphisms of the pattern graph, and how they can be broken. Experimental results show that global symmetry breaking is an effective way to increase the number of tractable instances of the subgraph isomorphism problem. We show that local symmetries can be detected by computing the set of automorphisms on various subgraphs of the target graph. Experimental results show that global symmetries solve more difficult instances compared to local symmetries.

Chapter 6 deals with decomposition. It is explained why state-of-the-art decomposition techniques do not apply to the CP approach of subgraph isomorphism. Hence, a dedicated decomposition approach is developed, based on a precomputation of a variable heuristics able to

decompose the subgraph isomorphism problem during search. The experiments suggest that this approach is effective on graphs that contain a lot of solutions, making this decomposition approach a promising tool for subgraph isomorphism enumeration.

We finally conclude this thesis and discuss future works.

Publications Part of the results of this thesis have already been published:

- Yves Deville, Grégoire Dooms, Stéphane Zampelli, and Pierre Dupont. Cp(graph+map) for approximate graph matching. *1st International Workshop on Constraint Programming Beyond Finite Integer Domains, CP2005*, pages 33–48, 2005
- Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate constrained subgraph matching. In *Principles and Pratice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 832–836, 2005
- Stéphane Zampelli, Yves Deville, and Pierre Dupont. Symmetry breaking in subgraph pattern matching. *Sixth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'06)*, 2006
- Stéphane Zampelli, Yves Deville, and Pierre Dupont. Symmetry breaking in subgraph pattern matching. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, pages 203–218. ISTE Hermes, 2007
- Stéphane Zampelli, Yves Deville, Christine Solnon, Sébastien Sorlin, and Pierre Dupont. Filtering for subgraph isomorphism. In *Proc. 13th Conf. of Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 728–742. Springer, 2007
- Yves Deville, Stéphane Zampelli, and Grégoire Dooms. Combining two structured domains for modeling various graph matching problems. In F. Fages, F. Rossi, and S. Soliman, editors, *Recent Advances in Constraint Programming*. Springer-Verlag, 2008

2

State of the Art of Exact Subgraph Isomorphism

The goal of this chapter is to describe and compare dedicated algorithms for subgraph isomorphism and existing CP models. First, the fundamental problem definitions of graph matching are described, based upon the concept of *morphism*, which includes subgraph isomorphism as a special case. Two dedicated algorithms for the subgraph isomorphism, the Ullmann algorithm together with the state-of-the-art vflib algorithm, are studied and compared. The graph morphism problems are then modeled in constraint programming, reviewing the literature on the subject, and existing implementation of the constraints is given. We conclude by a comparison of the complexities of the Ullmann and vflib algorithm against the existing CP approach, showing why it is usually believed that a declarative approach cannot compete with dedicated algorithms.

The next section introduces graph theory and the constraint programming framework.

2.1 Background

2.1.1 Graph Theory

Graphs are mathematical objects useful to represent binary relationships between a set of elements. They capture the idea of network, seen as a set of pairwise interconnected elements. The relationships can be directed or undirected. Suppose we want to represent social relationships. A

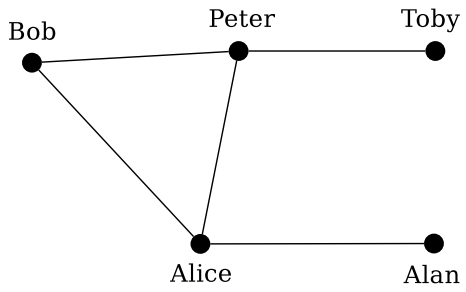

Figure 2.1: In this social network graph, two persons are related by a line if they know each other.

social network is made of persons, and some persons know each other. Each person is an element of the graph, and two persons are joined by a undirected line if those two persons know each other. Moreover, the points may be labeled, for example by name and surname. The lines may also be labeled, to denote the type of relationship the two persons, for example 'family' or 'friend'. Figure 2.1 gives an example of a graph representation of a social network. Similarly, graphs can also be used to represent other types of networks, like communication networks, phylogeny trees, the world wide web, etc. Once the network has been represented as a graph, many questions can be answered about this network by using graph theory. The following graph theory definitions are taken from [BE05].

Graph A *graph* $G = (V, E)$ is an object formed by a set V of *vertices* (nodes) and a set E of *edges* (links) that join (connect) pairs of vertices ($E \subseteq V \times V$). The vertex set and edge set of a graph G are also denoted by $V(G)$ and $E(G)$, respectively. The cardinality of V is usually denoted by n, the cardinality of E by m. By definition, a pair of vertices can be related by at most one edge. An edge can relate a vertex to itself and such an edge is called a *selfloop*. The two vertices joined by an edge are called *endvertices*. If two vertices are joined by an edge, they are *adjacent* and we call them *neighbors*. Graphs can be *undirected* or *directed*. In undirected graphs, the order of endvertices of an edge is irrelevant. An undirected edge joining vertices $u, v \in V$ is denoted by $\{u, v\}$. In directed graphs, each directed edge (arc) has an *origin* (*head*) and a *destination* (*tail*). An edge with origin $u \in V$ and destination $v \in$

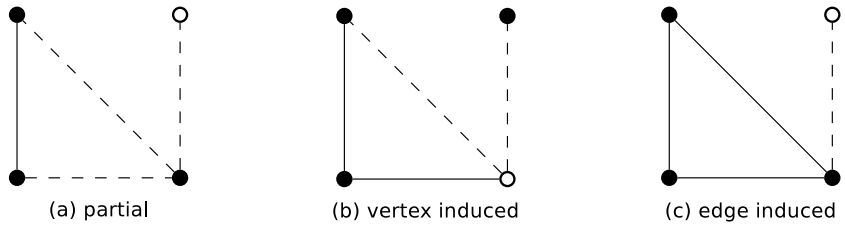

Figure 2.2: Example of partial, induced and edge-induced subgraphs.

V is represented by an ordered pair (u, v). In an undirected graph, $\{u, v\}$ and $\{v, u\}$ are the same. For a directed graph $G = (V, E)$, the *underlying undirected graph* is the undirected graph with vertex set V that has an undirected edge between two vertices $u, v \in V$ if (u, v) or (v, u) is in E. We do not consider *mixed graphs* that can have directed edges as well as undirected edges. A *labeled graph* is a tuple $G = (V, E, \alpha, \beta)$ where (V, E) is a graph, α is a function $V \to \mathbb{N}$ that associates each node with a label, and β is a function $E \to \mathbb{N}$ that associates each edge with a label. The set $\mathbb{N}$ is the usual set of integers.

Subgraphs Reasoning about a subset of a network leads to the notion of subgraph. A subgraph with respect to a graph is what a string is with respect to a text. A graph $G' = (V', E')$ is a *(partial) subgraph* of the graph $G = (V, E)$ if $V \subseteq V'$ and $E' \subseteq E$. It is a *(vertex-)induced subgraph* if E' contains all edges $e \in E$ that join vertices in V', i.e. $E' = E \cap (V \times V')$. The induced subgraph of $G = (V, E)$ with vertex set $V' \subseteq V$ is denoted by $G[V']$. The *(edge-)induced subgraph* with edge set $E' \subseteq E$, denoted by $G[E']$, is the subgraph $G' = (V', E')$ of G, where V' is the set of all vertices in V that are endvertices of at least one edge in E'. If C is a proper subset of V, then $G \setminus C$ denotes the induced graph obtained from G by deleting all vertices in C and their incident edges. If F is a subset of E, $G \setminus F$ denotes the partial graph obtained from G by deleting all edges in F. Figure 2.2 shows an example, where the selected nodes and edges are plain.

Degree and neighbors Given a social network, one may ask the number and the list of persons a person knows. This is achieved through the concepts of degree and neighbors of a node. The *degree* of a vertex

v of an undirected graph $G = (V, E)$, denoted by $d(v)$, is the number of edges that have v as an endvertex. The set of edges that have v as an endvertex is denoted by $\Gamma(v)$. The set $N(v) = \{u \in V | (v, u) \in E\}$ is the set of neighbors of v. In a directed graph $G = (V, E)$, the *out-degree* of $v \in V$, denoted by $d^+(v)$, is the number of edges in E that have origin v. The *in-degree* of $v \in V$, denoted by $d^-(v)$, is the number of edges in E with destination v. The set of edges with origin v is denoted by $\Gamma^-(v)$, the set of edges with destination v is denoted by $\Gamma^+(v)$. The set of *successors* of a node v is the set of tails of the edges $\Gamma^+(v)$ and is denoted $N^+(v)$. The set of *predecessors* of a node v is the set of heads of the edges $\Gamma^-(v)$ and is denoted $N^-(v)$. The *neighbors of a subgraph* $S = (V_S, E_S)$ of an undirected graph $G = (V, E)$ are the set $N(S) = \{n \in V \mid \exists\ m \in V_S : (n, m) \in E\}$. The *in-neighbors of a subgraph* $S = (V_S, E_S)$ of a directed graph $G = (V, E)$ are the set $N^-(S) = \{n \in V \mid \exists\ m \in V_S : (m, n) \in E\}$. The *out-neighbors of a subgraph* $S = (V_S, E_S)$ of a directed graph $G = (V, E)$ are the set $N^+(S) = \{n \in V \mid \exists\ m \in V_S : (n, m) \in E\}$. If the graph under consideration is not clear from the context, notations can be augmented by specifying the graph as an index. For example, $d_G(v)$ denotes the degree of v in G. The maximum and minimum degree of an undirected graph $G = (V, E)$ are denoted by $\Delta(G)$ and $\delta(G)$, respectively. The average degree is denoted by $d(G) = \frac{1}{|V|} \sum_{u \in V} d_G(v)$.

Walks, paths, and cycles Is there a way to reach a person B from a person A ? What is the exact list of intermediates to reach a person B from a person A ? Can a person reach himself through distinct intermediates ? Those questions can be answered with the concepts of walk, path and cycle. A *walk* from x_0 to x_k in a graph $G = (V, E)$ is an alternating sequence $x_0, e_1, x_1, e_2, x_2, \ldots, x_{k-1}, e_k, x_k$ of vertices and edges, where $e_i = \{x_{i-1}, x_i\} \in E$ in the undirected case and $e_i = (x_{i-1}, x_i) \in E$ in the directed case. The length of a walk is defined as the number of edges on the walk. The walk is called a *path*, if $e_i \neq e_j$ for $i \neq j$, and a path is a *simple path* if $x_i \neq x_j$ for $i \neq j$. A path with $x_0 = x_k$ is a *cycle*. A cycle is a *simple cycle* if $x_i \neq x_j$ for $0 \leq i < j \leq k - 1$.

2.1.2 Constraint Programming

Constraint programming is a *declarative* framework, aimed at solving NP-Complete combinatorial problems. The general idea is that the user

states the problem, and the solver produces a solution. Stating the problem means choosing the decision variables, their initial set of possible values, and the constraints that a solution should respect. Constraints can be viewed as a list of assignments of the variables, but generally constraints are high-level building blocks that the user can combine. The problem model is then given to a constraint solver which should automatically find an assignment of the variables that satisfies the constraints.

Constraint programming is a rich framework, with more than two decades of research. We only present here definitions useful for the rest of the dissertation. The interested reader may consult the Handbook of Constraint Programming [RvBW06].

CSP A *constraint satisfaction problem (CSP)* is a triple $(X, \mathcal{D}, C)$. Let X be a set of *variable*=$\{x_1, \ldots, x_n\}$. Each variable is associated with a *domain* $D(x_i) \subseteq \mathcal{U}$ where $\mathcal{U}$ is the universe and $D(x_i)$ represents the set of possible values for x_i. The set of domains associated with X is $\mathcal{D}$. A *constraint* c_i is defined on a subset of variables $x_1, \ldots, x_k$, denoted *scope(c)*, and $c_i \subseteq \mathcal{U}^k$. A constraint c *constraints* a variable x if $x \in scope(c)$. A *binary constraint* is a constraint c with $|scope(c)| = 2$. A *global constraint* is a constraint c whose $|scope(c)|$ can be arbitrary large. A variable with a singleton domain is *assigned*. A constraint c is *entailed* by $\mathcal{D}$ if $\times_{x_i \in scope(c)} D(x_i) \subseteq c$ (i.e. the constraint holds for all the values in the domain). A constraint is *satisfied* by $\mathcal{D}$ if all variables in its scope are assigned and $\times_{x_i \in scope(c)} D(x_i) \subseteq c$. The set of constraints is denoted C. A *solution* of a CSP is an assignment of all variables such that all constraints are satisfied. The *set of solutions* of a CSP P is noted $Sol(P)$. Finding $Sol(P)$ is NP-Complete in the general case. A CSP is *failed* if $Sol(P) = \emptyset$. Two CSPs P_1 and P_2 are equivalent if $Sol(P_1) = Sol(P_2)$.

Finite and finite set domains The notion of domain defined so far is quite abstract. Actual constraint programming systems act on concrete types, called computation domains, for example integers, sets or reals. A computation domain is based on a domain abstraction, that is a practical model to represent the domain, together with its set of basic operations. A fundamental computation domain is *integers*, also called *finite domains*. Integers are fundamental because they can be used to model other computation domains like booleans or sets (and

Table 2.1: Operation complexities of a range sequence of r ranges and a bit vector bounded by v.

Operations	Range sequence	Bitvector
$x.getmin()$	O(1)	O(1)
$x.getmax()$	O(1)	O(1)
$x.hasval(d)$	O(r)	O(1)
$x.adjmin(d)$	O(r)	O(1)
$x.adjmax(d)$	O(r)	O(1)
$x.excval(d)$	O(r)	O(v)
$i.done()$	O(1)	O(v)
$i.value()$	O(1)	O(1)
$i.next()$	O(1)	O(v)

any other discrete domains). Popular representations of finite domains are range sequences and bit vectors. A *range sequence* for a finite set of integers I is the shortest sequence $s = \{[n_1, m_1], \ldots, [n_k, m_k]\}$ such that I is covered ($I = \cup_{i=1}^{k}[n_i, m_i]$) and the ranges are ordered by their smallest elements ($n_i \leq n_{i+1}$ for $1 \leq i < k$). A range is unique, none of its ranges are empty and $m_i + 1 < n_{i+1}$ for $1 \leq i < k$. A *bit vector* for a finite set of integers of I is a string of bits such that the i^{th} bit is 1 iff $i \in I$. Basic operations for finite domain and their operations are summarized in Table 2.1.2 (taken from [SC06]). In this table, x is a variable, d a value, and i is an iterator.

Another important computation domain are sets. The most commonly used domain abstraction for sets are *set intervals*. Set intervals are intervals of the partial order defined by set inclusion. Set intervals are defined by two bounds, the greatest lower bound (glb), and the least upper bound (lub). The glb represents the set of values that must be in the set variable. The lub represents the set of values that can belong to the set variable. More formally, a set interval $[s_L, s_U]$ defines the following (possibly empty) set of sets: $\{s \mid s_L \subseteq d \subseteq s_U\}$. Set intervals define a domain abstraction as they allow to represent single sets ($[s, s]$) and are closed under intersection: $[s_L^1, s_U^1] \cap [s_L^2, s_U^2] = [s_L^1 \cup s_L^2, s_U^1 \cap s_U^2]$. Basic operations include intersection, union, difference, cardinality and membership and can be defined through filtering rules. See [Ger06] for a detailed explanation of those rules, and an overview of other domain

abstractions for sets.

Filtering and propagators A simple algorithm exists to find a solution to a CSP. A straightforward idea is to enumerate all assignments of the variables with respect to their domains, and check if the constraints are satisfied. This strategy is correct, but the time to find a solution is exponential. Individual constraints can be used to remove values.

Values in the domain of a CSP may violate some constraints of the CSP. *Filtering* removes values in the domain of $\mathcal{D}$ that have no support in a constraint c. A value d in $D(x)$ has a *support* in c with respect to $\mathcal{D}$ if there exists an assignment A of the variables in $\mathcal{D}$ with $x = d$ and $A \subseteq c$. The value d is also said to be *consistent* with the constraint c. A *propagator* is the implementation of a constraint that performs filtering. A propagator takes a domain $\mathcal{D}$ as input and returns a new domain where inconsistent values with respect to c are removed.

Comparing the output of a propagator with its constraint calls for notations. A constraint c is a set of tuples indexed by variables, and can be viewed as a domain of a CSP. Set operations can be extended to domains of CSPs. Given a set operation $\diamond$, $\mathcal{D}_1 \diamond \mathcal{D}_2$ means $D_1(x_i) \diamond D_2(x_i)$ for all $x_i \in X$. With these notations, properties of a propagator can be stated.

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two finite domains defined on the same set of vars. A propagator p_c implementing a constraint c must respect the following conditions :

- contractant: $p_c(\mathcal{D}_1) \subseteq \mathcal{D}_1$
- monotone: $\mathcal{D}_1 \subseteq \mathcal{D}_2 \Rightarrow p_c(\mathcal{D}_1) \subseteq p_c(\mathcal{D}_2)$
- correct: $Sol(X, \mathcal{D}, \{c\}) \subseteq Sol(X, p_c(\mathcal{D}), \{c\})$

Successive applications of a propagator may narrow domains, that is $p(p(\mathcal{D})) \subseteq p(\mathcal{D})$. A propagator p is said to be *idempotent* if $p(p(\mathcal{D})) = p(\mathcal{D})$. This is important in the filtering process, as idempotent propagators need to be applied only once.

Levels of consistency Propagators should remove inconsistent values with respect to a constraint. Doing so can however be NP-Complete. Moreover a polynomial optimal filtering algorithm is not always useful. This is the case, for example, when the density of solutions is high.

Removing values may be expensive, while enumeration leads quickly to a solution.

This motivates the introduction of levels of consistency. The filtering performed by a propagator p for a constraint c may vary. We enumerate here a number of definitions of consistency, from the cheaper to the more expensive in time complexity, but with increasing filtering power.

- checking: a propagator p is *checking* for a constraint c if $(\forall\ x \in scope(c) : |D(x)| = 1) \Rightarrow \times_{x_i \in X} D(x_i) \subseteq c$. Intuitively, the propagator waits for the instantiation of *all* the variables in its scope, and check if the constraint c is verified for this assignment.

- forward checking: a propagator is *forward checking* for a constraint c if all variables in $scope(c)$ are instantiated but the variable x_i and $\forall\ v_i \in D(x_i) : (v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k) \in c$.

- (hyper) arc consistency: A propagator is *hyper arc consistent* with respect to a constraint c with $scope(c) = x_1, \ldots, x_k$ if $\forall\ 1 \leq i \leq k : \forall\ v_i \in D_i,\ \exists\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k \in D_1 \times \cdots \times D_k$ such that $(v_1, \ldots, v_k) \in c$. For a binary constraint, it is called *arc consistency*, and for a unary constraint, *node consistency*.

- (hyper) bound consistency: The idea behind bound consistency is to consider the bounds of the domains. In total ordered finite domains, there exits a $minD(x)$ and a $maxD(x)$ elements, and we may prune only on those bounds. This idea leads to several definitions of bound consistency.

  - bound consistency: A propagator is *bound consistent* with respect to a constraint c with $scope(c) = x_1, \ldots, x_k$ if $\forall\ 1 \leq i \leq k : \exists\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k \in [minD_1, maxD_1] \times \cdots \times [minD_k, maxD_k]$ such that $(v_1, \ldots, v_{i-1}, minD_i, v_{i+1}, \ldots, v_k) \in c$ and $(v_1, \ldots, v_{i-1}, maxD_i, v_{i+1}, \ldots, v_k) \in c$.
  - range bound consistency: A propagator is *range bound consistent* with respect to a constraint c with $scope(c) = x_1, \ldots, x_k$ if $\forall\ 1 \leq i \leq k\ \forall\ v_i \in [minD_i, maxD_i]$ $\exists\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k \in [minD_1, maxD_1] \times \cdots \times [minD_k, maxD_k]$ such that $(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k) \in c$.
  - domain bound consistent: A propagator is *domain bound consistent* with respect to a constraint c with $scope(c) =$

$x_1, \ldots, x_k$ if $\forall\ 1 \leq i \leq k\ \exists\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k \in D_1 \times \cdots \times D_k$ such that $(v_1, \ldots, v_{i-1}, minD_i, v_{i+1}, \ldots, v_k) \in c$ and $(v_1, \ldots, v_{i-1}, maxD_i, v_{i+1}, \ldots, v_k) \in c$.

Bound consistency is weaker than range and domain bound consistency, which are incomparable.

Instead of performing consistency over a single (binary) constraint, a set of constraints can be considered for consistency. Let $c_{i,j} \in C$ denote a binary constraint between variables x_i and x_j.

- Two variables x_i and x_j are path consistent if for any pair of values $(d_i, d_j) \in D(x_i) \times D(x_j)$ and for any sequence of variables $Y = (x_i = x_{k_1}, x_{k_2}, \ldots, x_{k_p} = x_j)$ such that for all $q \in [1, p-1]$: $c_{k_q,k_{q+1}} \in C$, there exits a tuple of values $(d_i = d_{k_1}, \ldots, d_{k_p} = d_j)$ such that for all $q \in [1, p-1]$, $(v_{k_q}, v_{k_{q+1}}) \in c_{k_q,k_{q+1}}$.

Consistency of a CSP Many consistency levels have been defined. For an overview, see [Bes06]. We have so far discussed about the consistency of individual constraints. We now figure out what does it mean for the CSP itself to be consistent.

The local consistency of the constraints can be combined to reach a fixpoint of the propagation of the individual constraints. If a constraint c is made consistent by a propagator p, then other constraints (including c if p_c is not idempotent) may not be consistent anymore, since domains were reduced. All the propagators are ran in turn until they remove no values. The *propagation of a CSP* is the fixed point of all propagators associated to the current CSP. When each of its constraint is hyper arc consistent, we say that a CSP is *globally arc consistent (GAC)*. Algorithm 1 gives the pseudo code of such a fixpoint propagation algorithm. It is the building block of most constraint solvers. Note that Algorithm 1 exits once a domain is empty, instead of going to the fixpoint.

Search Propagation does not solve a CSP. Propagation must therefore be associated with a search component. After propagation, non-deterministic choices can be computed on the current CSP. This is usually done by adding a constraint such as $x = d$ or $x \neq d$, with $x \in X$, $d \in D(x)$ and $|D(x)| > 1$. The propagation can then be performed again. Interleaving propagation and non-deterministic choices creates a search tree, and the process is called *search*. Algorithm 2 implements

Algorithm 1: Propagation Algorithm

Input: A CSP $P = (X, \mathcal{D}, C)$ and a set P of propagators p_c for each $c \in C$

Output: $\mathcal{D}$ is filtered to the propagation fixpoint of P.

```
1 procedure propagate(X, D, C)
2 W ← P
3 while W ≠ ∅ ∧ ∀ x ∈ X : D(x) ≠ ∅ do
4     pop p_c from W
5     D' ← D
6     D ← p_c(D)
7     if D ≠ D' then
8         W ← W ∪ {p_c' | ∃ x ∈ scope(c') ∧ D(x) ≠ D'(x)}
9     if p_c is idempotent then W ← W \ {p_c}
```

search. Some mechanisms in this algorithm are actually generic and can be specialized in a constraint programming framework. The actual computed choices are defined by a *heuristics* (line 6). The way choices are selected is called *exploration* (line 7). The actual application of the choice is called *branching* (line 9). Branching and exploration are two independent issues: the heuristics determine the search tree, while the exploration determine how the search tree is traversed (for example by depth first search or by breadth first search). When branching, state has to be saved and restored. *State restoration* is a common issue in constraint programming engines.

Heuristics Heuristics determine the search tree. It also determine the position of the solutions and the non solutions in the leaves of the search tree. Heuristics are important because they may reduce the depth of the search tree, but also determine the position of the first solution in the leaves, which is important if the leaves are explored in a fixed order. From a practical point of view, heuristics have a strong influence on the performance for a particular problem.

Heuristics can take various forms. The most common heuristics is to select a non assigned variable x and a value d. The choices are then $x = d$ and $x \neq d$ which ensure the whole space is traversed. This is called a *variable value heuristics*. However, other forms of heuristics exist. For example, one may split domains in two equal range, leading

Algorithm 2: Search Algorithm

Input: A CSP $P = (X, \mathcal{D}, C)$ and a set of propagators p_c for each $c \in C$
Output: $Sol(P)$

```
1  procedure search(X, D, C)
2  S ← ∅
3  propagate(X, D, C)
4  if ∃ x ∈ X : D(x) = ∅ then return ∅.
5  if ∀ x ∈ X : |D(x)| = 1 then return D.
6  branchings ← < compute choices for (X, D, C) >
7  for next choice in branchings do
8      save state (X, D, C)
9      apply choice to (X, D, C)
10     S ← S ∪ search(X, D, C)
11     restore state (X, D, C)
12 return S
```

to a constraint of the form $x = [minD(x), (maxD(x) - minD(x))/2]$ or $x = [((maxD(x) - minD(x))/2) + 1, maxD(x)]$.

State restoration State restoration can be done in two ways. The first method is called *trailing*. Trailing stores the undo information. The memory location and the content of the entity to be saved is stored on a trail. Each time an exploration of a new node in the search tree is performed, a mark is put on the trail. Restoring state amounts to copying the entities from the trail and erase the previous mark. In the context of a constraint programming solver, a timestamp is also added with the entity, in order to ensure that even if the domains are narrowed multiple times, the domains are saved only once. The timestamp changes each time a mark is put on the trail.

Copying is a second method. The current state, that is the domains together with the propagators and their data structures, are simply copied and stored. All entities are extended with a copy method and copying a state amounts to recursively copy entities. Restoring a state amounts to remove and use a stored state.

Trailing aims at saving memory, while copying may seem an aggressive strategy regarding memory usage. Copying has one main advantage: it is more expressive than trailing. In trailing, the system is forced to

backtrack to the previous state, while in copying one can select a state in the list of stored states. Moreover, parallelism is straightforward with copying: one has to distribute stored states.

In order to trade space for time, copying usually offers *recomputation*. Recomputation saves a state in the search from times to times. When a backtrack is needed the last state saved on the path to the root is used. The intermediates branching decisions are stored, and using the last state, the branching decision constraints are added to the CSP and the propagation fixpoint is computed.

Model Given a problem, the programmer has to formulate the problem in the constraint programming framework. A *(CSP) model* consists of the following elements : the choice of the variables, the choice of the domains, the choice of the set of constraints, and for each constraint, the level of consistency; moreover, it may be critical to choose a heuristics and to define the exploration, for example depth first search. Depending on the actual solver, there may be some other fine tunings, like the use of recomputation and the distance between two saved states.

Constraint Programming Systems Constraint logic programming, that is logic programming augmented with syntactic constraints was first developped in Prolog [CKC82] with disequations. Finite domains [Hen89] were introduced in the CHIP [DvH87] constraint logic programming system. Other Prolog systems such as GNU Prolog [DC01] and ECLIPSE [WNS97] were then developped. OPL [Hen02] is the first modeling language to combine high-level algebraic and set notations which are then translated into a constraint program. The underlying constraint solver is ILOG Solver [Pug94].

Constraint Solvers such as ILOG Solver provide a constraint programming API for procedural languages for C and C++. The language Oz [Smo95, RH04] and its open-source implementation Mozart provide a multiparadigm framework in which, among others, the functional, concurrent and logic programming paradigms can be combined with the concept of first-class computation spaces to implement a constraint programming framework [Sch02]. The Gecode (http://www.gecode.org) constraint developement environnement is an open-source C++ library. Its architecture is described in [SS04a, ST06]. We use the Gecode system to implement our constraint programming approach of graph matching.

2.2 Comparing graphs

Basic measures are not sufficient for graph comparison. Two graphs may have the same number of nodes and edges and the same average degree, but still be different. Given two graphs, $G_p = (V_p, E_p)$, called the *pattern graph*, and $G_t = (V_t, E_t)$, called the *target graph*, one may ask how they relate to each other, *regarding their structure*. Many relationships between G_p and G_t can occur regarding their structure. The graphs G_p and G_t may be identicals. The graph G_p may be identical to a subgraph of G_t. There may exist a subgraph of G_p of maximal size that is identical to G_t. There may exist a subgraph of G_p and a subgraph of G_t that are identical. The graphs G_p and G_t may be identical if some operations are performed on G_p and G_t, like removing a node or an edge. Those relationships can be captured through the concept of matching, which associates the nodes of the pattern with the nodes of the target.

2.2.1 Matching

Matching two graphs $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$ consists in relating their vertex sets V_p and V_t and their edge sets E_p and E_t. A general way is to associate vertices through relations.

Relation and function A *relation* R is an ordered triple (X, Y, H) where X and Y are arbitrary sets, and H is a subset of the Cartesian product $X \times Y$. The sets X and Y are called the domain and codomain, respectively, of the relation. The statement $(x, y) \in R$ is read "x is R-related to y", and is denoted by xRy or $R(x, y)$. The latter notation corresponds to viewing R as the characteristic function of the set of pairs H. The order of the elements in each pair of H is important: if $a \neq b$, then aRb and bRa can be true or false, independently of each other. A function F is a relation (X, Y, H) with the restriction that F pairs each $x \in X$ with at most one $y \in Y$. The relational notation xFy is usually written $F(x) = y$ in the case of a function.

Definition. (**Matching**) A matching between graphs $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$ is a relation (V_p, V_t, F).

For labeled graphs, labeling compatibility between nodes and edges may be required. Each node and edge is associated with an integer through a function α and β respectively. We suppose that there exits compatibility functions for the nodes and the edges.

Definition. (**Compatibility functions**) A node compatibility function C_n and an edge compatibility function C_e are functions of the type $\mathbb{N} \times \mathbb{N} \rightarrow \{true, false\}$.

Definition. (**Labeled matching**) Given two labeled graphs $G_p = (V_p, E_p, \alpha_p, \beta_p)$, $G_t = (V_t, E_t, \alpha_t, \beta_t)$, a matching F is labeled matching between G_p and G_t if $\forall\ v_1 \in V_p\ \forall\ v_2 \in V_t : v_1 F v_2 \Rightarrow C_n(\alpha_p(v_1), \alpha_t(v_2))$ and $\forall\ (v_1, v_2) \in E_p\ \forall\ (v_2, v_4) \in E_t : v_1 F v_3 \wedge v_2 F v_4 \Rightarrow C_e(\beta_p(v_1, v_2), \beta_t(v_3, v_4))$.

Matchings are not equally desirable. Interesting matchings should preserve some structure of G_p onto G_t.

2.2.2 Morphisms

An interesting class of matchings are *morphisms*. This class imposes unary relationships between the nodes of the pattern graph and the nodes of the target graph, that is the relation is a function. The specifity of a morphism is that a morphism always preserves the structure of G_p onto G_t.

Definition. (**(Homo)morphism**) Let $\square$ be a binary operation on a set X, while $\square'$ is another binary operation on a set X'. A *morphism* $f : (X, \square) \rightarrow (X', \square')$ is defined to be a function on X to X' which "carries" the operation $\square$ on X onto the operation $\square'$ on X', in the sense that $f(x \square y) = f(x) \square' f(y)$. A morphism is also called a *homomorphism*.

In graph matching, the $\square$ and $\square'$ operators are the neighborhood relation between vertices. By constraining the function f, new types of morphism are defined:

1. *isomorphism*, if f is a bijection,
2. *epimorphism*, if f is a surjection,
3. *monomorphism*, if f is an injection.

Two graphs $G_p = (V_p, E_p, \alpha_p, \beta_p)$ and $G_t = (V_t, E_t, \alpha_t, \beta_t)$ are graph homomorphic if the edges of G_p can be conserved into G_t by mapping the nodes of G_p onto the nodes of G_t.

Definition. (**Graph homomorphism**) There exists a *graph homomorphism* between G_p and G_t if there exists a labeled matching f such that $(a, b) \in E_p \Rightarrow (f(a), f(b)) \in E_t$.

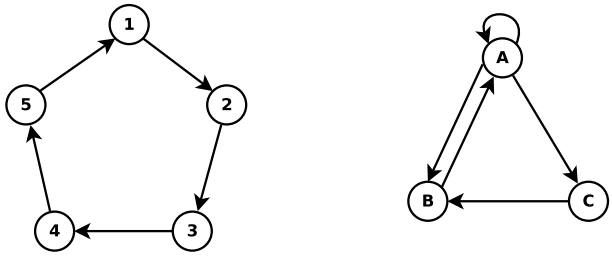

Figure 2.3: Example of graph homomorphism with $f = \{(1, A), (2, B), (3, A), (4, B), (5, A)$.

Graph homomorphism is illustrated in Figure 2.3.

Two graphs $G_p = (V_p, E_p, \alpha_p, \beta_p)$ and $G_t = (V_t, E_t, \alpha_t, \beta_t)$ are graph isomorphic if they have the same number of nodes and the same structure.

Definition. (**Graph isomorphism**) There exists a *graph isomorphism* between G_p and G_t if there exists a labeled matching f such that f if a bijective function and $(a, b) \in E_p \Leftrightarrow (f(a), f(b)) \in E_t$. Graph isomorphism is illustrated in Figure 2.4.

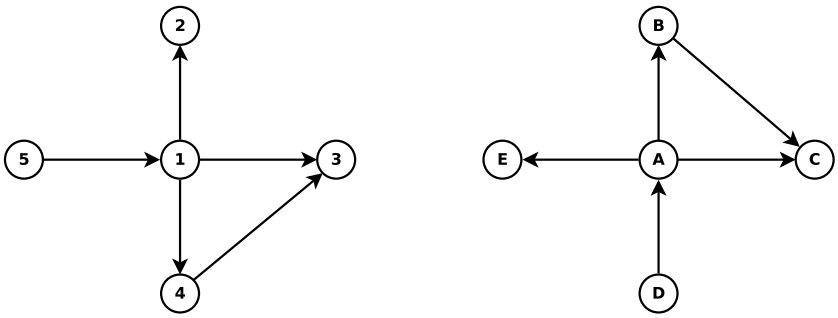

Figure 2.4: Example of graph isomorphism with $f = \{(1, A), (2, E), (5, D), (4.B), (3, C)\}$.

Two graphs $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$ are graph epimorphic if G_p can be contracted (by merging nodes) to match the structure of G_t. Graph epimorphism is illustrated in Figure 2.5.

Definition. (**Graph epimorphism**) There exists a *graph epimorphism* between G_p and G_t if there exists a labeled matching f such that f if a surjective function and $(a, b) \in E_p \Leftrightarrow (f(a), f(b)) \in E_t$.

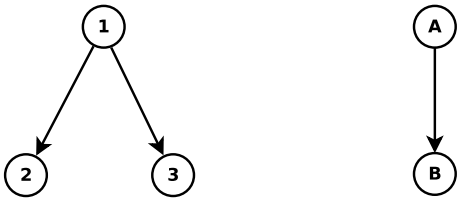

Figure 2.5: Example of graph epimorphism with $f = \{(1, A), (2, B), (3, B)\}$.

Graph monomorphism looks for a subgraph of G_t that is isomorphic to G_p. Hence graph monomorphism is also known as subgraph isomorphism. Subgraph isomorphism is usually splitted in two definitions corresponding to the two definitions of a subgraph. Partial subgraph isomorphism is a classical graph monomorphism that imposes that each edge of G_p is mapped to an edge of G_t. Induced subgraph isomorphism adds the condition that non adjacent vertices of G_p match to non adjacent nodes of G_t.

Definition. (**Graph monomorphism or partial subgraph isomorphism**) There exists a *partial subgraph isomorphism* between G_p and G_t if there exists a labeled matching f such that f is a total injective function and $(a, b) \in E_p \Rightarrow (f(a), f(b)) \in E_t$.

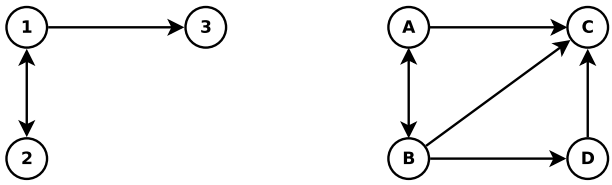

Figure 2.6: Example of partial subgraph isomorphism with $f = \{(1, A), (2, B), (3, C)\}$. There is no induced subgraph isomorphism, because the induced subgraph A, B, C has one more edge.

Partial subgraph isomorphism is illustrated in Figure 2.6.

Definition. (**Induced subgraph isomorphism**) There exists an *induced subgraph isomorphism* between G_p and G_t if there exists a labeled matching f such that f is a total injective function and $(a, b) \in E_p \Leftrightarrow$

$(f(a), f(b)) \in E_t$. Induced subgraph isomorphism is illustrated in Figure 2.7.

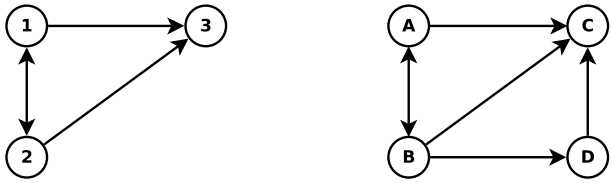

Figure 2.7: Example of induced subgraph isomorphism with $f = \{(1, A), (2, B), (3, C)\}$.

Suppose there is no graph or subgraph isomorphism between G_p and G_t. We may still found however two subgraphs of G_p and G_t that are isomorphic. Matching a subgraph of G_p can be done by forcing the function f to be partial.

Definition. (**Common subgraph**) A *common subgraph* is a subgraph G of G_p such that G is isomorph to a subgraph of G_t.

Definition. (**Maximum common subgraph**) A *maximum common subgraph* between G_p and G_t is a common subgraph G between G_p and G_t of maximal size (number of nodes).

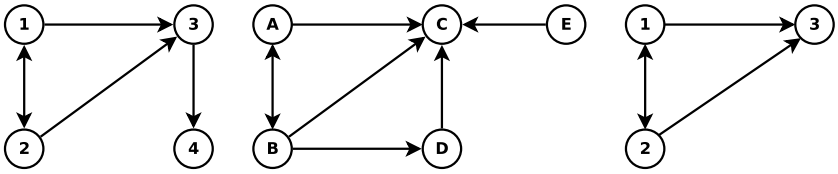

Figure 2.8: Example of maximum common induced subgraph with $f = \{(1, A), (2, B), (3, C)\}$. The resulting common subgraph is shown on the right.

Because the subgraph isomorphism can be induced or partial, we define accordingly the maximum *partial* and *induced* common subgraph problems. Maximum common induced subgraph is illustrated in Figure 2.8.

2.2.3 Graph Similarity

Graph similarity measures the distance between two graphs G_p and G_t. While graph morphism empaphises on preserving the structure (that is the neighboorhood relationship), graph similarity measures a distance. Graph similarity is thus more general than graph morphism. Graph morphism is a possible criteria for graph similarity. Moreover, graph morphism is based on the notion of function, while graph similarity has no constraint on the existence and type of matching. Graph similarity is thus based on relations, and is often required to be a distance, that is a measure reflexive, symmetric, and that respects the triangle inequality.

We give here two examples of graph similarity. The first one, called graph edit distance, is a classic graph similarity measure. The second one is a generic graph similarity measure, able to express many graph similarities.

Given a set of allowed operations on graphs associated with costs, the *graph edit distance* is the minimal cost of a sequence of operations to transform G_p into a new graph G isomorphic to G_t. There is no general agreement on the set of operations and associated costs. A basic set of operations consists of the following set of 4 operations:

- vertex insertion: a new isolated vertex is added to the graph
- vertex deletion: an isolated vertex is deleted from the graph
- edge insertion: a new edge is added between arbitrary vertices of the graph
- edge deletion: an edge is deleted from the graph.

The cost of all vertex operations are one, and all edge operations cost zero. The distance between the two graphs is then equal to the number of $|n_p - n_t|$.

More recently, Solnon *et al.* [SSJ07] have proposed a generic graph distance measure, that is a unifying framework for all distance measures. This similarity measure considers a relation m. The similarity of two graphs G_p and G_t is defined as a triple $< \sigma_{vertex}, \sigma_{edge}, \bigotimes >$ where σ_{vertex} is the vertex distance function, σ_{edge} is the edge distance function, and $\bigotimes$ is the function used to aggregate those distances. Those three functions are defined with respect to a specific application domain. The vertex distance function σ_{vertex} returns the distance between a node of

one graph and a set of nodes of the other graph. The edge distance function σ_{edge} returns the distance between an edge of one graph and a set of edges of the other graph. The aggregate function $\bigotimes$ returns the measure by using σ_{vertex}, σ_{edges}, and m.

Such a formulation can express all graph morphism problems together with many published graph similarities in a very elegant way.

2.3 Computing Matchings

Most graph morphism problems presented are at least NP-Complete.

Graph homomorphism is the most general graph morphism problem and is NP-Complete. Computing graph homomorphism between two graphs is done by representing a function and maintaining the neighborhood relationship. Rudolf [Rud98a] proposes a CSP modeling where the function is represented as a vector of finite domain variable and constraints enforce the neighborhood relationship. Valiente *et al.* [VM97] develop a dedicated algorithm. The function is represented as a boolean matrix of size $n_p * n_t$. All matrices are enumerated in a search tree and leaves are tested for graph homomorphism. Pruning operators are used during search to enforce the neighboorhood condition.

The graph isomorphism problem is not known to be nor NP-Complete, nor in P. Although there is no polynomial time algorithm, most of the instances can be solved efficiently. This had lead researchers to define an intermediate class of complexity, for problems that can be reduced to graph isomorphism. A problem is said to be *Graph isomorphism complete (GI-complete)* if there exists a polynomial turing machine reduction to graph isomorphism. The following class of graphs are polynomial for graph isomorphism: trees, planar graphs, interval graphs, permutation graphs, partial k-trees, bounded-parameter graphs, graphs of bounded genus, graphs of bounded degree, graphs with bounded eigenvalue multiplicity [BE05]. The state-of-the-art software for graph isomorphism is Nauty [McK81]. An extensive review of this algorithm can be found in [BE05]. A filtering algorithm has also been proposed in the context of constraint programming [SS04b].

Subgraph isomorphism is also tractable for some special classes of graphs, such as trees [Val02], planar graphs [HW74], and bounded valence graphs [Luk80]. A series of theoretical works also look for particular classes of polynomial time algorithm based on bounded tree-width (see for example [HN02]). An extensive review of subgraph isomorphism

exact algorithm can be found in the next section.

There are two main approaches for the maximum common subgraph problem. The first is a backtrack search algorithm where the set of assignments is extended until a largest common subgraph is found. This is the approach used initially by McGregor [McG82]. Similar exact algorithms have been proposed more recently [Öst02]. The vflib library also proposed an algorithm to solve the maximum common subgraph (which is a straightforward extension of the subgraph isomorphism algorithm presented in the next sections). The second approach is based on the fact that a maximum common subgraph of two graphs is equivalent to a maximum clique in the product graph of the pattern and the target. The advantage of this second approach is that various bounds on the clique problem can be used. Many exact algorithms have been proposed for the clique problem. One of them is the well-known algorithm of Bron and Kerbosch [BK73]. The Durand-Parasi algorithm is also based on clique detection [DPBT99]. In constraint programming, J.C. Régin has designed bounds and search strategies for the clique problem [Rég03], and claimed to be state-of-the-art for exact approches, by competing on a set of challenging instances. A recent paper presents an experimental comparison of three well-known algorithms [DCV07].

More recently, a class of graph matching was defined, where labels are unique in graphs [DBDK04]. For this class, polynomial time algorithm for all graph matching problems have been found, including maximum common subgraph and graph edit distance.

The complexity of graph similarity depends on the definition of the distance, and is usually at least NP-complete. Because graph similarity is usually harder than graph morphism problems, graph similarity is computed by local search. For instance, the approach from [SSJ07] defines the neighborhood by adding or removing a couple (i, j) with $i \in N_p$ and $j \in N_t$ to the relation m. A greedy search is first performed to reach an initial (good) solution [CS03a]. Reactive tabu search [SS05] and ant colony optimization [SSSG06] are then used to improve the locally optimum greedy solution. It is worth noting that the authors from [SSJ07] claim that their algorithms are not competitive for exact graph isomorphism and subgraph isomorphism, as dedicated algorithms are able to use filtering techniques whereas their algorithm explores potentially all mappings.

The next section focuses on dedicated algorithms for exact subgraph isomorphism.

2.4 Subgraph Isomorphism Algorithms

In this section two well-known dedicated algorithms for subgraph isomorphism are presented in details. We call *dedicated algorithms* algorithms that were designed specifically for a problem, in order to distinguish those algorithms from a constraint programming approach.

Dedicated algorithms usually implement a backtracking algorithm. They differ in the way they represent the matching function and how they ensure that the condition of morphism and injection are respected. Instead of presenting each algorithm in details, we present two well-known algorithms that illustrate the principles used by other algorithms.

2.4.1 Ullman algorithm

The Ullmann algorithm is based on a matrix view of the SIP. Let $P = [p_{i,j}]$ for $1 \leq i, j \leq n_p$ and $T = [t_{i,j}]$ for $1 \leq i, j \leq n_t$ be the adjacency matrices for G_p and G_t. Let M be a binary matrix with n_p rows and n_t columns, so that a row of M contains exactly one 1 and no column contains more than one 1. The matrix M represents an injective function from N_p to N_t. Ullmann notices that M can be applied to permute rows and columns of T:

$$C = [c_{i,j}] = M(MT)^T$$

where exponent T denotes transposition. If C is equal to P:

$$c_{i,j} = p_{i,j} \quad \forall\, i, j \in [1, n_p] \tag{1}$$

then M is an monomorphism function from G_p to G_t.

The basic Ullmann algorithm is an enumeration algorithm. All possible matrices representing a monomorphism are enumerated and then checked against condition (1). Ullman also proposes a refinement procedure of the matrix M that prunes the search space of the enumeration. We start by describing a preprocessing step of the matrix M. We detail the enumeration algorithm and then present its extension using the refinement procedure. The algorithm preprocesses M by forcing some $m_{i,j}$ to 0 by comparing the the degrees of the pattern and target vertices.

Degree condition Two vertices a from G_p and b from G_t can be mapped through a morphism iff $d(a) \leq d(b)$.We restrict the description

for undirected graphs, but the method can be extended to the directed case.

The degree condition defines an initial matrix M^0:

$$M^0 = m^0_{i,j} = \begin{cases} 1 & \text{if } d(i) \leq d(j) \\ 0 & \text{otherwise.} \end{cases}$$

Starting from M^0, the Ullmann algorithm performs a backtrack search by generating all possible injective functions M such that $m_{i,j} = 1 \Rightarrow m^0_{i,j} = 1$ To perform backtracking, the algorithm uses several data structures:

- a variable d denoting the current depth in the search tree
- a variable k denoting the last column selected for the current line
- a vector $F =< F_1, \ldots, F_{n_p} >$ where $F_i = 1 \Leftrightarrow i$th column has been selected
- a vector $H =< H_1, \ldots, H_{n_p} >$ where $H_i = j \Leftrightarrow j$th column has been selected at depth i in the search tree.
- a vector $M_v =< M_1, \ldots, M_{n_p} >$ where M_i is the last matrix generated at depth i.
- a matrix M representing the current matrix.

The vectors H, M_v and the variable d are used for the backtracking. The vector F and the variable k are used to ensure the injective condition.

Algorithm 3 gives the pseudo-code. The enumeration algorithm is read by including the text between $< \cdots >$, and the refinement extension is read by including the text between $[\ldots]$. Line 1 initializes the current matrix M with M^0, sets current depth level at 1, indicates that no column has been selected at depth 1 and that no column has been selected in the current line. Line 2 indicates that no column has been selected yet. Line 3 enters a loop that browses the search tree. The while loop exits when depth 0 is reached, that is when all 1 in row 1 of M have been searched. Line 4 looks for a column k in row d which has not been selected (we ignore the `refine` procedure for now). If such a k is found, then all other 1 in row d are set to zero in matrix M in line 5, otherwise the algorithm backtracks. If a final state is reached

Algorithm 3: Ullmann Algorithm

Input: Two adjacency matrices P and T, an initial matrix M^0
Output: *fail* or a $|N_p| * |N_t|$ matrix M, with M representing a subgraph isomorphism function.

```
1  M ← M^0; d ← 1; H_1 ← 0; k ← 0; backtrack ← true;
2  for i in [1, |N_p|] do F_i ← 0;
3  while d ≠ 0 do
4      if [ refine(M, P, T) ∧ ] (∃ k : m_{d,k} = 1 ∧ F_k = 0) then
5          ∀ j ≠ k : m_{d,j} ← 0
6          if (d = |N_p| < ∧ condition (1) is satisfied > ) then return
           M
7          backtrack ← false
8      if backtrack then
9          F_k ← 0; d ← d − 1;
10         if d > 0 then M ← M_d; k ← H_d;
11     else
12         H_d ← k; F_k ← 1; M_d ← M; d ← d + 1;
13 return fail
```

and condition (1) is true then the current monomorphism M is returned (line 6). Line 7 prevents backtracking as a column k has been selected in a non final state. Line 8 to line 12 handle the search. If backtracking was selected, then the current column k is deselected, depth is diminished by 1, and the matrix M of previous depth is restored together with the column k selected at previous step (line 9 and line 10). Otherwise, the current state is saved (line 12). Finally, if the while loop exits, no monomorphism was found and $fail$ is returned (line 14).

The refinement procedure change some $m_{i,j} = 1$ to 0, by observing that a pattern vertex can be mapped to a target vertex if and only if their respective neighbors can also be mapped.

Neighbor condition An entry of $m_{i,j}$ is equal to 1 iff

$$\forall\, x \in [1, n_p] : p_{i,x} = 1 \Rightarrow \exists\, y \in [1, n_t] : m_{x,y}.t_{j,y} = 1. \tag{2}$$

For each entry $m_{i,j} = 1$, the neighbor condition can be tested. If it is false, $m_{i,j}$ can be changed to 0 as $m_{i,j} = 0$ for any monomorphism

Algorithm 4: Refinement Algorithm

Input: a matrix M and two adjacency matrices P and T
Output: *true* if M was refined,
false if a row contains no 1.

```
1 procedure refine(M, P, T)
2 fixpoint ← true;
3 repeat
4     for each (i, j) s.t. m_{i,j} = 1 do
5         if Condition (2) is not satisfied then
6             m_{i,j} ← 0; fixpoint ← false;
7             if ∀ k : m_{i,k} = 0 then return false;
8 until fixpoint ;
9 return true
```

derived from M. The modification of one $m_{i,j}$ can modify the neighbor condition for the other entries of M. The neighbor condition is thus tested until a fixpoint with respect to M is reached.

If a matrix M' is an injective function from N_p to N_t, a necessary and sufficient condition is that `refine`(M', P, T) leaves M' unchanged since all neighbor relations are respected. Checking condition 1 can be replaced by the `refine` procedure.

Algorithm 4 gives the pseudo-code of the refinement procedure. Line 3 introduces a repeat loop that exists when the fixpoint is reached (line 8). For each possible matching between pattern vertex i and target vertex j in M (line 4), condition (2) is checked (line 5). If condition (2) is verified, the matching between i and j is pruned, and the fixpoint is not reached (line 6). Moreover, $false$ is returned if the current row i has no 1 (line 7). If a fixpoint is reached, *true* is returned (line 9).

Using the `refine` procedure in the enumeration is straightforward. Algorithm 3 can be read by including the text $< \cdots >$ and excluding $[\ldots]$. Condition (1) is removed, as the refinement is a necessary and sufficient condition if depth is n_t.

Complexity Space complexity for a single state is $O(n_p^2.n_t)$. The vector M_v stores n_p matrices of size $n_p * n_t$. The best case is a direct path of the search starting from the root and ending at a solution node. The worst case is an exploration of all nodes of the search space, which

can contain almost $N!$ states. In the best case and in the worst case, the memory requirement is $\Theta(n_p^2 n_t)$, because the vectors are not copied but only updated. Time complexity of the `refine` procedure is $O(n_p n_t d)$, (where d is the maximum degree of pattern and target graphs). Line 4 of Algorithm 4 iters at most $O(n_p n_t)$ times and condition (2) is $O(d)$ because only the neighboors are checked. The test in line 7 can be easily transformed to a check by using counters and can be considered $O(1)$. In the best case, a linear branch to a solution has a time complexity of $O(n_p^2 n_t d)$. Worst case time complexity is $O(n_p! n_p n_t d)$.

2.4.2 Vflib

The key points of the vflib algorithm are the incremental building of connected graphs with cheap pruning rules and a clever choice of data structures leading to a lower memory footprint. The following presentation is based on [CFSV01] which describes the lastest version of the algorithm. The same type of algorithm is described in [Val02].

While Ullmann enumerates all monomorphisms and prunes some of them, vflib builds incrementally the monomorphism M through a backtracking procedure, stopping whenever the extension M is impossible. Vflib represents the monomorphic function by a subset M of $N_p \times N_t$. A state s of the search is a partial (injective) mapping $M(s) \subseteq N_p \times N_t$, together with G_p and G_t. This $M(s)$ defines two subgraphs, one of G_p and one of G_t, denoted by $M_1(s) = (V_1, E_1)$ and $M_2(s) = (V_2, E_2)$ which are isomorph by construction. The extension of $M(s)$ is done by selecting direct neighbors of those subgraphs. Those two subgraphs are thus constrained to be connected if they are contained in a connected component of their respective graph.

Algorithm 5 gives the pseudo-code. If s is final state, the associated mapping $M(s)$ is returned (line 2). The procedure `genneigh` in line 3 computes the candidate assignments given the state s. For each such a candidate (n, m), the procedure `feasible` checks if assigning n to m violates the monomorphism (line 4 and 5). If vertex n can be assigned to vertex m, the state s is extended to the state s' by adding the assignment (n, m) (line 6). The procedure `vf2` is called recursively on the new state s'. Line 8 handles the backtracking by restoring data structures.

The procedure `genneigh` generates the cartesian product of the neighbors of subgraphs $M_1(s)$ and $M_2(s)$. It considers first the cartesian product of the in-neighbors, then the cartesian product of the out-neighbors. In case those two sets are empty, it generates the cartesian

Algorithm 5: `vflib2` Algorithm

Input: two graphs G_p and G_t, a state s
Output: a total mapping $M(s)$ if a monomorphism exists
fail otherwise.

```
1 procedure vf2(s, G_p, G_t)
2 if M(s) covers all G_p then return M(s);
3 Candidate ← genneigh(s, G_p, G_t);
4 for each (n, m) in Candidate do
5     if feasible(s, n, m) then
6         create a new state s' from s by adding the mapping (n, m);
7         vf2(s', G_p, G_t)
8 restore data structures;
```

product of what is left in the pattern graph and in the target. This corresponds to the case where at least one of the input graphs have several distinct connected components. More formally, one of the following set is generated:

1. $N^+(s) = N^+(M_1(s)) \times N^+(M_2(s))$,
2. $N^-(s) = N^-(M_1(s)) \times N^-(M_2(s))$ if $N^+(s)$ is empty,
3. $N_1 - M_1(s) \times (N_2 - M_2(s))$ if $N^+(s)$ and $N^-(s)$ are empty.

In the actual implementation, the candidates are scanned without an explicit generation.

The procedure `feasible` first checks the necessary morphism condition. The two cases of partial subgraph isomorphism and induced subgraph isomorphism are distinguished. If a partial subgraph is searched, each in-neighbor n' of n in the partial mapping corresponds to a node m' in the in-neighbors of m. If an induced subgraph is searched, the following condition is added: each in-neighbor m' of m in the partial mapping corresponds to a node n' in the in-neighbors of n. More formally, these necessary conditions can be expressed as:

$$\forall\, v \in N_1^-(n) \cap N_1(s) \quad \exists\, v' \in N_2^-(m) \cap N_2(s) : (v, v') \in M(s) \wedge (v, n) \in E_p \wedge (v', m) \in E_t \qquad \text{(partial)}$$

$$\forall\, v' \in N_2^-(m) \cap N_2(s) \quad \exists\, v \in N_1^-(n) \cap N_1(s) : (v, v') \in M(s) \wedge (v, n) \in E_p \wedge (v', m) \in E_t. \quad \text{(induced)}$$

Redundant conditions that prune the search space are also added. The first pruning rule checks if the number of in-neighbors of n that are in the direct in-neighbor of $M_1(s)$ is less or equal to the number of in-neighbors of m that are in the direct in-neighbor (out-neighbor) of $M_2(s)$. The second pruning rule checks if the number of in-neighbors of n are neither in $M_1(s)$ nor in $N(M_1(s))$ is less or equal to the number of in-neighbors of m that are neither in $M_2(s)$ nor in $N(M_2(s))$. More formally, the pruning rules are :

1. $|N_1^-(n) \cap N(M_1(s))| \leq |N_2^-(m) \cap N(M_2(s))|$
2. $|N_1^-(n) \setminus (M_1(s) \cup N(M_1(s)))| \leq |N_2^-(m) \setminus (M_2(s) \cup N(M_2(s)))|$.

The morphism condition together with the pruning conditions are also checked for the out-neighbors. Those additional conditions can be obtained by replacing the in-neighbor functions N_1^- and N_2^- by the out-neighbors functions N_1^+ and N_2^+ respectively, which leads to a total of six conditions.

The morphism condition is illustrated in Figure 2.9. In this figure, $M(s) = \{(1, A), (2, B)\}$ and the `feasible` function is testing the candidate $(3, C)$. The partial morphism condition is verified, since $(1, A) \in M(s)$ and $(1, 3) \in E_p$ and $(A, C) \in E_t$. The out-neighbor vertex 3 of vertex 1 can be associated with the out-neighbor vertex C of vertex A. The induced morphism condition fails, since $(2, B) \in M(s)$ and $(2, 3) \notin G_p$. The vertex C has an in-neighbor B which is matched with the vertex 2, but the vertex 2 is an in-neighbor of vertex 3.

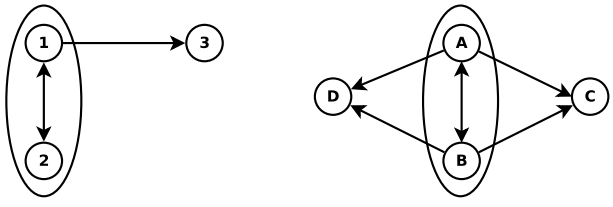

Figure 2.9: Instance where the subgraph is partial and not induced.

The first pruning condition is illustrated in Figure 2.10, where no solutions exist. In this figure, $M(s) = \{(1, A), (2, B)\}$ and the `feasible`

function is testing the candidate $(3, C)$. The morphism condition is verified as the edge $(1, 3)$ can be matched with the edge (A, C). The first pruning rule fails. The set of in-neighbors of the node 3 in the direct neighbor the partial matching is the vertex 4. For the vertex C, there is no such node. The third pruning rule however, is trivially verified, since all nodes in the pattern and target graph are either contained in the partial mapping or in its direct neighbor.

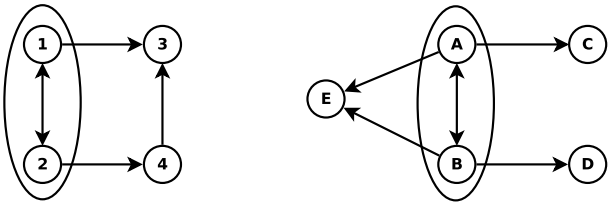

Figure 2.10: Instance where only the first pruning condition fails

The second pruning condition is illustrated in Figure 2.11, where no solution exits. In this figure, $M(s) = \{(1, A), (2, B)\}$ and the `feasible` function is testing the candidate $(3, C)$. The morphism condition is verified, since the edge $(1, 3)$ can be matched with the edge (A, C). The first pruning condition is also verified, since there no neighbors of vertex 3 and C which are in the direct neighbor of the partial mapping. The second pruning condition fails. There are two in-neighbor vertices 4 and 5 of vertex 3 which are outside the partial mapping and its direct neighborhood. There is one in-neighbor vertex E of vertex C which is outside the partial mapping and its direct neighborhood.

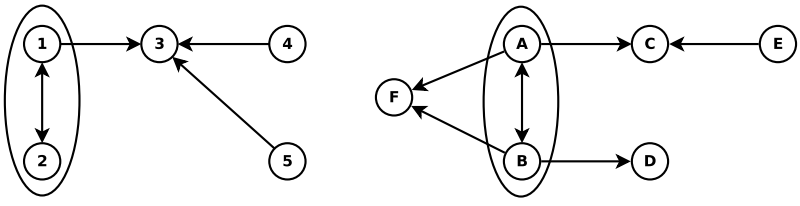

Figure 2.11: Instance where only the second pruning condition fails

Labelling compatibility is also checked by the `feasible` function. If the labels of the vertices n and m or the edges relating n and m to the

current subgraphs are not compatible, the algorithm backtracks.

The injective condition of subgraph isomorphism follows from the generation of candidates in the `genneigh` function. When a couple (n, m) is selected and is feasible, the target node m will not be considered as a candidate until the next backtrack. Hence the morphism is guaranteed to be injective.

Implementation The vflib algorithm uses the following data structures:

- $core_1[n] = m \Leftrightarrow$ vertex n of G_p is matched with vertex m of G_t.
- $core_2[m] = n \Leftrightarrow$ vertex m of G_t is matched with vertex n of G_p.
- $in_1[n] = 1 \Leftrightarrow n \in M_1(s) \vee n \in N^-(M_1(n))$
- $in_2[m] = 1 \Leftrightarrow m \in M_2(s) \vee m \in N^-(M_2(n))$
- out_1 and out_2 are defined similarly.
- d is the current depth of the search ($|M(s)|$).
- $nbrin_1$, $nbrin_2$, $nbrout_1$, $nbrout_2$, are the number of vertices in $N^-(M_1(s))$, etc.
- (n_1, n_2) are the pair of nodes added to s with respect to its direct ancestor.

Complexity Time complexity of `feasible` is proportional to the degree of n and m. Indeed the partial and induced morphism condition can be checked in $O(d_1(n)+d_2(m))$. The second rule can also be checked in $O(d_1(n) + d_2(m))$, since determining the appartenance of a vertex a to $N^-(M_1(s))$ is $O(1)$, thanks to $in_1[a]$. The third rule is also in $O(1)$ since checking if $a \in M_1(s)$ is $O(1)$, thanks to $core_1[a]$. Hence best case time complexity is $O(n_p d)$ and worst case is $O(n_p! d)$.

Memory complexity is a striking feature of vflib. Backtracking can be done without the need of any copy of states. Backtracking from a state s amounts to setting $core_1[n_1]$ and $core_2[n_2]$ to $\perp$. Restoring in_1 for instance can be done in a time proportional to the degree of n_1 and n_2. This ensures that at any depth in the search tree memory requirement is $\Theta(n_t)$.

Comparing Ullman and vflib Both in time and memory requirements, vflib looks more attractive. However, it should be reminded that vflib and Ullman algorithms perform a different level of consistency. As proved by McGregor [McG79], the Ullmann algorithm is equivalent to arc consistency on the morphism condition (see Section 2.5.2 for more details). Vflib performs a forward checking consistency, as matched nodes are used to prune future matchings. So it is not obvious, based on the best and worst case, that the vflib algorithm outperforms the Ullmann algorithm. Its efficiency is confirmed by experimental results on randomly generated graphs [CFSV01]. The main advantage of the vflib algorithm is its low memory footprint and management overhead.

2.4.3 Related Works

There exists a number of subgraph isomorphism algorithms. Ullman is an instance of the early backtrack based algorithms. Many early subgraph isomorphism algorithms were formulated in the context of substructure search in chemical components. Ullmann [Ull76] is one of the first paper and algorithm dedicated to graph and subgraph isomorphism problem itself. Moreover, the Ullmann algorithm is the baseline for more advanced algorithms and it is still cited in recent applications (see for example [GK07]).

Another important algorithm is the vflib algorithm [CFSV01], which is considered as the state-of-the-art for subgraph isomorphism. It extends assignment by growing two connected graphs inside the pattern and the target graph respectively, and by performing the checking of conditions in the neighborhood of the current subgraphs. For example, [Val02] also describes subgraph isomorphism with this approach. The popularity of vflib is due to several papers [CFSV99, CFS$^+$98, CFSV01] demonstrating the efficiency of vflib algorithm against the Ullman algorithm, the public availability of its source code, and the systematic experiments over a synthetic database for graph matching problems [FSV01b].

Some authors have also suggested to use maximum common subgraph algorithms, or the clique algorithms for solving the SIP. Although those approaches were initially interesting, it proved quickly that those approaches were outperformed by dedicated algorithms (see for example [Rég95]).

2.5 CP Approaches to Subgraph Isomorphism

In this section, we review the constraint programming approaches for the graph morphism problems, mainly graph homomorphism and (sub)graph isomorphism. Section 2.5.1 describes common modelings. Section 2.5.2 describes the implementation for the constraints stated in the models. Finally, Section 2.5.3 discusses the related works published in the constraint programming field.

2.5.1 Modeling

The graph morphism problem has been modeled in several ways in the literature. In the following, we omit the constraint that impose the compatibility of the labelings together with the degree preprocessing constraint, as those are straightforward to formulate.

Node based [McG79, Rég95, LV02] In the node based model, the matching function maps nodes to nodes, and mapping of edges is ignored. The standard model states the morphism condition, together with additional conditions to produce a particular type of morphism. More formally, the node based model is defined as :

- a vector of variables $[x_1, \ldots, x_n]$ with $D(x_i) = V_t$, representing the function.
- a constraint
$$(i, j) \in E_p \Rightarrow (x_i, x_j) \in E_t \qquad \text{(MC)}$$
representing the morphism condition, or the constraint
$$(i, j) \in E_p \Leftrightarrow (x_i, x_j) \in E_t \qquad \text{(IMC)}$$
for the induced morphism constraint.

Another formulation of the morphism constraint enforces the neighborhood relationship between pattern nodes instead of mapping pattern edges:

$$\forall\, i \in V_p\ \forall\, j \in N_p(i) : x_j \in \cup_{a \in D(x_i)} N_t(a). \qquad \text{(NC)}$$

This former condition is equivalent to the morphism condition (MC).

Node and edge based [Rud98a] In the node and edge based model, the nodes and edges are explicitly represented as variables. A first function maps nodes to nodes, while a second one maps edges to edges. Morphism constraint are implemented by linking the nodes to the edges. If a vertex k is the source of an edge e in the pattern graph, then they must be assigned to a vertex k' and an edge e' such that k' is the source of the edge e'. The same constraint hold for the target of an edge. More formally, the node and edge based model is defined as :

- a vector of variables $[x_1, \ldots, x_n]$ with $D(x_i) = V_t$, representing the node function.
- a vector of variables $[e_1, \ldots, e_{|E_p|}]$ with $D(e_i) = E_t$, representing the edge function.
- $i \in V_p \wedge j \in E_p \wedge head(j) = i \Rightarrow head(e_j) = x_i$.
- $i \in V_p \wedge j \in E_p \wedge tail(j) = i \Rightarrow tail(e_j) = x_i$.

Since edges must be mapped to edges, the node and edge based model is a partial morphism model.

The most used model in the literature is the node based. Indeed, the node and edge based model creates an additional $O(m_p m_t)$ vector (where m denotes the number of edges of a graph). This would slow the constraint programming approach, especially comparing to dedicated algorithms that uses $\Theta(n_t)$ vectors, such as vflib.

Deriving other graph morphism problems Deriving the other types of graph morphism problems can be done by constraining the mapping function f, represented by the vector of variables. Those constraints can be systematically derived from Section 2.2.2. For graph isomorphism, the function is total and bijective. For subgraph isomorphism, the function is total and injective. The constraint $\forall\, i, j \in V_p : i \neq j \Rightarrow x_i \neq x_j$ is added, usually modeled as a global alldiff constraint. For graph epimorphism, the function is total and surjective. For maximum common subgraph, the function is partial and injective, and its domain must be maximized. We will give detailed models for those problems in the next chapter.

2.5.2 Implementing morphism constraints

The injective condition in the subgraph isomorphism is forced by an alldiff constraint. We focus first on the implementation of morphism constraint (MC). We also show how to implementent an additional redundant constraint, and the neigborhood morphism constraint (NC). Finally, we compare the complexities of those implementations.

Morphism constraint (c2) A classical AC-consistency algorithm would cost $O(m_p n_t^2)$ amortized time. By using the problem structure, its amortized complexity can be reduced to $O(n_p n_t d)$ [LV02]. We also call this constraint `c2`, following the naming convention introduced in [LV02].

The propagator keeps track of relations between all the target nodes and the domain $D(x_i)$ in a structure $S(i,a) = |D(x_i) \cap N_t(a)|$ representing the number of relations between a target node a and $D(x_i)$. Whenever the neighbors of a target node a have no relation with $D(x_i)$, that is when $S(i,a) = 0$, node a is pruned from all neighbors of x_i. Algorithm 6 shows an implementation of the morphism constraint. It has an $O(n_p n_t d)$ amortized time complexity, and the structure $S(i,a)$ has a $\Theta(n_p n_t)$ spatial complexity. Line 3 is called at most $O(n_p n_t)$ times, and line 4 is executed at most $O(n_p n_t d)$ times. Condition in line 5 is true only $O(n_p n_t)$ times, and hence line 7 is executed at most $O(n_p n_t d)$ times. The preprocessing to compute $S(i,a)$ costs $O(n_p n_t d)$.

Algorithm 6: Morphism Propagator

```
1 Propagate_MC(i,a)
2 // Element a exits from D(x_i)
3 for b ∈ N_t(a) do
4     S(i,b) ← S(i,b) − 1
5     if S(i,b) = 0 then
6         foreach j ∈ N_p(i) do
7             D(x_j) ← D(x_j) \ {b}
```

McGregor [McG79] demonstrated that making the graph morphism CSP GAC is equivalent to performing the refine procedure of the Ullmann algorithm (see Section 2.4.1).

It is worth noting that the Algorithm 6 proposed by [LV02] prop-

agates only the morphism constraint (MC), that is $(x, y) \in E_p \Rightarrow (f(x), f(y)) \in E_t$. We will see in the next chapter how to extend efficiently this implementation of the (MC) condition to the (IMC) condition.

Forward checking morphism constraint A forward checking morphism constraint can be designed. This is especially useful when instances with a lot of solutions are considered. Algorithm 7 gives the pseudo-code. Line 6 and 7 implement the induced morphism propagator and are optional. The forward checking propagator implementing the induced morphism constraint is obtained by replacing the neighboor function by their complementary neighboorhood functions, that is $\overline{N}_p(i) = \{j \mid (i, j) \notin E_p\}$ and $\overline{N}_t(a) = \{b \mid (a, b) \notin E_t\}$. The amortized time complexity is $O(n_p d)$ for the partial morphism constraint, and $O(n_p^2)$ for the induced morphism constraint.

Algorithm 7: Forward checking morphism constraint

```
1 Propagate_FC(i)
2 // Variable x_i instantiated
3 a ← D(x_i)
4 for j ∈ N_p(i) do
5 |  D(x_j) ← D(x_j) ∩ N_t(a)
6 [ for j ∈ N̄_p(i) do
7 |  D(x_j) ← D(x_j) \ N̄_t(a) ]
```

Local Alldiff Constraint (c3) A redundant constraint pruning the search space has been proposed in [LV02]. We also call this redundant constraint `c3`, following the naming convention introduced in [LV02]. This constraint is a local Alldiff constraint [Reg94] upon the neighborhood of a node, by noting that the number of candidates available in the union of x_i neighbors domain could be less than the actual number of x_i neighbors in the pattern graph :

$$LA(x_i) \equiv |\cup_{j \in N_p(i)} D(x_j) \cap N_t(x_i)| \geq |N_p(i)| \,. \tag{2.1}$$

A structure $CT(i, a) = |\cup_{j \in N_p(i)} D(x_j) \cap N_t(a)|$ is updated through the use of an intermediate structure $R(i, a) = |\{j \in N_p(i) \mid a \in D(x_j)\}|$.

It counts the number of neighbors of x_i which have a in their domain. Whenever $R(i,a)$ equals 0, $CT(i,b)$ decreases by 1 for all b in the neighbor of a in the target graph. The expression $|N_p(i)|$ can be obtained in $O(1)$. Algorithm 8 describes an implementation of the $LA(x_1,...,x_n)$ constraint. The amortized complexity of this redundant constraint is $O(n_p n_t d)$ and its space complexity is $\Theta(n_p n_t)$. Line 3 is called at most $O(n_p n_t)$ times. Line 4 is called at most $O(n_p n_t d)$ times. Condition at line 5 is true at most $O(n_p n_t)$ times. Line 7 is this executed at most $O(n_p n_t d)$ times, and line 8 and line 9 can be considered as constant time. The preprocessing time to build the $CT(i,a)$ and $R(i,a)$ structures is $O(n_p n_t d)$.

Algorithm 8: Local alldiff constraint

```
1 Propagate_LA(i,a)
2 // Element a exits from D(x_i)
3 for j ∈ N_p(i) do
4     R(j,a) ← R(j,a) − 1
5     if R(j,a) = 0 then
6         foreach b ∈ N_t(a) do
7             CT(j,b) ← Ct(j,b) − 1
8             if CT(j,b) < |N_t(j)| then
9                 D(x_j) ← D(x_j) \ {b}
```

Neighborhood morphism constraint An arc consistent algorithm can also be designed for the neighborhood morphism constraint (NC). Algorithm 9 gives the pseudo-code. The propagator is called at most $O(n_p n_t)$ times and line 4 is executed at most $O(n_t d)$ times, and hence the total time complexity is $O(n_p n_t^2 d)$. The induced version is obtained by adding lines 7 to 10. Then the amortized complexity is $O(n_p^2 n_t^2)$.

Complexity We consider the most widely used model in the literature, that is the node based model and arc-consistent constraint for the morphism condition (MC). For such a model the arc-consistent alldiff constraint costs $O(n_p^2 n_t^2)$, while the arc-consistent morphism constraint has an amortized complexity of $O(n_p n_t d)$. Hence in the best case the time complexity is $O(n_p^3 n_t^2)$, while the wost case time complexity is

Algorithm 9: Neighborhood morphism constraint

```
1  Propagate_NEIGH(i,a)
2  // Element a exits from D(x_i)
3  for b ∈ D(x_i) do
4      R ← R ∪ N_t(b)
5  for j ∈ N_p(i) do
6      D(x_j) ← D(x_j) ∩ R
7  [ for b ∈ D(x_i) do
8      R̄ ← R̄ ∪ N_t(b)
9  for j ∈ N̄_p(i) do
10     D(x_j) ← D(x_j) ∩ R̄ ]
```

$O(n_p! n_p^2 n_t^2)$. Regarding memory, we consider that the restoration strategy is copying. The variables, the domains and the constraints have to be saved and restore. Saving the variables and the domains costs $\Theta(n_p n_t)$. The alldiff constraint and the morphism constraints have a memory complexity of $\Theta(n_p n_t)$. Hence the memory complexity of the constraint programming approach is $\Theta(n_p^2 n_t)$, as only $\Theta(n_p)$ of previous states have to be kept during the search.

2.5.3 Related Works

Several authors in the constraint programming fields have proposed models and implementation of constraints:

- [McG79]: McGregor introduces a simple model based on the nodes of the graphs, and investigates path consistency with (sub)graph isomorphism. It is the first work about graph matching and constraint programming.

- [Rég95]: J.C. Régin also uses a model based on the nodes of the graphs, and uses subgraph isomorphism to propose $AC7$.

- [Rud98a]: the main contribution resides in modeling. Rudolf proposes the node and edge model, together with the semantic of its constraints, in the context of graph homomorphism. However, checking is used, and arc consistency is introduced at a very late

stage in the paper, and no clear implementation and experiments are given.

- [LV02]: Valiente is one of the most recent work about graph matching and constraint programming. This work uses subgraph isomorphism to compare efficiency of levels of consistency. He uses a model based on the mapping of the nodes of the graphs. The main contribution of this paper is to give concrete implementation of the morphism constraints. Moreover, the global constraint alldiff is used. Finally, another contribution of this paper is to present a redundant constraint and its implementation. It is shown that it is powerful on difficult instances, even thought the mean time is increased.

- [Pug05a] [Pug03] : Puget uses (sub)graph isomorphism problems as tools for dominance checks in the context of symmetry [Pug05a] and for symmetry detection [Pug03]. A node based model is used and the neighborhood constraint is proposed.

2.6 Conclusion

The exact graph matching problems can be seen as an instantiation of a morphism problem. Subgraph isomorphism for example can be seen as an injective graph morphism. Graph morphism problems can be modeled as constraints. However, regarding subgraph isomorphism, it remains to be proved that constraint programming can give rise to a competitive approach. Table 2.2 compares the complexities of the dedicated algorithm together with the arc-consistent CP approach for the partial subgraph isomorphism problem. It is clear that vflib outperforms both the Ullmann approach and the arc-consistent CP approach. We could look for a lower level of consistency. With forward checking, the morphism constraint is $O(d)$, and the alldiff constraint is $O(n_t)$. The best case and the worst case complexities are thus $O(n_p n_t d)$ and $O(n_p! n_t d)$, still worse than vflib. Moreover, the practical overhead created by the usage of a constraint engine may also decrease the performance of the CP approach. The memory complexity is where the constraint programming cannot beat vflib. The memory requirement of $\Theta(n_p^2 n_t)$ is inherent to constraint programming, since variables and domains have to be explicitly represented, and have to be saved and restored. Those reasons explain why it is usually believed that a declarative approach

cannot compete against a fast and simple dedicated algorithms for the subgraph isomorphism problem. However, we will show in the next chapters that not only can CP outperforms state-of-the-art algorithms, but the CP approach can even offer an attractive declarative framework.

Table 2.2: Comparison of the complexities of Ullman, vflib and the arc-consistent CP approach for the best case and the worst case.

	Ullmann		Vflib	
	B.C.	W.C.	B.C.	W.C.
time	$O(n_p^2 n_t d)$	$O(n_p! n_p n_t d)$	$O(n_p d)$	$O(n_p! d)$
memory	$\Theta(n_p^3 n_t)$	$\Theta(n_p^3 n_t)$	$\Theta(n_t)$	$\Theta(n_t)$

CP AC		CP FC	
B.C.	W.C.	B.C.	W.C.
$O(n_p^3 n_t^2)$	$O(n_p! n_p^2 n_t^2)$	$O(n_p n_t d)$	$O(n_p! n_t d)$
$\Theta(n_p^2 n_t)$	$\Theta(n_p^2 n_t)$	$\Theta(n_p^2 n_t)$	$\Theta(n_p^2 n_t)$

3

Models using structured constraints[1]

3.1 Introduction

Declarative modeling together with efficiency is the goal of constraint programming. In this chapter we propose a modeling schema for graph matching problems inside constraint programming. We show how the integration of two domains of computation over countable structures, *graphs* and *maps*, can be used for modeling and solving various graph matching problems. To achieve this, we extend map variables allowing the domain and range to be non-fixed and constrained. We describe how such extended maps are designed and realized on top of finite domain and finite set variables with specific propagators. Moreover, thanks to graph variables, morphism constraints can be stated between the pattern graph, the target graph, and the matching function.

Using graph variables instead of graph structure opens the constraint programming to approximation. In many areas, the structure of the pattern can only be approximated and exact matching is then far too stringent. Approximate matching is a possible solution, and can be handled in several ways. In a first approach, the matching algorithm may allow part of the pattern to mismatch the target graph (e.g. [WZC95, MB98, DK03, CS03b]). The matching problem can then be stated in a probabilistic framework (see, e.g. [RKH05]). In a second approach,

[1] Part of this chapter has been published in [DDZD05], [ZDD05] and [DZD08].

the approximations are declared by the user within the pattern, stating which part could be discarded (see, e.g. [GS02, DDD05]). This approach is especially useful in fields where one faces a mixture of precise and imprecise knowledge of the pattern structures. In this approach, which will be followed in this chapter, the user is able to declare parts of the pattern open to approximation. The ideal would be that the user states properties on the pattern to be found and the CP engine finds it efficiently.

In constraint programming, two domains of computation over countable structures have received recent attention : graphs and maps. In CP(Graph) [DDD05], graph variables, and constraints on these variables are described (see also [Ger93, CDPP04] for similar ideas). CP(Graph) can be used to express and solve combinatorial graph problems modeled as constrained subgraph extraction problems. In [Ger97, FHK01], function variables are proposed, but the domain and range are limited to ground sets. Such function variables are useful for modeling problems such as warehouse location.

There is a trade-off between declarative modeling and efficiency. We will show that our approach outperforms the state-of-the-art in subgraph isomorphism and compete with the vflib approach of maximum common subgraph.

Contributions

The main contributions of this chapter are the following:

- Extension of function variables, where the domain and range of the mapping are not limited to ground sets, but can be finite set variables. Introduction of the MapVar and Map constraints which allow to use the non-fixed feature of our map variables.
- Demonstration of how a single constraint is able to express a wide range of graph matching problems thanks to three high-level structured variables. In particular, we show how switching a parameter from a fixed graph to a graph interval opens a new spectrum of matching problems. We show how additional constraints imposed on this graph interval enable the expression of hybrid problems such as approximate graph matching. The beauty and originality of this approach resides in that those problems are either new or were always treated separately, illustrating the expressive power and generality of constraint programming.

- Experimental evaluation of our CP approach. We show that this modeling exercise is not only aesthetic but is actually competitive with the current state-of-the-art in subgraph isomorphism (vflib). The generic approach does not hinder the efficiency of the solver. On a standard benchmark set, we show that our approach solves in a given time limit a fourth of the instances which cannot be solved by vflib while only spending between 9% and 22% more time on instances solved by the two competing approaches.

The next section describes the basic idea behind the CP(Graph) framework. CP(Map), our extension to function variables in CP is described in Section 3. Approximate graph matching is defined in Section 4, and its modeling within CP(Graph+Map) is handled in Section 5. Section 6 discusses the implementation of the morphism constraint. Section 7 analyzes experimental results, and Section 8 concludes this Chapter.

3.2 CP(Graph)

Graphs have already been studied in constraint programming. Some problems involving undetermined graphs have been formulated using either binary variables, sets ([Ger93, CDPP04]) or integers (successor variables e.g. in [BC94, PGPR98]). CP(Graph) [DDD05] unifies those models by recognizing a common structure. Graph variables are variables whose domain ranges over a set of graphs and as with set variables [Pug92, Ger97], this set of graphs is represented by a graph interval $[\underline{D}(G), \overline{D}(G)]$ where $\underline{D}(G)$, the greatest lower bound (glb) and $\overline{D}(G)$, the least upper bound (lub) are two graphs such that $\underline{D}(G)$ a (partial) subgraph of $\overline{D}(G)$ (we write $\underline{D}(G) \subseteq \overline{D}(G)$). These two bounds are referred to as the lower and the upper bound. The lower bound $\underline{D}(G)$ is the set of all nodes and arcs which *must* be part of the graph in a solution while the upper bound $\overline{D}(G)$ is the set of all nodes and arcs which could be part of the graph in some solution. The domain of a graph variable with $D(G) = [\underline{D}(G), \overline{D}(G)]$ is the set of graphs g such that $\underline{D}(G) \subseteq g \subseteq \overline{D}(G)$. Here, g is used to denote a constant graph and G is used to denote a graph variable. This notation is used throughout this chapter: in CSP, lowercase letters denote constants and uppercase letters denote domain variables.

Graph variables can be implemented using a dedicated data-structure or translated into set variables, integer variables or binary

variables. For instance, a graph variable G can be modeled as a set of nodes N and a set of arcs E with an additional constraint enforcing the relation $E \subseteq N \times N$. Whatever the graph variable implementation, two basic constraints $Nodes(G, SN)$ and $Arcs(G, SA)$ allow to access respectively the set of nodes SN and the set of arcs SA of the graph variable G. To simplify the notation the expression $Nodes(G)$ is used to represent a set variable constrained to be equal to the set of nodes of G. A similar notation is used for arcs.

Various constraints have been defined over such graph variables (or their preceding specialized models); see for instance the cycle [BC94], tree [BFL05], path [Sel03, CB04], minimum spanning tree [DK06] or spanning tree optimization constraint [DK07]. In the remainder of this article, we only use the two simple constraints $Subgraph(G_1, G_2)$ (also denoted $G_1 \subseteq G_2$) and $InducedSubgraph(G_1, G_2)$ (also denoted $G_1 \subseteq^* G_2$). $G_1 \subseteq G_2$ holds if G_1 is a partial subgraph of G_2, its propagator enforces that the lower and upper bounds of G_1 are subgraphs of the lower and upper bounds of G_2 respectively. The constraint $G_1 \subseteq^* G_2$ states that G_1 is the node-induced subgraph of G_2. It holds if G_1 is a subgraph of G_2 such that for each arc a of G_2 whose end-nodes are in G_1, a is also in G_1.

The CP(Graph) computation domain is an approximation of the exponential number of graphs that is part of the domain of a graph variable G. This is also the case for the set computation domain. It could be argued that such an upper and lower bound representation is very crude. For instance, a graph inside the lattice $D(G)$ may not be removed as only bounds can be modified, and inferences of filtering algorithms are only limited to the bounds. However, even for sets, the state-of-the-art resides in the lattice representation, for time and space complexity concerns. Only few recent works propose new representations of set, using for example a length-lexicographic ordering [GH06] or [DBL06] ROBDDs. Delving in a more accurate representation for the graph computation domain is outside the scope of this work.

3.3 CP(Map)

The value of a map variable is a mapping from a domain set to a range set. By map or mapping, we mean here a function, following the mathematical usage. The domain of a map variable is thus a set of mappings. Map variables were first introduced in CP in [Ger97] where Gervet de-

fines relation variables. However, the domain and the range of the relations were limited to ground finite sets. Map variables were also introduced as high level type constructors, simplifying the modeling of combinatorial optimization problems. This was first defined in [FHK01] as a relation or map variable M from set v into a set w, where supersets of v and w must be known. Such map variables are then compiled into OPL. This idea is developed in [Hni03], but the domain and range of a map variable are limited to ground sets. Relation and map variables are also described in [FJHM05] as a useful abstraction in constraint modeling. Rules are proposed for refining constraints on these complex variables into constraints on finite domain and finite set variables. Map variables were also introduced in modeling languages such as ALICE [Lau78], REFINE [Smi87] and NP-SPEC [CPSV99]. To the best of our knowledge, map variables were not yet introduced directly in a CP language. One challenge is then to extend current CP languages to allow map variables as well as constraints on these variables.

In the remaining of this section, we show how a CP(Map) extension can be realized on top of finite domain and finite set variables.

3.3.1 The Map domain

We consider the domain of total surjective functions. Given two elements $m_1 : s_1 \rightarrow t_1$ and $m_2 : s_2 \rightarrow t_2$, where s_1, s_2, t_1, t_2 are sets, we have $m_1 \subseteq m_2$ iff $s_1 \subseteq s_2 \wedge t_1 \subseteq t_2 \wedge \forall x \in s_1 : m_1(x) = m_2(x)$. We also have that $m = \text{glb}(m_1, m_2)$ is a map $m : s \rightarrow t$ with $s = \{x \in s_1 \cap s_2 \mid m_1(x) = m_2(x)\}$, $t = \{v \mid \exists x \in s : m_1(x) = v\}$, and $\forall x \in s : m(x) = m_1(x) = m_2(x)$. The lub between two elements m_1, m_2 exists only if $\forall x \in s_1 \cap s_2 : m_1(x) = m_2(x)$. In that case the lub is a map $m : s \rightarrow t$ with $m(x) = m_1(x)$ if $x \in s_1$, and $m(x) = m_2(x)$ if $x \in s_2$, $s = s_1 \cup s_2$, and $t = \{v \mid \exists x \in s : m(x) = v\}$. The domain of total surjective functions is then a meet semi lattice, that is a semi lattice where every pairs of elements has a glb.

3.3.2 Map variables and the MapVar constraint

A map variable is declared with the constraint $MapVar(M, S, T)$, where M is the map variable and S, T are finite set variables of Cardinal [Aze07]. The domain of M is all the *total surjective* functions from s to t, where s, t are in the domain of S, T. We call S the *source set* of M, and T the *target set* of M. When M is instantiated (when its domain is

a singleton), the source set and the target set of M are ground sets corresponding to the domain and the range of the mapping. As usual, the domain of a set variable S is represented by a set interval $[\underline{D}(S), \overline{D}(S)]$, the set of sets s with $\underline{D}(S) \subseteq s \subseteq \overline{D}(S)$.

Example Let M be a map variable declared in $MapVar(M, S, T)$, with $dom(S) = [\{8\}, \{4, 6, 8\}]$ and $dom(T) = [\{\}, \{1, 2, 4\}]$. A possible instance of M is $\{4 \rightarrow 1, 8 \rightarrow 4\}$. On this instance, $S = \{4, 8\}$, and $T = \{1, 4\}$. Another instance is $M = \{4 \rightarrow 1, 8 \rightarrow 1\}$, $S = \{4, 8\}$, and $T = \{1\}$.

Map variables can be used for defining various kinds of mappings, such as :

- Surjective function : $SurjectFct(M, S, T) \equiv MapVar(M, S, T)$.
- Bijective function : $BijectFct(M, S, T) \equiv SurjectFct(M, S, T) \wedge \forall i, j \in S : i \neq j \Rightarrow M(i) \neq M(j)$.
- Injective function : $InjectFct(M, S, T) \equiv T' \subseteq T \wedge BijectFct(M, S, T')$
- Total function : $TotalFct(M, S, T) \equiv T' \subseteq T \wedge SurjectFct(M, S, T')$
- Partial function : $PartialFct(M, S, T) \equiv S' \subseteq S \wedge TotalFct(M, S', T)$

In order to access individual elements of the map, we define the constraint $Map(M, X, V)$, where X and V are finite domain variables. Given a map variable declared with $MapVar(M, S, T)$, the constraint $Map(M, X, V)$ holds when $X \in S \wedge V \in T \wedge M(X) = V$. We also define the constraint $M1 \subseteq M2$ where $M1$ and $M2$ are map variables.

3.3.3 Implementing Map Variables in a Finite Domain Solver

When a map variable M is declared by $MapVar(M, S, T)$, a finite domain (FD) variable M_x is associated to each element x of the upper bound of the source set ($\overline{D}(S)$).

The semantics of these FD variables is simple : M_x represents $M(x)$, the image of x through the function M. Since the source set S can be non-fixed, x might not be in S and its image would not be defined. A special value $\perp$ is used for this purpose. The relationship between the

domain of each variable M_x and the set variables S and T can be stated as follows :

- (1) $S = \{x \mid M_x \neq \perp\}$ (M is total)
- (2) $T = \{v \mid \exists x : M_x = v \neq \perp\}$ (M is surjective)

Given $MapVar(M, S, T)$, the domain of M is the set of total surjective functions $m : s \rightarrow t$ with $s \in D(S)$, $t \in D(T)$, $\forall x \in s : m(x) \in D(M_x)$, and $\forall x \notin s : \perp \in D(M_x)$.

As can be seen on Figure 3.1, these variables are stored in an array and accessed by value x through a dictionary data structure (e.g. hashmap) *index* used to store the index in the array of each value of $\overline{D}(S)$. The initial domain of each FD variable is $\overline{D}(T) \cup \{\perp\}$.

3.3.4 Additional Constraints and Propagators

Given two map constraints $MapVar(M1, S1, T1)$ and $MapVar(M2, S2, T2)$ the constraint $M1 \subseteq M2$ is implemented as $S1 \subseteq S2 \wedge T1 \subseteq T2 \wedge \forall x \in S1 : M1_x = M2_x$. The last conjunct can be implemented as a set of propagation rules :

- $x \in \underline{D}(S1) \rightarrow M1_x = M2_x$
- for each $x \in \overline{D}(S1) \setminus \underline{D}(S1) : M1_x \neq M2_x \rightarrow x \notin S1$.

The constraint $Map(M, X, V)$ is translated to $Element(index(X), I, V) \wedge X \in S \wedge V \in T$, where S and T are the source and target sets of M, I is the array representing the FD variables M_x, and $index(X)$ is a finite domain obtained by taking the index of each value of the domain of X using the *index* dictionary.

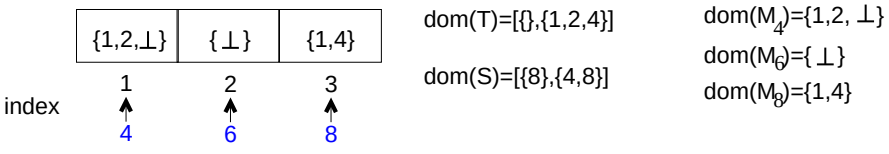

Figure 3.1: Implementation of $MapVar(M, S, T)$ (with initial domain $dom(S) = [\{8\}, \{4, 6, 8\}]$ and $dom(T) = [\{\}, \{1, 2, 4\}]$), assuming (other) constraints already achieved some pruning.

The implementation of $BijectFct(M, S, T)$ is realized through $MapVar(M, S, T) \wedge AllDiffExceptVal(I, \perp) \wedge |S| = |T|$, where I is the array representing the FD variables M_x, and $AllDiffExceptVal$ holds when all the FD variables in I are different when their value is not $\perp$ [Bel00].

Given $MapVar(M, S, T)$, the propagation between M, S and T is based on their relationship described in the previous section, and is achieved by maintaining the following invariants :

- $\overline{D}(S) = \{x \mid D(M_x) \neq \{\perp\}\}$
- $\underline{D}(S) = \left\{x \in \overline{D}(S) \mid \perp \notin D(M_x)\right\}$
- $\overline{D}(T) = \{v \mid v \neq \perp \wedge \exists x : v \in D(M_x)\}$
- $\underline{D}(T) \supseteq \{v \mid v \neq \perp \wedge \exists x : D(M_x) = \{v\}\}$

The last invariant is not an equality because when a value is known to be in T, it is not always possible to decide which element in I should be assigned to v.

Propagations rules are then easily derived from these invariants (two rules per invariant) :

$$\begin{aligned}
M_x = \perp &\rightarrow x \notin \overline{D}(S) \\
x \notin \overline{D}(S) &\rightarrow M_x = \perp \\
x \in \underline{D}(S) &\rightarrow M_x \neq \perp \\
M_x \neq \perp &\rightarrow x \in \underline{D}(S) \\
v \notin \overline{D}(T) \wedge v \neq \perp &\rightarrow v \notin D(M_x) \\
NbOccur(I, v) = 0 \wedge v \neq \perp &\rightarrow v \notin \overline{D}(T) \\
M_x = v \neq \perp &\rightarrow v \in \underline{D}(T) \\
v \in \underline{D}(T) \wedge NbOccur(I, v) = 1 \wedge v \in D(M_x) &\rightarrow M_x = v
\end{aligned}$$

where $NbOccur(I, v)$ denotes the number of occurrences of v in the domains of the FD variables in I. Each of these propagation rules can be implemented in $O(1)$ (assuming a bit representation of sets). The implementation of propagators also exploits the cardinality information associated with set variables.

3.3.5 A global constraint based on matching theory

The above propagators do not prune the M_x FD variables (except the $\perp$ value). We show here how flow and matching theory can be used to design a complete filtering algorithm for the $MapVar(M, S, T)$ constraint. The algorithm is similar to that of the GCC and Alldiff constraints but is based on a slightly different notion: the V-matchings (see [Thi04]). In the remainder of this section we show that V-matchings characterize the structure of the $MapVar$ constraint. Note that it also has similarities with the Nvalue, Range and Roots constraints ([BHH+05a, BHH+05b]).

Definition. The *variable-value* graph of a $MapVar(M, S, T)$ constraint is a bipartite graph where the two classes of nodes are the elements of $\overline{D}(S)$ on one side and the elements of $\overline{D}(T)$ plus $\perp$ on the other side. An arc (x, v) is part of the graph iff $v \in D(M_x)$.

Definition. In a bipartite graph $g = (N_1 \cup N_2, A)$, a *matching* M is a subset of the arcs such that no two arcs share an endpoint : $\forall (u_1, v_1) \neq (u_2, v_2) \in M : u_1 \neq u_2 \wedge v_1 \neq v_2$. A matching M *covers* a set of nodes V, or M is a V-matching of g iff $\forall x \in V : \exists (u, v) \in M : u = x \vee v = x$

The following property states the relationship between matching in the bipartite graphs and solutions of the $MapVar$ constraint.

Property 1. Given the constraint $MapVar(M, S, T)$ and its associated variable-value graph g, assuming the constraint is consistent, we have :

- (1) Any solution $m : s \rightarrow t$ contains a t-matching of g, and any t-matching can be extended to a solution.
- (2) An arc (x, v) belongs to a $\underline{D}(T)$-matching of g, *iff* there exists a solution m with $m(x) = v$.

Proof. (1) The solution m is surjective; every node of t must have at least one incident arc. If we choose one incident arc per node in t, we have a t-matching as m is a function.

Given a t matching, let $m : s \rightarrow t$ be the bijective function corresponding to this matching. Adding arcs to t leads to a surjective function. Let $s' = \underline{D}(S) \cup s$, and $t' = \underline{D}(T) \cup t$. Since the constraint is consistent, $\forall x \in s' \setminus s \ \exists (x, v) \in g : v \neq \perp$, and $\forall v \in t' \setminus t \ \exists (x, v) \in g$. Adding all these arcs leads to a surjective function which is a solution.

(2) ($\Rightarrow$) This is a special case of the second part of (1).

($\Leftarrow$)Let $m : s \rightarrow t$ be a solution with $m(x) = v$. We then have $(x, v) \in g$. By (1), the graph g contains a t-matching M which is also a $\underline{D}(T)$-matching as $\underline{D}(T) \subseteq t$. If $(x, v) \in M$ we are done. Assume $(x, v) \notin M$. Then x is free with respect to M because $M(x) = v$. As $v \in t$, v is covered by M; there is a variable node w such that $(w, v) \in M$. Then, x, v, w is an even alternating path starting in a free node. Replacing (w, v) by (x, v) leads to another t-matching, hence a $\underline{D}(T)$-matching of g. ■ □

From Property 1, an arc-consistency filtering algorithm can be derived : compute the set A of arcs belonging to some $\underline{D}(T)$-matching of the bipartite graph; if $(x, v) \notin A$, remove v from $D(M_x)$. The computation of this set can be done using techniques such as described in [Thi04], with a complexity of $O(mn)$, where n is the size of T, and m is the number of arcs in the variable-value graph.

3.4 Approximate graph matching and other matching problems

In this section, we define different matching problems ranging from graph homomorphism to approximate subgraph matching.

A useful extension is *approximate* subgraph matching, where the pattern graph and the found subgraph in the target graph may differ with respect to their structure [ZDD05]. We choose an approach where the approximations are declared by the user in the pattern graph through optional nodes and forbidden arcs.

In many graph morphism problems, all the specified nodes in the patterns must be matched. It would be interesting to allow some of them to be optional. Such nodes are declared *optional* in the pattern graph. Arcs can also be incident to optional nodes. Once an optional node is matched, all its incident arcs to other matched nodes must be matched too. The selected pattern must thus be an induced subgraph of the complete pattern.

In graph isomorphism, if two nodes in the pattern are not related by an arc, this absence of arc is an implicit forbidden arc in the matching. It would be interesting to declare explicitly which arcs are *forbidden*, leading to problems between monomorphism and isomorphism.

In Figure 3.2, mandatory nodes are represented as filled nodes, and optional nodes are represented as empty nodes. Mandatory arcs are

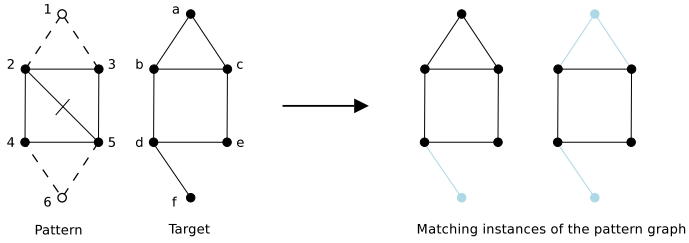

Figure 3.2: Example of approximate matching.

represented with plain line, and arcs incident to optional nodes are represented with dashed lines. Forbidden arcs are represented with a plain line crossed.

In that figure, node 6 cannot be matched to node f because only one of the arcs $(6, 4)$ and $(6, 5)$ in the pattern can be matched in the target. The right side of the figure presents two solutions of the matching problem. The nodes and arcs not matched in the target graph are greyed.

A pattern graph with optional nodes and forbidden arcs forms an *approximate pattern graph*, and the corresponding matching is called an *approximate subgraph matching*[ZDD05]. We focus here on approximate graph monomorphism.

Definition 1. An **approximate pattern graph** is a tuple (V_p, O_p, E_p, F_p) where (V_p, E_p) is a graph, $O_p \subseteq V_p$ is the set of optional nodes and $F_p \subseteq V_p \times V_p$ is the set of forbidden arcs, with $E_p \cap F_p = \emptyset$.

Definition 2. An **approximate subgraph matching** between an approximate pattern graph $P = (V_p, O_p, E_p, F_p)$ and a target graph $G = (V_t, E_t)$ is a *partial* function $f : V_p \to V_t$ such that:

1. $V_p \setminus O_p \subseteq dom(f)$
2. $\forall\, i, j \in dom(f) : i \neq j \Rightarrow f(i) \neq f(j)$
3. $\forall\, i, j \in dom(f) : (i, j) \in E_p \Rightarrow (f(i), f(j)) \in E_t$
4. $\forall\, i, j \in dom(f) : (i, j) \in F_p \Rightarrow (f(i), f(j)) \notin E_t$

The notation $dom(f)$ represents the domain of f. Elements of $dom(f)$ are called the selected nodes of the matching. According to this definition, if $F_p = \emptyset$ the matching is a subgraph monomorphism, and if $F_p = V_p \times V_p \setminus E_p$, the matching is an isomorphism.

3.5 Modeling graph matching and related problems

In this section, we show how CP(Graph+Map) can be used for modeling and solving a wide range of graph matching problems.

The problems of graph matching can be stated along three different dimensions:

- homomorphism versus monomorphism versus isomorphism;
- graph versus subgraph matching;
- exact versus approximate matching

These different problems illustrated in Table 3.1. All these problems can be modeled and solved through a morphism constraint on a map variable and two graph variables.

3.5.1 The basic morphism constraints

The two important morphism constraints introduced in this chapter are the $SurjMC(P, G, M)$ and $BijMC(P, G, M)$ constraints, which holds when M is a total surjective / bijective mapping from P to G respecting the morphism constraint.

$$SurjMC(P, G, M) \equiv SurjectFct(M, Nodes(P), Nodes(G)) \wedge MC(P, G, M)$$
$$BijMC(P, G, M) \equiv BijectFct(M, Nodes(P), Nodes(G)) \wedge MC(P, G, M)$$
$$\text{with} \quad MC(P, G, M) \equiv \forall (i, j) \in Arcs(P) : (M(i), M(j)) \in Arcs(G)$$

We now show how these two morphism constraints can be used to solve the different classes of problems.

3.5.2 Exact matching

Let p be a pattern graph and g be a target graph. The graphs p and g are ground objects in CP(Graph+Map). Graph homo and monomorphism can easily be modeled as shown in Table 3.1. Homomorphism (resp. monomorphism) requires a surjective (resp. bijective) function between p and a subgraph of g, respecting the morphism constraint. We use here a graph variable instead of a graph constant for the target graph (G with $D(G) = [\emptyset, g]$)

Exact matching	
homomorphism	$G \subseteq g \wedge SurjMC(p, G, M)$
monomorphism	$G \subseteq g \wedge BijMC(p, G, M)$
isomorphism	$BijMC(p, g, M)$ $\wedge\, BijMC(Comp(p), Comp(g), M)$
subgraph isomorph.	$G \subseteq^* g \wedge BijMC(p, G, M)$ $\wedge\, BijMC(Comp(p), Comp(G), M)$
Optional nodes	
homomorphism	$P \in [p_{man}, p] \wedge P \subseteq^* p \wedge G \subseteq g$ $\wedge\, SurjMC(P, G, M)$
monomorphism	$P \in [p_{man}, p] \wedge P \subseteq^* p \wedge G \subseteq g$ $\wedge\, BijMC(P, G, M)$
isomorphism	$P \in [p_{man}, p] \wedge P \subseteq^* p \wedge BijMC(P, g, M)$ $\wedge\, BijMC(Comp(P), Comp(g), M)$
subgraph isomorph.	$G \subseteq^* g \wedge P \in [p_{man}, p] \wedge P \subseteq^* p$ $\wedge\, BijMC(P, G, M)$ $\wedge\, BijMC(Comp(P), Comp(G), M)$
Forbidden arcs	
monomorphism	$G \subseteq^* g \wedge BijMC(p, G, M)$ $\wedge\, BijMC(p_{forb}, Comp(G), M)$

Table 3.1: Constraints for graph matching problems

Graph isomorphism requires a bijective function between p and g respecting two morphism constraints : one between the graphs, and a second between the complementary graphs. This requires a complementary graph constraint $CompGraph(G, Gc)$ which holds if $Nodes(G) = Nodes(Gc) = V_t$ and $Arcs(Gc) = (V_t \times V_t) \setminus Arcs(G)$. For conciseness, we also use the functional notation $Comp(G) = Gc$. In the subgraph isomorphism problem, there should exist an isomorphism between p and an induced subgraph of g.

3.5.3 Optional nodes and forbidden arcs

To cope with the optional nodes in the pattern graph, we replace the fixed graph pattern by a constrained graph variable, as illustrated in Table 3.1. Let p be the pattern graph with optional nodes, and p_{man} be the subgraph of p induced by the mandatory nodes of p. Graph

monomorphisms with optional nodes amounts to find an intermediate graph between p_{man} and p which is monomorphic to the target graph. However, between p_{man} and p, only the subgraphs induced by p should be considered. When two optional nodes are selected in the matching, if there is an arc between these nodes in pattern graph p, this arc must be considered in the matching, according to our definition of optional nodes, this is done through the use of the induced subgraph relation ($\subseteq^*$).

When all the nodes of the pattern graph are optional in the graph monomorphism, we have the *maximum common subgraph* problem by adding the size of P as an objective function. Similarly for subgraph isomorphism, this leads to the *maximum common induced subgraph* problem.

Allowing the specification of a set of forbidden arcs amounts to a simple generalization of the isomorphism problem, lying between monomorphism and isomorphism. As in the model for isomorphism, forbidden arcs are handled through a morphism constraint on the complement of the target graph. This time, only a specified set p_{forb} of arcs are forbidden. Isomorphism constitutes a special case where $p_{forb} = Arcs(Comp(p))$. This illustrated for the monomorphism problem in Table 3.1

The problem of approximate subgraph matching as defined in section 3.5, simply combines the use of optional nodes and forbidden arcs. Given an approximate pattern graph (V_p, O_p, E_p, F_p) where (V_p, E_p) is a graph, $O_p \subseteq V_p$ is the set of optional nodes, and $F_p \subseteq V_p \times V_p$ is the set of forbidden arcs, and a target graph (V_t, E_t), we define the following CP(Graph+Map) constants :

- p: the pattern graph (V_p, E_p),
- p_{man}: the subgraph of p induced by the mandatory nodes $V_p \setminus O_p$ of p,
- g: the target graph (V_t, E_t),
- p_{forb} : the graph (V_p, F_p) of the forbidden arcs.

The modeling of approximate matching is then a combination of graph monomorphism with optional nodes, and forbidden arcs.

$$G \subseteq^* g \wedge P \in [p_{man}, p] \wedge P \subseteq^* p \wedge BijMC(P, G, M)$$
$$\wedge Nodes(Pc) = Nodes(P) \wedge Pc \subseteq^* p_{forb} \wedge BijMC(Pc, Comp(G), M)$$

3.6 Implementing the $MC(P, G, M)$ constraint

The main difference between the $SurjMC(P, G, M)$ and $BijMC(P, G, M)$ constraints is an alldiff constraint ensuring the bijective property of the mapping M. A central constraint is the $MC(P, G, M)$ constraint, that ensures the morphism condition. The goal of this section is to explain the implementations of the MC constraint, based on a generalization to map and graph variable of the morphism constraint introduced in Chapter 2 (see Section 2.5.2). To simplify the presentation, the proposed implementations consider undirected graphs, using the undirected neighboorhood function $N_G(\dot{})$ where G is a graph variable. The algorithms can be however extended to the directed case, by instantiating the undirected $N_G(\dot{})$ by the directed $N^+_G(\dot{})$ and $N^-_G(\dot{})$. We start by studying the complexity of the MC constraint. We then describe the morphism propagator, followed by the description of the induced morphism propagator. We will show how the introduction of induced constraints over P and G can fasten the propagator. Finally, we will study a forward checking implementation of the MC constraint.

3.6.1 Complexity

The semantic of the MC constraint is an extension of the semantic of the morphism constraint and the induced morphism constraint (see Section 2.5.1):

$$(i, j) \in E_p \Rightarrow (x_i, x_j) \in E_t \quad \text{(MC)}$$

$$(i, j) \in E_p \Leftrightarrow (x_i, x_j) \in E_t \quad \text{(IMC)}$$

that implements the morphism constraints in state-of-the-art CP models.

Defining the semantic of the $MC(P, G, M)$ constraint to enforce graph homomorphism:

$$\forall\, (i, j) \in Arcs(P) \Rightarrow (M[i], M[j]) \in Arcs(G), \quad (3.1)$$

is NP-complete. The following formulation for induced graph homomorphism:

$$\forall\, (i,j) \in Arcs(P) \Leftrightarrow (M[i], M[j]) \in Arcs(G) \tag{3.2}$$

is also NP-Complete. It is already NP-Complete for the particular case of $MC(p, G, M)$. If there exists a polynomial algorithm to achieve GAC on $MC(p, G, M)$, then graph homomorphism can be solved in polynomial time. Indeed, proving there is no solution to $MC(p, G, M)$ could be done in polynomial time. Finding a solution could be done by $O(n_p)$ applications of the GAC MC propagator.

The proposed propagator achieves only GAC on the decomposition of 3.1 and 3.2, that is GAC is ensured on each following constraints:

$$(i,j) \in Arcs(P) \Rightarrow (M[i], M[j]) \in Arcs(G). \tag{3.3}$$

separately for each $i, j \in Arc(P)$ and for each following constraints:

$$(i,j) \in Arcs(P) \Leftrightarrow (M[i], M[j]) \in Arcs(G). \tag{3.4}$$

regarding the induced case.

3.6.2 Morphism Propagator

We propose an implementation of the general $MC(P, G, M)$ constraint. The propagator is an extension of the Valiente propagator (see Section 2.5.2) that handles the Valiente morphism propagator as a particular case.

Recall that the Valiente morphism constraint is based on a $S(i, a)$ data structure equal to $|M[i] \cap N_G(a)|$ for all $i \in Nodes(P)$ and for all $a \in Nodes(G)$. The structure $S(i, a)$ represents the number of target nodes in the domain of $M[i]$ that are neighboor of target node a. Whenever $S(i, a)$ is equal to zero, the target node a must be removed from the neighbors of i. The implementation of the rule:

$$S(i,a) = 0 \Rightarrow M[j] \leftarrow M[j] \setminus \{a\}\ \forall\, j \in N_P(i) \tag{3.5}$$

can be achieved by maintaining the $S(i, a)$ structure.

The main difference in our new modeling framework is that the pattern and target graphs are variables, and have a greatest lower bound (noted *glb* or simply *lb*) together with a least upper bound (noted *lub* or

simply ub). In this context the $S(i, a)$ structure has a lower bound and an upper bound, and there are two propagation rules:

$$S_{lb}(i,a) = |M[i] \cap N_{\underline{G}}(a)| = 0 \\ \Rightarrow i \in Nodes(\underline{P}) \wedge M[j] \leftarrow M[j] \setminus \{a\} \ \ \forall\, j \in N_{\underline{P}}(i) \qquad (3.6)$$

$$S_{ub}(i,a) = |M[i] \cap N_{\overline{G}}(a)| = 0 \\ \Rightarrow i \in Nodes(\underline{P}) \wedge M[j] \leftarrow M[j] \setminus \{a\} \ \ \forall\, j \in N_{\underline{P}}(i). \qquad (3.7)$$

Since the condition for pruning is $S(i, a) = 0$, only $S_{ub}(i, a)$ has to be checked to be equal to zero. Hence the $S(i, a)$ structure is now defined as $|M[i] \cap N_{\overline{G}}(a)|$, and has to be maintained in the algorithm.

The equation 3.7 depends on the variables P, G, M and the following four rules:

1. target node a exits from $M[i]$: the $S(i, a)$ structure has to be decremented and checked for the application of rule 3.7
2. target arc (a, b) exits from variable G: for all node $i \in Nodes(\overline{P})$, $S(i, a)$ has to be decremented if $b \in M[i]$ and checked for the application of rule 3.7.
3. pattern node i enters P: for all node $a \in Nodes(\overline{G})$, $S(i, a)$ has to checked for the application of rule 3.7.
4. pattern arc (i, j) enters P: for all $a \in M[i]$, if $S(i, a) = 0$ then a must be removed from $M[j]$ following rule 3.7.

The initial morphism algorithm from section 2.5.2 was only aimed at pruning on M. Hence the four rules deduced from equation 3.7 are partial. We also need to maintain P and G, regarding their arcs, following 3.3.

Algorithm 10 implements those five rules. The procedure `Propagate_M` implements the first rule by updating the $S(i, a)$ structure and checking for the propagation condition $S(i, a) = 0$. Then it checks for instantiated neighboor variables (i, j), forces (i, j) to belong to P and $(M[i], M[j])$ to belong to G. This implements the last rule not handled by the computation of the $S(i, a)$ structure. Since it can be called $O(n_p n_t)$ times, the total amortized cost of `Propagate_M` is $O(n_p n_t d_t)$. The procedure `Propag` has an amortized complexity of $O(n_p n_t d_p)$ since the condition $S(i, a) = 0$ can be met at most $O(n_p n_t)$

times. The procedure `Propagate_Arc_G` handles the second rule. It has an amortized complexity of $O(m_t n_p)$ since at most $O(m_t)$ edges can exit G. The procedure `Propagate_Node_P` has an $O(n_p n_t)$ amortized complexity, since there can be at most $O(n_p)$ nodes that enter P. Total amortized complexity of Algorithm 10 is $O(m_t n_p)$, still better than a generic consistency (see Section 2.5.2) which has a an amortized complexity of $O(m_p n_t^2)$.

3.6.3 Induced morphism propagator

Algorithm 10 proposed by [LV02] and presented in Section 2.5.2 propagates only the morphism constraint, that is $(x, y) \in E_p \Rightarrow (f(x), f(y)) \in E_t$.

We discuss here how to extend Algorithm 10 to implement equation 3.4. The only condition still to implement is:

$$(i, j) \notin Arcs(P) \Rightarrow (M[i], M[j]) \notin Arcs(G) \tag{3.8}$$

which can be rewritten as:

$$(i, j) \in Arcs(P_c) \Rightarrow (M[i], M[j]) \in Arcs(G_c), \tag{3.9}$$

where $P_c = Compl(P)$ and $G_c = Compl(G)$. The equation 3.9 can be implemented the same way as the equation MC. The propagation rule is thus:

$$\begin{aligned} &\overline{S}(i, a) = |M[i] \cap N_{\overline{G_c}}(a)| = 0 \\ &\qquad \Rightarrow i \in Nodes(P) \wedge M[j] \leftarrow M[j] \setminus \{a\} \; \forall \, j \in N_{\underline{P_c}}(i) \end{aligned} \tag{3.10}$$

The following rules can be deduced in order to prune M:

1. target node a exits from $M[i]$: the $\overline{S}(i, a)$ structure has to be decremented and checked for the application of rule 3.10
2. target arc (a, b) enters variable G: for all node $i \in Nodes(\overline{P})$, $\overline{S}(i, a)$ has to be decremented if $b \in M[i]$ and checked for the application of rule 3.10.
3. pattern node i enters P: for all node $a \in Nodes(\overline{G})$, $\overline{S}(i, a)$ has to checked for the application of rule 3.7.

Algorithm 10: Propagating $MC(P, G, M)$.

```
1  procedure Propagate_M(i,a)
2  // Element a ≠⊥ exits from M[i]
3  for b ∈ N_{\overline{G}}(a) do
4      S(i,b) ← S(i,b) − 1
5      Propag(i,b)
6  if M[i] = {u} ≠⊥ then
7      for j ∈ N_{\underline{P}}(i) s.t. M[j] = {v} ≠⊥ do
8          \underline{G} ← \underline{G} ∪ (M[i], M[j])
9          \underline{P} ← \underline{P} ∪ (i,j)
10 procedure Propagate_Arc_G(a,b)
11 // Arc (a,b) exits G
12 for i ∈ Nodes(\overline{P}) do
13     if b ∈ M[i] then
14         S(i,b) ← S(i,b) − 1
15         Propag(i,b)
16 procedure Propagate_Node_P(i)
17 // Node i enters P
18 for a ∈ M[i] do
19     Propag(i,a)
20 procedure Propagate_Arc_P(i,j)
21 // Arc (i,j) enters P
22 for a ∈ M[i] do
23     if S(i,a) == 0 then
24         M[j] ← M[j] \ {a}
25 procedure Propag(i,b)
26 if S(i,b) == 0 ∧ i ∈ Nodes(\underline{P}) then
27     for j ∈ N_{\underline{P}}(i) do
28         M[j] ← M[j] \ b
```

4. pattern arc (i, j) exits P: for all $a \in M[i]$, if $\overline{S}(i, a) = 0$ then a must be removed from $M[j]$ following rule 3.7.

Among those rules, there are two new events: a target arc (a, b) enters G (which is equivalent to a target arc (a, b) exits G_c) and a pattern arc enters P (which is equivalent to a pattern arc exits P_c). They are implemented in two new separated procedures, `Propagate_Arc_Enters_G` and `Propagate_Arc_Exits_P`. The two other rules are implemented in the same procedure as in Algorithm 10. We also need to add a rule to update P and G following equation 3.4.

Algorithm 11 implements those five rules. The rules associated with the constraint $MC(P, G, M)$ from Algorithm 10 that are unmodified are not reported in Algorithm 11. Only modified procedures are shown in Algorithm 11. The first rule is implemented from line 6 to line 8. It updates $\overline{S}(i, a)$ whenever a target node a exits $M[i]$. The procedure `NegPropag` applies the rule condition from equation 3.10. The amortized complexity of procedure `Propagate_M` is $O(n_p n_t^2)$, as the procedure is called at most $O(n_p n_t)$ and in the worst case all target nodes $N_G(a) \cup N_{\overline{G}}(a)$ are scanned. The fifth rule is implemented from line 13 to line 15. The second rule is implemented in the new procedure `Propagate_Arc_Enters_G`. Whenever an arc (a, b) enters G, the structure $\overline{S}(i, a)$ is updated and rule 3.10 is applied. The procedure has an amortized complexity of $O(n_p^2 n_t)$, since it can be called at most $O(n_p)$ times, $M[i]$ is scanned in $O(n_t)$, and `Propag` and `NegPropag` cost $O(d_p + \overline{d_p}) = O(n_p)$ in the worst case. The procedure `Propagate_Arc_Exits_P` has an amortized complexity of $O(m_p n_t)$, since it can be called $O(m_p)$ times, and it scans each time $M[i]$ in $O(n_t)$. The lines 30 and 31 cost $O(n_p n_t)$ since $\overline{S}(i, a) = 0$ happens at most $O(n_p n_t)$. Hence the total amortized complexity of the propagation for $MC(P, G, M) \wedge MC(Pc, Gc, M)$ costs $O(n_p n_t^2)$.

3.6.4 Induced constraints

If the constraint $P \subseteq^* p$ (that is P is induced on p) is used, the morphism constraint can be rewritten as:

$$S_{ub}(i, a) = |M[i] \cap N_{\overline{G}}(a)| = 0 \\ \Rightarrow i \in Nodes(\underline{P}) \wedge M[j] \leftarrow M[j] \setminus \{a\} \ \ \forall\, j \in N_p(i). \quad (3.11)$$

Propagation can be done using $N_{\overline{P}}(i)$ instead of $N_{\underline{P}}(i)$ (for example in line 27 of Algorithm 10) and using $N_{\overline{P_c}}(i)$ instead of $N_{\underline{P_c}}$ (for example in

line 34 of Algorithm 11). This affords a better propagation. Moreover, there is no need to check for any arc (i, j) to enter or to exit P, since the check for the entrance or exit of any node i from P is sufficient. More interestingly, if the constraint $G \subseteq^* g$ is added, the morphism constraint can be rewritten as:

$$\begin{aligned} S_{ub}(i, a) = |M[i] \cap N_g(a)| = 0 \\ \Rightarrow i \in Nodes(\underline{P}) \wedge M[j] \leftarrow M[j] \setminus \{a\} \ \ \forall\, j \in N_p(i). \end{aligned} \quad (3.12)$$

The rules concerning the arcs of G can be ignored. This is interesting because those rules dominated the complexity of Algorithm 10 and 11. By using induced graphs, the complexity of Algorithm 10 is $O(n_p n_t d_p)$ instead of $O(m_t n_p)$, since the procedure `Propagate_Arc_G` can be discarded. Regarding Algorithm 11, its complexity is still $O(m_t n_t^2)$ (because of `Propagate_M`), but the procedures implementing rules concerning G are discarded.

3.6.5 Forward Checking

The direct application of the definition of forward checking to MC leads to poor pruning, since we have to wait for two of the variables P,G,M to be instantiated. Hence we define the forward checking of MC based on the instantiation of particular $M[i]$. The rules are applied only if the $M[i]$ are assigned.

Algorithm 12 implements forward checking for MC. The first rule and the fifth rule are implemented in `Propagate_M`. The procedure waits for $M[i]$ to be instantiated to $\{a\} \neq \perp$, and propagates to every mandatory neighboor of i. The fifth rule for a pattern arc (i, j) is applied only when $M[i]$ and $M[j]$ are instantiated to a target node (line 4 to 7). The second rule is implemented in the procedure `Propagate_Arc_Exits_G`. Whenever an arc (a, b) exits G, all $M[i]$ are checked to be instantiated to a. If it is the case, the target node b is removed from the mandatory neighbors of i. The third rule does not apply in our forward checking procedure. Indeed, whenever a node i enters P, we have to wait for the $M[i]$ to be instantiated, which is exactly what the forward checking version of the first rule does. Hence the procedure `Propagate_Arc_P` is discarded. The fourth rule is implemented in the procedure `Propagate_Arc_Enters_P`. Whenever an arc (i, j) enters P, we check for $M[i]$ to be instantiated to a target node a. If it is the case, the target node b is removed from the neighboor j of $M[i]$.

Algorithm 11: Propagating $MC(P, G, M) \wedge MC(Pc, Gc, M)$.

```
1  procedure Propagate_M(i,a)
2  // Element a ≠⊥ exits from M[i]
3  for b ∈ N_Ḡ(a) do
4      S(i,b) ← S(i,b) − 1
5      Propag(i,b)
6  for b ∈ N_Ḡc(a) do
7      S̄(i,b) ← S̄(i,b) − 1
8      NegPropag(i,b)
9  if M[i] = {u} ≠⊥ then
10     for j ∈ N_P(i) s.t. M[j] = {v} ≠⊥ do
11         P ← P ∪ (i,j)
12         G ← G ∪ (M[i], M[j])
13     for j ∈ N_Pc(i) s.t. M[j] = {v} ≠⊥ do
14         P ← P \ (i,j)
15         G ← G \ (M[i], M[j])
16 procedure Propagate_Arc_Enters_G(a,b)
17 // Arc (a,b) enters G
18 for i ∈ Nodes(P̄) do
19     if b ∈ M[i] then
20         S̄(i,b) ← S̄(i,b) − 1
21         NegPropag(i,b)
22 procedure Propagate_Node_P(i)
23 // Node i enters P
24 for a ∈ M[i] do
25     Propag(i,a)
26     NegPropag(i,a)
27 procedure Propagate_Arc_Exits_P(i,j)
28 // Arc (i,j) exits P
29 for a ∈ M[i] do
30     if S̄(i,a) == 0 then
31         M[j] ← M[j] \ {a}
32 procedure NegPropag(i,b)
33 if S̄(i,b) == 0 ∧ i ∈ Nodes(P_c) then
34     for j ∈ N_Pc(i) do M[j] ← M[j] \ b
```

The amortized complexity of Algorithm 12 is dominated by the procedure `Propagate_Arc_Exits_G`. This procedure can be called at most $O(m_t)$ times, and propagates at most $O(n_p)$ times, leading to a complexity of $O(m_t n_p)$. When the constraint $G \subseteq^* g$ is added, the amortized complexity of the forward checking version is the amortized complexity of the `Propagate_M` procedure, that is $O(n_p d_p)$.

Algorithm 12: Forward Checking $MC(P, G, M)$.

```
1  procedure Propagate_M(i,a)
2  // M[i] is instantiated to {a} ≠⊥
3  Propag(i,a)
4  if M[i] = {u} ≠⊥ then
5      for j ∈ N_P(i) s.t. M[j] = {v} ≠⊥ do
6          G ← G ∪ (M[i], M[j])
7          P ← P ∪ (i, j)
8  procedure Propagate_Arc_Exits_G(a, b)
9  // Arc (a, b) exits G
10 for i ∈ Nodes(P) do
11     if M[i] = {a} ≠⊥ then
12         M[j] ← M[j] \ {b}
13 procedure Propagate_Arc_Enters_P(i, j)
14 // Arc (i, j) enters P
15 if M[i] = {a} then
16     M[j] ← M[j] ∩ N_Ḡ(a)
17 procedure Propag(i, b)
18 for j ∈ N_P(i) do
19     M[j] ← M[j] ∩ N_Ḡ(b)
```

3.7 Experimental results

The critical question is to know if our framework can compete with dedicated algorithms. Should our framework be considered as a pure modeling tool or as an effective tool for graph matching ? Regarding the different types of matching defined, we selected the subgraph isomorphism problem. Indeed, efficient dedicated constraints to solve the isomorphism problem have been developed in constraint programming

[SS08], and constraints have also been proposed for the maximum common subgraph [Rég03]. The performance of our framework is compared to the performance of the `vflib` dedicated algorithm [FSV01a, CFSV99] over the subgraph isomorphism problem and the induced subgraph isomorphism problem.

The CP(Graph+Map) framework has been implemented over the `Gecode` system (`http://www.gecode.org`). This includes graph variables and propagators, map variables and propagators, together with matching propagators. Regarding the $MapVar$ constraint, the propagation over M of Section 3.3.5 was not implemented.

Moreover, we use the induced morphism propagator for the constraint $MC(P, G, M)$, for the subgraph isomorphism problem. We use the induced morphism propagator for the constraints $MC(P, G, M) \wedge MC(P_c, G_c, M)$, but we use only forward checking for the $MC(P_c, G_c, M)$ constraint. An adaptation of the local alldiff constraint (see Section 2.5.2) is also used in the model.

Our benchmark set consists of graphs made of different topological structures as explained in [LV02]. These graphs were generated using the Stanford GraphBase [Knu93], consisting of 1225 undirected instances, and 405 directed instances. The graphs range from 10 to 125 nodes for undirected graphs, and from 10 to 462 for directed graphs.

The experiments consist in performing partial subgraph isomorphism over the 1225 undirected instances, and induced subgraph isomorphism over the 405 instances. All solutions are searched. Searching for a single solution would bias the results because of the heuristics. Indeed, one of the algorithm could find a solution by chance because of the choice points made during the search. Moreover, it is difficult to ensure that algorithms use the same heuristics, as they use different levels of consistency: equivalent choices may not be available. Following the methodology used in [LV02], we ran the competing algorithms for five minutes on each of the problem instances. A run is called *solved* if it finishes under five minutes or *unsolved* otherwise. All benchmarks were performed on an Intel Xeon 3 Ghz.

Table 3.2 shows the experimental results. We report the percentage of solved instances (sol.), the total running time (tot.T), the mean running time (av.T) and memory (av.M) and the mean running time and memory over instances solved by both approaches (resp. "av.T com." and "av.M com.").

The CP(Graph+Map) model solves more problem instances than the

specialized `vflib` algorithm. This difference is significant for subgraph monomorphism (61% vs. 48%). It is interesting to notice that around 4% of the instances solved by `vflib` were not solved by our CP model. This shows that on some instances, standard algorithms can be better, but that globally, CP(Graph+Map) solves more instances. It is clear that the CP approach consumes more memory. The comparison of the average time is clearly in favor of CP(Graph+Map) as it solves more instances. It is more interesting to compare the mean execution time on the commonly solved instances. This shows that the time overhead induced by the CP approach is minimal on the commonly solved instances : about 9% for monomorphism over undirected graphs and 22% for isomorphism over directed graphs.

Regarding memory consumption, the CP framework tends to use more memory than vflib, as expected in Chapter 2.

We conclude that our approach is beneficial to someone willing to pay an average time overhead of 9% to 22% on "simple" instances to be able to solve a fourth of the instances of the benchmark which cannot be solved in the time limit by the other method. The low memory consumption is a clear advantage to dedicated algorithms such as `vflib`.

Another important question to answer is how is performing our framework for subgraph isomorphism against a direct CP model of subgraph isomorphism.

We thus compared a standard subgraph isomorphism CP model against CP(Graph+Map). The standard CP model uses finite domain variables, a forward checking alldiff constraint, and arc-consistent morphism constraints. The CP(Graph+Map) model used is equivalent regarding the set of constraints and their level of consistency.

We propose a second set of experiments in order to measure the factor lost due to the use of our framework. Random sparse graph instances were selected from the vflib graph matching database [FSV01b]. Given the size of the target graph, the pattern graphs have a size of 20% of the target graph. Moreover, the graphs are generated with a probability to have an edge between two nodes of 0.01. For each size of the target graph, there are 100 subgraph isomorphism instances. Target graphs have increasing size of 200, 400, 800, and 1600 nodes. Note that we generated the instances with target graphs of 1600 nodes (following [FSV01b]), as there are no instances in the database beyond a target graph of size of 1000 nodes. For each size, 10 instances were executed using each model. A time limit of 30 minutes was used, and 2 instances

Partial SIP over undirected graphs (5 min. limit)						
	solved	tot.T min	av.T sec	av.M kb	av.T com. sec	av.M com. kb
vflib	48%	3273	160	11.91	4.96	97.6
CP(Graph+Map)	61%	2479	121	9115.46	5.43	8243
Induced SIP over directed graphs (5 min. limit)						
	solved	tot.T min	av.T sec	av.M kb	av.T com. sec	av.M com. kb
vflib	92%	181	26.95	114.28	4.11	4.22
CP(Graph+Map)	96%	109	16.22	2859.85	5.04	2754

Table 3.2: Comparison of our CP framework against vflib on subgraph isomorphism problems.

of the 1600 set ran out of time. A limit of 10^6 was also set on the maximum number of solutions to be found.

This set of instances is challenging for CP(Graph+Map), because there is a high number of solutions, and few fails. Most of the time is spent enumerating the solutions (see Table 3.3). Thus it is expected that a simple CP model, minimizing the overhead of managing memory and updating the structure, should perform better.

The results of those experiments are described in Table 3.4 and 3.5. Table 3.4 reports the average time and the standard deviation for the standard CP model and the CP(Graph+Map) model, together with the lost due to our framework. The times reported are averaged over ten instances. The lost reported is clearly not significant, less than 1%. Table 3.5 reports the average memory and the standard deviation for the standard CP model and the CP(Graph+Map) model, together with the lost. Once again, the memory consumption is averaged over ten instances. The reported memory lost is less than 1%.

This second set experiments show that using CP(graph+map) for subgraph isomorphism creates no overhead, but still the expressiveness is available.

3.8 Conclusion

In this chapter, we showed how the integration of two domains of computation over countable structures, *graphs*[DDD05] and *maps*, [Ger97],

Table 3.3: Comparison of the number of solutions and the number of fails. Note that for the size 1600, the number of fails is different because some instances were not solved within the timelimit.

	#sol		std cp (#fails)		cp(graph+map) (#fails)	
size	avg	stdd	avg	stdd	avg	stdd
200	585461	473893	1429.1	3986.29	1429.1	3986.29
400	914464	270489	255.8	147.53	255.8	147.53
800	10^6	0	309.2	129.95	309.2	129.95
1600	4680.67	13607.3	1066.33	799.75	1288.78	475.92

Table 3.4: Comparison of the time (in seconds) averaged over ten instances for increasing size of the target graph. The pattern graph have a size of 20% of the target graph.

	std cp		cp(graph+map)		
size	avg	stdd	avg	stdd	lost (%)
200	8.02	6.58	8.53	6.97	.064
400	18.98	6.07	20.8	6.53	.096
800	46.65	22.36	51.08	22.23	.095
1600	987.95	665.01	992.11	663.1	.004

Table 3.5: Comparison of the memory consumption averaged over ten instances for increasing size of the target graph. The pattern graph has a size of 20% of the target graph.

	std cp		cp(graph+map)		
size	avg	stdd	avg	stdd	lost (%)
200	5433.6	2743.22	5509.6	2463.97	.014
400	15386.8	104.24	15624	117.24	.015
800	49034.8	106.87	49302.8	115.05	.005
1600	183124	1028.88	183398	1042.15	.001

can be used for modeling and solving a wide spectrum of graph matching problems with any combination of the following properties : monomorphism or isomorphism, graph or subgraph matching, exact or approximate matching (user-specified approximation [ZDD05]). To achieve this, we needed to generalize the map variables with non-fixed source and target sets (of the Cardinal kind [Aze07]).

We showed how a single constraint able to use both fixed and non-fixed graph variables is sufficient to model all these graph matching problems. Furthermore we showed that this constraint programming approach is competitive with the state-of-the-art algorithm for subgraph isomorphism vflib based on the Ullman graph matching algorithm; by solving substantially more instances (our approach solves more complex instances) and requiring a small overhead over the simple instances. Moreover, our framework performs as well as a direct CP model.

The next steps is to design new propagators to achieve stronger consistencies for the subgraph isomorphism problem. This is the focus of the next chapter.

4

Global constraints based on graph structures[1]

4.1 Introduction

In this chapter, we introduce a new filtering algorithm for the subgraph isomorphism problem. Instead of considering the conjunction of constraints:

$$\begin{aligned} & alldiff(x_1, \ldots, x_{n_p}) \\ & \wedge \forall\, (u,v) \in V_p \times V_p :\ ((u,v) \in E_p \Rightarrow (x_u, x_v) \in E_t), \end{aligned} \tag{4.1}$$

we consider the *global* semantic of the subgraph isomorphism problem:

$SIP(p, G, M) \equiv M$ is a subgraph isomorphism between p and G.

Inside the declarative framework of Chapter 3, the $SIP(p, G, M)$ constraint is viewed as a redundant constraint that is added to the $BijMC(p, G, M)$ constraint (implementing equation 4.1 for graph and map variables), and our new filtering algorithm prunes over p and $(lub(G))$. It can also be applied to classical models described in Section 2.5.1.

Our new filtering algorithm exploits the global structure of the graph to achieve a stronger partial consistency. This work takes inspiration from the partition refinement procedure used in Nauty [McK81] and

[1] Part of this chapter has been published in [ZDS+07].

Saucy [PTDM04] for finding graph automorphisms: the idea is to label every node by some invariant property, such as node degrees, and to iteratively extend labels by considering labels of adjacent nodes. Similar labelings are used in [SS04b, SS06, SS08] to define filtering algorithms for the graph isomorphism problem: the idea is to remove from the domain of a variable associated to a node v every node the label of which is different from the label of v. The extension of such a label-based filtering to subgraph isomorphism problems mainly requires to define a partial order on labels in order to express compatibility of labels for subgraph isomorphism: this partial order is used to remove from the domain of a variable associated to a node v every node the label of which is not compatible with the label of v. We show that this extension is more effective on difficult instances of scale free graphs than state-of-the-art subgraph isomorphism algorithms and other CP approaches.

Section 4.2 describes the theoretical framework of our filtering: it first introduces the concept of labeling, and shows how labelings can be used for filtering; it then shows that labelings can be iteratively strengthened by adding information from labels of neighbors. Section 4.3 introduces the practical framework and describes how to compute a label strengthening. An exact algorithm as well as an approximate version are provided. Experimental results are described in Section 4.4.

In the following, we assume $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$ to be the underlying instance of subgraph isomorphism problem. We also define $Node = V_p \cup V_t$, $Edge = E_p \cup E_t$, $n_p = \#V_p$, $n_t = \#V_t$, $n = \#Node$, d_p and d_t the maximal degree of the graphs G_p and G_t, and $d = max(d_p, d_t)$. We also assume that the graphs are undirected. The proposed framework can be extended to directed graphs as discussed in Section 4.4.

4.2 Theoretical Framework

This section introduces a new filtering algorithm for subgraph isomorphism. We will show in the next section how filtering can be achieved in practice from this theoretical work.

4.2.1 Subgraph Isomorphism Consistent Labelings

Definition 4.1. A *labeling* l is defined by a triple $(\mathbb{L}, \preceq, \alpha)$ such that

- $\mathbb{L}$ is a set of labels that may be associated to nodes;

- $\preceq \subseteq \mathbb{L} \times \mathbb{L}$ is a partial order on $\mathbb{L}$;
- $\alpha : \mathit{Node} \rightarrow \mathbb{L}$ is a total function assigning a label $\alpha(v)$ to every node v.

A labeling induces a compatibility relation between nodes of the pattern graph and the target graph.

Definition 4.2. The set of *compatible couples of nodes* induced by a labeling $l = (\mathbb{L}, \preceq, \alpha)$ is defined by $CC_l = \{(u, v) \in V_p \times V_t \mid \alpha(u) \preceq \alpha(v)\}$

This compatibility relation can be used to filter the domain of a variable x_u associated with a node u of the pattern graph by removing from it every node v of the target graph such that $(u, v) \notin CC_l$.

The goal of this work is to find a labeling that filters domains as strongly as possible *without removing solutions* to the subgraph isomorphism problem, *i.e.*, if a node v of the pattern graph may be matched to a node u of the target graph by a subgraph matching function, then the label of v must be compatible with the label of u. This property is called subgraph isomorphism consistency.

Definition 4.3. A labeling l is *subgraph isomorphism consistent* (SIC) iff for any subgraph matching function f, we have $\forall v \in V_p, (v, f(v)) \in CC_l$.

In the context of graph isomorphism, such as in [McK81], as opposed to subgraph isomorphism studied here, an SIC labeling is often called an *invariant*. In this case, the partial ordering is replaced by an equality condition: two nodes are compatible if they have the same label.

Many graph properties, that are "invariant" to subgraph isomorphism, may be used to define SIC labelings such as, e.g., the three following SIC labelings:

- $l_{\mathit{deg}} = (\mathbb{N}, \leq, \mathit{deg})$ where deg is the function that returns node degree;
- $l_{\mathit{distance}_k} = (\mathbb{N}, \leq, \mathit{distance}_k)$ where $\mathit{distance}_k$ is the function that returns the number of nodes that are reachable by a path of length smaller than k;
- $l_{\mathit{clique}_k} = (\mathbb{N}, \leq, \mathit{clique}_k)$ where clique_k is the function that returns the number of cliques of size k that contain the node.

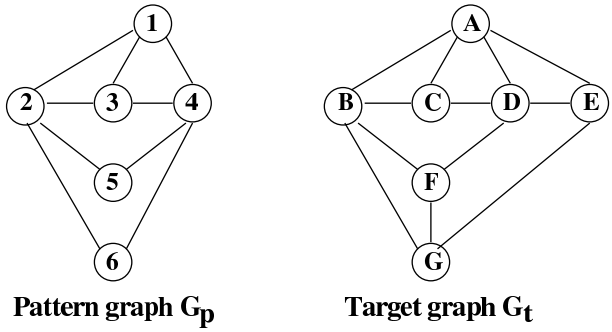

Figure 4.1: Instance of subgraph isomorphism problem.

Example

Let us consider for example the subgraph isomorphism problem displayed in Fig. 4.1. Note that this instance has no solution as G_p cannot be mapped into a subgraph of G_t. The labeling $l_{deg} = (\mathbb{N}, \leq, deg)$ assigns the following labels to nodes.

$$\begin{aligned} deg(A) = deg(B) = deg(D) = deg(2) = deg(4) &= 4 \\ deg(C) = deg(E) = deg(F) = deg(G) = deg(1) = deg(3) &= 3 \\ deg(5) = deg(6) &= 2 \end{aligned}$$

Hence, the set of compatible couples induced by this labeling is

$$\begin{aligned} CC_{l_{deg}} &= \{(u,v) \mid u \in \{2,4\}, v \in \{A,B,D\}\} \\ &\cup \{(u,v) \mid u \in \{1,3,5,6\}, v \in \{A,B,C,D,E,F,G\}\} \end{aligned}$$

This set of compatible couples allows one to remove values C, E, F and G from the domains of the variables associated with nodes 2 and 4.

4.2.2 Strengthening a Labeling

We propose to start from an elementary SIC labeling that is easy to compute such as the l_{deg} labeling defined above, and to iteratively strengthen this labeling. The strength of labelings is defined with respect to the induced compatible couples as follows.

Definition 4.4. Let l and l' be two labelings. *l' is strictly stronger than* l iff $CC_{l'} \subset CC_l$, and *l' is equivalent to* l iff $CC_{l'} = CC_l$.

A stronger labeling yields a better filtering, as it contains less compatible couples.

To strengthen a labeling, the idea is to extend the label of a node by adding information from the labels of its neighbors. This information is a multiset (as several neighbors may have the same label). We shall use the following notations for multisets.

Definition 4.5. Given an underlying set A, a *multiset* is a function $m : A \to \mathbb{N}$, such that $m(a)$ is the multiplicity (i.e., the number of occurrences) of a in m. The multiset m can also be represented by the bag $\{a_0, \ldots, a_0, a_1, \ldots\}$ where elements are repeated according to their multiplicity.

For example, the multiset m that contains 2 occurrences of a, 3 occurrences of b, and 1 occurrence of c is defined by $m(a)$=2, $m(b)$=3, $m(c)$=1, and $\forall x \notin \{a, b, c\}, m(x)$=0. This multiset may also be represented by $\{a, a, b, b, b, c\}$.

Given a partial order on a set A, we extend the partial order to multisets over A as follows.

Definition 4.6. Given two multisets m and m' over a set A, and a partial order $\preceq \subseteq A \times A$, we define $m \preceq m'$ iff there exists a total injective mapping $t : m \to m'$ such that $\forall a_i \in m, a_i \preceq t(a_i)$.

In other words, $m \preceq m'$ iff for every element of m there exists a different element of m' which is greater or equal. For example, if we consider the classical ordering on $\mathbb{N}$, we have $\{3, 3, 4\} \preceq \{2, 3, 5, 5\}$, but $\{3, 3, 4\}$ is not comparable with $\{2, 5, 6\}$. Note that comparing two multisets is not trivial in the general case, especially if the order relation on the underlying set A is not total. This point will be handled in the next section.

We now define the labeling extension procedure.

Definition 4.7. Given a labeling $l = (\mathbb{L}, \preceq, \alpha)$, the *neighborhood extension* of l is the labeling $l' = (\mathbb{L}', \preceq', \alpha')$ such that:

- every label of $\mathbb{L}'$ is composed of a label of $\mathbb{L}$ and a multiset of labels of $\mathbb{L}$, i.e., $\mathbb{L}' = \mathbb{L} \cdot (\mathbb{L} \to \mathbb{N})$;
- the labeling function α' extends every label $\alpha(v)$ by the multiset of the labels of the neighbors of v, i.e., $\alpha'(v) = \alpha(v) \cdot m$ where $\forall l_i \in \mathbb{L}$,
  $m(l_i) = \#\{u \mid (u, v) \in Edge \wedge \alpha(u) = l_i\}$;

- the partial order on the extended labels of $\mathbb{L}'$ is defined by $l_1 \cdot m_1 \preceq' l_2 \cdot m_2$ iff $l_1 \preceq l_2$ and $m_1 \preceq m_2$.

The next theorem states that the neighborhood extension of a SIC labeling is a stronger (or equal) SIC labeling.

Theorem 4.1. Let $l = (\mathbb{L}, \preceq, \alpha)$ be a labeling, and $l' = (\mathbb{L}', \preceq', \alpha')$ be its neighborhood extension. If l is an SIC labeling, then (i) l' is also SIC, and (ii) l' is stronger than or equal to l.

Proof. (i): Let f be a subgraph matching function and $v \in V_p$. We show that $\alpha'(v) \preceq' \alpha'(f(v))$, that is $\alpha(v) \preceq \alpha(f(v))$ and $m \preceq m'$, with m (resp. m') the multiset of the labels of the neighbors of v in G_p (resp. of $f(v)$ in G_t):

- $\alpha(v) \preceq \alpha(f(v))$ because l is SIC;
- $m \preceq m'$ because m' contains, for each neighbor u of v, the label $\alpha(f(u))$ of the node matched to u by the subgraph matching function f; as l is SIC, $\alpha(u) \preceq \alpha(f(u))$, and thus $m \preceq m'$.

(ii) : This is a direct consequence of the partial order on the extended labels in $\mathbb{L}'$ (Definition 4.7) : $\alpha(u) \preceq \alpha(v)$ is one of the conditions to have $\alpha'(u) \preceq \alpha'(v)$. □

Example

Let us consider again the subgraph isomorphism problem displayed in Fig. 4.1, and the labeling $l_{deg} = (\mathbb{N}, \leq, deg)$ defined in 4.2.1. The neighborhood extension of l_{deg} is the labeling $l' = (\mathbb{L}', \preceq', \alpha')$ displayed below. Note that we only display compatibility relationships $l_i \preceq l_j$ such that l_i is the label of a node of the pattern graph and l_j is the label of a node of the target graph as other relations are useless for filtering purposes.

$$
\begin{array}{rll}
\alpha'(A) & = 4 \cdot \{3,3,4,4\} & \\
\alpha'(B) = \alpha'(D) & = 4 \cdot \{3,3,3,4\} & \\
\alpha'(2) = \alpha'(4) & = 4 \cdot \{2,2,3,3\} & \preceq' 4 \cdot \{3,3,4,4\} \text{ and } 4 \cdot \{3,3,3,4\} \\
\alpha'(C) & = 3 \cdot \{4,4,4\} & \\
\alpha'(E) = \alpha'(F) & = 3 \cdot \{3,4,4\} & \preceq' 4 \cdot \{3,3,4,4\}, 3 \cdot \{4,4,4\} \text{ and } 3 \cdot \{3,4,4\} \\
\alpha'(1) = \alpha'(3) & = 3 \cdot \{3,4,4\} & \preceq' 4 \cdot \{3,3,4,4\}, 3 \cdot \{4,4,4\} \text{ and } 3 \cdot \{3,4,4\} \\
\alpha'(G) & = 3 \cdot \{3,3,4\} & \\
\alpha'(5) = \alpha'(6) & = 2 \cdot \{4,4\} & \preceq' 4 \cdot \{3,3,4,4\}, 3 \cdot \{4,4,4\} \text{ and } 3 \cdot \{3,4,4\}
\end{array}
$$

Hence, the set of compatible couples induced by this extended labeling is

$$\begin{array}{rcl} CC_{l'} & = & \{(u,v) \mid u \in \{2,4\}, v \in \{A,B,D\}\} \\ & \cup & \{(u,v) \mid u \in \{1,3,5,6\}, v \in \{A,C,E,F\}\} \end{array}$$

As compared to the initial labeling l_{deg}, this set of compatible couples allows one to further remove values B, D and G from the domains of the variables associated with nodes 1, 3, 5 and 6.

4.2.3 Iterative Labeling Strengthening

The strengthening of a labeling described in the previous section can be repeated by relabeling nodes iteratively, starting from a given SIC labeling l.

Definition 4.8. Let $l = (\mathbb{L}, \preceq, \alpha)$ be an initial SIC labeling. We define the sequence of SIC labelings $l^i = (\mathbb{L}^i, \preceq^i, \alpha^i)$ such that $l^0 = l$ and $l^{i+1} =$ neighborhood extension of $l^i \quad (i \geq 0)$.

A theoretical filter can be built on this sequence. Starting from an initial SIC labeling function $l = l^0$, we iteratively compute l^{i+1} from l^i and filter domains with respect to the set of compatible couples induced by l^i until either a domain becomes empty (thus indicating that the problem has no solution) or reaching some termination condition.

A termination condition is to stop iterating when the sequence reaches a fixpoint, *i.e.*, a step where any further relabeling cannot change the strength of the labeling. Theorem 4.2 shows that a fixpoint is reached when both the set of compatible couples and the number of different labels are not changed between two consecutive steps.

Definition 4.9. Given a labeling $l = (\mathbb{L}, \preceq, \alpha)$, we note labels($l$) the set of labels associated to nodes, i.e., labels(l) = $Codom(\alpha)$, and #labels(l) the cardinality of this set.

Theorem 4.2. Let $l = (\mathbb{L}, \preceq, \alpha)$ be a SIC labeling. The following properties hold:

1. $\forall k \geq 0, CC_{l^{k+1}} \subseteq CC_{l^k}$ and #labels(l^{k+1}) $\geq$ #labels(l^k)
2. $\forall k \geq 0$, if $CC_{l^{k+1}} = CC_{l^k}$ and #labels(l^{k+1}) = #labels(l^k), then $\forall j > k, CC_{l^j} = CC_{l^k}$ and #labels(l^j) = #labels(l^k)
3. $\exists k \geq 0$ such that $CC_{l^{k+1}} = CC_{l^k}$ and #labels(l^{k+1}) = #labels(l^k)

Proof. (1) The inclusion is a direct consequence of property (ii) in Theorem 4.1. For the cardinality of the labels, by Definition 4.7, we have $\alpha^{k+1}(v) = \alpha^k(v).m$, with m some multiset. Hence $\alpha^{k+1}(u) = \alpha^{k+1}(v)$ only if $\alpha^k(u) = \alpha^k(v)$. (2) The set of labels at step k define a partition P_k of $N_p \times N_t$. The relation between a set P_1 of P_k and a set P_2 of P_{k+1} must be either $P_1 \cap P_2 = \emptyset$ or $P_1 \subseteq P_2$. This follows from Definition 4.7: $\alpha^{k+1}(v) = \alpha^k(v).m$ implies that a node v must be in a partition $\alpha^{k+1}(v)$ finer than the partition $\alpha^k(v)$. Hence, when $\#labels(l^k) = \#labels(l^{k+1})$, the underlying partitions are the same for all subsequent steps. In particular the number of labels will never change. It remains to show that
$\forall\, u, v \in N_p \times N_t$
if $(\alpha^k(u) \preceq^k \alpha^k(v)) \Leftrightarrow (\alpha^{k+1}(u) \preceq^{k+1} \alpha^{k+1}(v))$
and $(\alpha^k(u) = \alpha^k(v)) \Leftrightarrow (\alpha^{k+1}(u) = \alpha^{k+1}(v))$
then $(\alpha^{k+1}(u) \preceq^{k+1} \alpha^{k+1}(v)) \Leftrightarrow (\alpha^{k+2}(u) \preceq^{k+2} \alpha^{k+2}(v))$.
We know that the underlying partitions and the set of compatible couples are the same for step k and step $k+1$. This means that there exists an isomorphism preserving $\preceq$ between $labels(l^k)$ and $labels(l^{k+1})$. (3) is a consequence of (1). The fixpoint is reached in at most $\#labels(l) + \#CC_l$ steps, that is $O(n_p.n_t)$. □

Theorem 4.2 gives a simple fixpoint condition. The fixpoint of the iteration is reached as soon as $\#CC_k = \#CC_{k+1}$ and $\#\text{labels}(l^{k+1}) = \#\text{labels}(l^k)$ with $k > 0$. This can be tested in $O(1)$.

Example

Let us consider again the subgraph isomorphism problem displayed in Fig. 4.1, and let us suppose that the sequence of SIC labelings is started from $l^0 = l_{deg} = (\mathbb{N}, \leq, deg)$ as defined in 4.2.1. After the first iteration, the neighborhood extension l^1 of l^0 is the labeling displayed in the example of section 4.2.2. To improve reading, we rename these labels as follows:

$$\begin{array}{rlll}
\alpha^1(A) & = 4 \cdot \{3,3,4,4\} & \equiv m_1 & \\
\alpha^1(B) = \alpha^1(D) & = 4 \cdot \{3,3,3,4\} & \equiv m_2 & \\
\alpha^1(2) = \alpha^1(4) & = 4 \cdot \{2,2,3,3\} & \equiv m_3 & \preceq^1 \{m_1, m_2\} \\
\alpha^1(C) & = 3 \cdot \{4,4,4\} & \equiv m_4 & \\
\alpha^1(1) = \alpha^1(3) = \alpha^1(E) = \alpha^1(F) & = 3 \cdot \{3,4,4\} & \equiv m_5 & \preceq^1 \{m_1, m_4\} \\
\alpha^1(G) & = 3 \cdot \{3,3,4\} & \equiv m_6 & \\
\alpha^1(5) = \alpha^1(6) & = 2 \cdot \{4,4\} & \equiv m_7 & \preceq^1 \{m_1, m_4, m_5\}
\end{array}$$

From labeling l^1, we compute the following extended labels and partial order:

$$\begin{array}{rlll}
\alpha^2(A) & = m_1 \cdot \{m_2, m_2, m_4, m_5\} & \equiv n_1 & \\
\alpha^2(B) & = m_2 \cdot \{m_1, m_4, m_5, m_6\} & \equiv n_2 & \\
\alpha^2(C) & = m_4 \cdot \{m_1, m_2, m_2\} & \equiv n_3 & \\
\alpha^2(D) & = m_2 \cdot \{m_1, m_4, m_5, m_5\} & \equiv n_4 & \\
\alpha^2(E) & = m_5 \cdot \{m_1, m_2, m_6\} & \equiv n_5 & \\
\alpha^2(F) & = m_5 \cdot \{m_2, m_2, m_6\} & \equiv n_6 & \\
\alpha^2(G) & = m_6 \cdot \{m_2, m_5, m_5\} & \equiv n_7 & \\
\alpha^2(1) = \alpha^2(3) & = m_5 \cdot \{m_3, m_3, m_5\} & \equiv n_8 & \preceq^2 \{n_1, n_3\} \\
\alpha^2(2) = \alpha^2(4) & = m_3 \cdot \{m_5, m_5, m_7, m_7\} & \equiv n_9 & \preceq^2 \{n_4\} \\
\alpha^2(5) = \alpha^2(6) & = m_7 \cdot \{m_3, m_3\} & \equiv n_{10} & \preceq^2 \{n_1, n_3, n_5, n_6\}
\end{array}$$

The set of compatible couples induced by this labeling is

$$\begin{array}{rcl}
CC_{l^2} & = & \{(2, D), (4, D), (1, A), (1, C), (3, A), (3, C)\} \\
& \cup & \{(u, v) \mid u \in \{5, 6\}, v \in \{A, C, E, F\}\}
\end{array}$$

This set of compatible couples allows one to remove values A and B from the domains of the variables associated with nodes 2 and 4. Hence, the domains of these two variables only contain one value (D), and thanks to the *alldiff* constraint on the variables, an inconsistency is detected.

4.3 Practical Framework

Algorithm 13 describes the overall filtering procedure. Starting from an initial SIC labeling, that may be, e.g., l_{deg}, this procedure first filters domains with respect to this initial labeling (lines 1–2) and then iteratively extends this labeling (lines 5–9) and filters domains with respect to the new extended labeling (lines 10-11) until some domain becomes empty, or a maximum number of iterations have been performed, or the fixpoint (see Theorem 4.2) is reached (line 4).

Labeling extension (lines 5–9) is decomposed into three steps:

- lines 5–7: α^i is computed from α^{i-1}; this step is done in $\mathcal{O}(\#Edge)$;
- line 8: labels of $\mathbb{L}^i$ are renamed; this step is done in $\mathcal{O}(d \cdot \#Node)$;

Algorithm 13: Filtering procedure

Input: two graphs $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$ such that $V_p \cap V_t = \emptyset$, an initial SIC labeling $l^0 = (\mathbb{L}^0, \alpha^0, \preceq^0)$, initial domains $D : V_p \to \mathcal{P}(V_t)$, a limit k on the number of iterations

Output: filtered domains

1 **for** *every node* $u \in V_p$ **do**
2 $\quad D(u) \leftarrow D(u) \cap \{v \in V_t \mid \alpha^0(u) \preceq^0 \alpha^0(v)\}$
3 $i \leftarrow 1$
4 **while** $\forall\, u \in V_p, D(u) \neq \emptyset$ **and** $i \leq k$ **and** *fixpoint not reached* **do**
5 $\quad$ **for** *every node* $u \in V_p \cup V_t$ **do**
6 $\quad\quad m_u^i \leftarrow$ multiset containing an occurrence of $\alpha^{i-1}(v), \forall (u,v) \in E_p \cup E_t$
7 $\quad\quad \alpha^i(u) \leftarrow \alpha^{i-1}(u) \cdot m_u^i$
8 $\quad \mathbb{L}^i \leftarrow \{\alpha^i(u) | u \in V_p \cup V_t\}$; rename labels in $\mathbb{L}^i$ and α^i
9 $\quad \preceq^i \leftarrow \{(\alpha^i(u), \alpha^i(v)) \mid u \in V_p \wedge v \in D(u) \wedge \mathit{test}(m_u^i, m_v^i, \preceq^{i-1})\}$
10 $\quad$ **for** *every node* $u \in V_p$ **do**
11 $\quad\quad D(u) \leftarrow D(u) \cap \{v \in V_t \mid \alpha^i(u) \preceq^i \alpha^i(v)\}$
12 $\quad i \leftarrow i + 1$
13 **return** D

- line 9: the partial order $\preceq^i$ is computed, i.e., for every couple of nodes (u, v) such that u is a node of the pattern graph and v is a node of the target graph which was compatible with u at step $i-1$, we test for the compatibility of the multisets m_u^i and m_v^i to determine if the labels of u and v are still compatible at step i. Testing the compatibility of two multisets is not trivial. We show in 4.3.1 how to do this exactly in $\mathcal{O}(d^{5/2})$, so that line 9 has a time complexity of $\mathcal{O}(n_p \cdot n_t \cdot d^{5/2})$. We then show in 4.3.2 how to compute an order inducing a weaker filtering in $\mathcal{O}(n_t \cdot d \cdot (n_p + d_t \cdot \log n_t))$. These two variants are experimentally compared in Section 4.4.

The filtering step (lines 10–11) is done in $\mathcal{O}(n_p \cdot n_t)$.

4.3.1 Exact computation of the partial order

Given two multisets m_u and m_v, and a partial order $\preceq$, the function $test(m_u, m_v, \preceq)$ determines if $m_u \preceq m_v$, i.e., if there exists for each label occurrence in m_u a distinct label occurrence in m_v which is greater or equal according to $\preceq$.

Property 2. Let $G = (V = (V_u, V_v), E)$ be the bipartite graph such that V_u (resp. V_v) associates a different node with every label occurrence in the multiset m_u (resp. m_v), and E contains the set of edges (i, j) such that $i \preceq j$. We have $m_u \preceq m_v$ iff there exists a matching that covers V_u in G.

Hopcroft [HK73] proposes an algorithm for solving this problem in $\mathcal{O}(|V_u| \cdot |V_v| \cdot \sqrt{|V|})$. As the sizes of m_u and m_v are bounded by the maximal degree d, the test function can be done in $O(d^{5/2})$.

4.3.2 Computation of an approximated order

If $\preceq$ is a total order, the function $test(m_u, m_v, \preceq)$ can be implemented more efficiently, by sorting each multiset and matching every label of m_u with the smallest compatible label of m_v. In this case, the complexity of *test* is $\mathcal{O}(d \cdot \log d)$.

When $\preceq$ is not a total order, one may extend it into a total order $\leq$. This total order can then be used in the *test* function to determine if $m_u \leq m_v$. However, the total order introduces new label compatibilities so that $test(m_u, m_v, \leq)$ may return true while $test(m_u, m_v, \preceq)$ returns false. As a consequence, using this approximated order may induce a weaker filtering.

In this section, we first introduce the theoretical framework that defines a new neighborhood labeling extension based on a total order and proves its validity; then we show how it can be achieved in practice.

Neighborhood labeling extension based on a total order

The next definition gives a simple condition on the total order to ensure its consistency with respect to the partial order, i.e., to ensure that $test(m_u, m_v, \preceq) = \text{True} \Rightarrow test(m_u, m_v, \leq) = \text{True}$.

Definition 4.10. Let $l = (\mathbb{L}, \preceq, \alpha)$ be a labeling. A *consistent total order for* l is a total order $\leq$ on $\mathbb{L}$ such that $\forall u \in n_p, \forall v \in n_t, \alpha(u) \preceq \alpha(v) \Rightarrow \alpha(u) \leq \alpha(v)$

We extend the order $\leq$ on multisets like for partial orders in Definition 4.6, i.e., $m \leq m'$ iff there exists an injective function $t : m \to m'$ such that $\forall a_i \in m, a_i \leq t(a_i)$. Hence, $m \preceq m' \Rightarrow m \leq m'$. Let us note however that this extension of $\leq$ to multisets only induces a partial order on multisets as some multisets may not be comparable.

We can then define a new neighborhood extension procedure, based on a consistent total order.

Definition 4.11. Let $l = (\mathbb{L}, \preceq, \alpha)$ be a labeling, and $\leq$ be a consistent total order for l. The *neighborhood extension of l based on $\leq$* is the labeling $l'_{\leq} = (\mathbb{L}', \preceq'_{\leq}, \alpha')$ where $\mathbb{L}'$ and α' are defined like in Definition 4.7, and the order relation $\preceq'_{\leq} \subseteq \mathbb{L}' \times \mathbb{L}'$ is defined by

$$l_1 \cdot m_1 \preceq'_{\leq} l_2 \cdot m_2 \text{ iff } l_1 \preceq l_2 \wedge m_1 \leq m_2$$

The next theorem shows that the neighborhood extension $l'_{\leq}$ based on $\leq$ may be used in our iterative labeling process, and that it is stronger or equal to l. However, it may be weaker than the neighborhood extension based on the partial order $\preceq$. Indeed, the total order induces more compatible couples of labels than the partial order.

Theorem 4.3. Let $l = (\mathbb{L}, \preceq, \alpha)$, $l' = (\mathbb{L}', \preceq', \alpha')$, and $l'_{\leq} = (\mathbb{L}', \preceq'_{\leq}, \alpha')$, be three labelings such that l' is the neighborhood extension of l and $l'_{\leq}$ is the neighborhood extension of l based on a consistent total order $\leq$.

If l is an SIC labeling, then (i) $l'_{\leq}$ is SIC, (ii) $l'_{\leq}$ is stronger than (or equal to) l, and (iii) l' is stronger than (or equal to) $l'_{\leq}$.

Proof. (ii) and (iii): For labeling l', we have $l_1 \cdot m_1 \preceq' l_2 \cdot m_2$ iff $l_1 \preceq l_2 \wedge m_1 \preceq m_2$. As $\leq$ is consistent w.r.t. $\preceq$, we have $m \preceq m' \Rightarrow m \leq m'$. Hence, $CC_{l'} \subseteq CC_{l'_{\leq}} \subseteq CC_l$. (i) is a direct consequence of (iii), as l' is SIC (Theorem 4.1). □ □

Different consistent total orders may be derived from a given partial order, leading to prunings of different strength: the less new couples of compatible nodes are introduced by the total order, the better the filtering. However, finding the best consistent total order is NP-hard [CH07]. Hence, we propose a heuristic algorithm that aims at computing a total order that introduces few new compatible couples without guarantee of optimality. Let us note L_p (resp. L_t) the set of labels associated with nodes of the pattern graph G_p (resp. target graph G_t). We shall suppose

without loss of generality[2] that $L_p \cap L_t = \emptyset$. The idea is to sequence the labels of $L_p \cup L_t$, thus defining a total order on these labels, according to the following greedy principle: starting from an empty sequence, one iteratively adds some labels of $L_p \cup L_t$ at the end of the sequence, and removes these labels from L_p and L_t, until $L_p \cup L_t = \emptyset$.

To choose the labels added in the sequence at each iteration, our heuristic is based on the fact that the new couples of compatible nodes are introduced by new couples of compatible labels (e_p, e_t) such that $e_p \in L_p$ and $e_t \in L_t$. Hence, the goal is to sequence as late as possible the labels of L_p. To this aim, we first compute the set of labels $e_t \in L_t$ for which the number of labels $e_p \in L_p, e_p \preceq e_t$ is minimal. To break ties, we then choose a label e_t such that the average number of labels $e'_t \in L_t, e_p \preceq e'_t$, for every label $e_p \in L_p, e_p \preceq e_t$, is minimal. Then, we introduce in the sequence the selected label e_t, preceded by every label $e_p \in L_p$ such that $e_p \preceq e_t$.

The time complexity of this heuristic algorithm is in $\mathcal{O}(n_t \cdot \log n_t \cdot d_p \cdot d_t)$.

Practical computation of an approximate partial order

In practice, one has to compute a total order $\leq^{i-1}$ that approximates the partial order $\preceq^{i-1}$ at each iteration i of Algorithm 13. This must be done between lines 7 and 8. Then each call to the test function, line 8, is performed with the total order $\leq^{i-1}$ instead of the partial order $\preceq^{i-1}$.

In this case, the time complexity of the computation of $\preceq^i$ (line 8) is in $\mathcal{O}(n_t \cdot n_p \cdot d \cdot \log d)$. This complexity can be reduced to $\mathrm{O}(n_t \cdot n_p \cdot d)$ by first sorting all the multisets. When adding the time complexity of the computation of the total order by our heuristic algorithm, we obtain an overall complexity in $\mathcal{O}(n_t \cdot d \cdot (n_p + d_t \cdot \log n_t))$.

4.3.3 Filtering within a Branch and Propagate framework

In this section, we introduce two optimizations that may be done when filtering is integrated within a branch and propagate search, where a variable assignment is done at each step of the search.

A first optimization provides an entailment condition for the filtering. If the initial labeling l^0 is such that the maximum label of the pattern

[2] If a label e both belongs to L_p and L_t, it is always possible to rename e into e' in L_t (where e' is a new label), and to add a relation $e'' \preceq e'$ for every label $e'' \in L_p$ such that $e'' \preceq e$.

graph is smaller or equal to the minimum label of the target graph, every label of nodes of the pattern graph is compatible with all the labels of nodes of the target graph so that no domain can be reduced by our filtering procedure.

A second optimization is done when, during the search, the variable associated with a pattern node is assigned to a target node. In this case, the neighborhood extension procedure is modified by forcing the two nodes to have a same new label which is not compatible with other labels as follows:

Definition 4.12. Let $l = (\mathbb{L}, \preceq, \alpha)$ be an SIC labeling, and let $(u, v) \in V_p \times V_t$ such that $v \in x_u$. The propagation of $x_u = v$ on l is the new labeling $l' = (\mathbb{L}', \preceq', \alpha')$ such that

- $\mathbb{L}' = \mathbb{L} \cup \{l_{uv}\}$ where l_{uv} is a new label such that $l_{uv} \notin \mathbb{L}$;
- $\preceq' = \preceq \cup \{(l_{uv}, l_{uv})\}$ so that the new label l_{uv} is not comparable with any other label except itself;
- $\alpha'(u) = \alpha'(v) = l_{uv}$ and $\forall w \in \text{Nodes} \setminus \{u, v\}, \alpha'(w) = \alpha(w)$

This labeling l' is used as a starting point of a new sequence of labeling extensions. Note that this propagation is done every time a domain is reduced to a singleton.

4.4 Experimental Results

Considered instances

We evaluate our approach on graphs that are randomly generated using a power law distribution of degrees $P(d = k) = k^{-\lambda}$: this distribution corresponds to scale-free networks which model a wide range of real networks, such as social, Internet, or neural networks [Bar03]. We have made experiments with different values of λ, ranging between 1 and 5, and obtained similar results. Hence, we only report experiments on graphs generated with the standard value $\lambda = 2.5$.

We have considered 6 classes of instances, each class containing 20 different instances. For each instance, we first generate a connected target graph which node degrees are bounded between d_{min} and d_{max}. Then, a connected pattern graph is extracted from the target graph by randomly selecting a percentage p_n (resp. p_e) of nodes (resp. edges).

All instances of classes A, B, and C are non directed feasible instances that have been generated with $d_{min} = 5$, $d_{max} = 8$, and $p_n = p_e = 90\%$. Target graphs in A (resp. B and C) have 200 (resp. 600 and 1000) nodes.

All instances of class D are directed feasible instances that have been generated with $d_{min} = 5$, $d_{max} = 8$, and $p_n = p_e = 90\%$. Target graphs have 600 nodes. Edges of target graphs have been randomly directed. To solve these directed instances, the filtering procedure is adapted by extending labelings with two multisets that respectively contain labels of successors and predecessors.

All instances of classes E and F are non directed instances that have been generated with $d_{min} = 20$, $d_{max} = 300$, and $p_n = 90\%$. Target graphs have 300 nodes. Instances of class E are feasible ones that have been generated with $p_e = 90\%$. Instances of class F are non feasible ones: for these instances, pattern graphs are extracted from target graphs by randomly selecting 90% of nodes and 90% of edges, but after this extraction, 10% of new edges have been randomly added.

For all experimentations reported below, each run searches for all solutions of an instance.

Comparison of different variants of our filtering algorithm

Algorithm 13 has been implemented in Gecode (http://www.gecode.org), using CP(Graph) and CP(Map) [DDD05, DDZD05] which provide graph and function domain variables. The global subgraph isomorphism constraint has been combined with $c2$ constraints (as defined in Section 4.2.1) and a global AllDiff constraint which are propagated by forward checking.

Table 4.1 compares different variants of Algorithm 13, obtained by either computing an exact partial order or an approximated one (as described in 4.3.1 and 4.3.2), and by considering different limits k on the number of iterations. In all variants, the initial labeling l^0 is the labeling l_{deg} defined in 4.2.1. Note that the order of l_{deg} is a total order so that in this case the exact and approximated variants are equivalent for $k = 1$.

Let us first compare the exact and approximated variants. The number of failed nodes with Approx./$k = 2$ is greater than Exact/$k = 2$, but it is smaller than with Exact/$k = 1$. This shows us that the total order computed by our heuristic algorithm is a quite good approximation of the partial order. When considering CPU-times, we note that

	Solved instances (%)						Average time						Average failed nodes					
	Exact			Approx.			Exact			Approx.			Exact			Approx.		
	k=0	k=1	k=2	k=2	k=4	k=8	k=0	k=1	k=2	k=2	k=4	k=8	k=0	k=1	k=2	k=2	k=4	k=8
A	100	100	100	100	100	100	2.2	**0.6**	23.4	1.3	1.9	3.2	440	14	0	13	0	0
B	100	100	100	100	100	100	61.4	**5.6**	144.2	24.5	28.7	59.6	1314	8	0	3	0	0
C	45	100	45	100	100	100	439.2	**26.3**	495.8	101.8	110.4	227.8	1750	13	0	2	0	0
D	100	100	100	100	100	100	**0.7**	2.6	99.6	7.5	24.7	56.3	2	0	0	0	0	0
E	80	60	0	75	80	**85**	126.7	98.6	-	35.2	18.8	36.7	4438	159	-	39	13	7
F	23	20	0	38	63	**68**	186.4	109.9	-	45.0	10.5	3.9	18958	3304	-	2323	481	107

Table 4.1: Comparison of different variants of Algorithm 13: Exact (resp. Approx.) refers to the implementation of *test* (line 8 of Algorithm 13) described in 4.3.1 (resp. 4.3.2); k gives the maximum number of iterations. Each line successively reports the percentage of instances that have been solved within a CPU time limit of 600s on an Intel Xeon 3,06 Ghz with 2Go of RAM; the average run time for the solved instances; and the average number of failed nodes in the search tree for the solved instances.

Approx./$k = 2$ is significantly quicker than Exact/$k = 2$.

Table 4.1 also shows that the best performing variant differs when considering different classes of instances. Instances of class D are best solved when $k = 0$, i.e., with the simple l_{deg} labeling: these instances are easy ones, as adding a direction on edges greatly narrows the search space. Instances of classes A, B and C are more difficult ones, as they are not directed; these instances are best solved when $k = 1$, i.e., after one iteration of the exact labelling extension. Instances of classes E and F, which have significantly higher node degrees, are very difficult ones. For these instances, and more particularly for those of class F which are not feasible ones and which appear to be even more difficult, iterative labeling extensions actually improve the solution process and the best results are obtained when $k = 8$.

As a conclusion, these experimentations show us that (1) the approximated variant offers a good compromise between filtering's strength and time, and (2) the optimal limit k on the number of iterations depends on the difficulty of instances. The best average results are obtained with Approx./$k = 4$.

Comparison with state-of-the-art approaches

We now compare the variant Approx./k=4 of Algorithm 13 with a state-of-the-art algorithm coming from a C++ library `vflib`, and with CP. We consider two different CP models:

- $c2$ is the model using (MC) morphism constraints described in Section 2.5.1;
- $c2 + c3$ is the model that uses additional LocalAlldiff redundant constraints introduced in [LV02] described in Section 2.5.1.

These two models are combined with a global Alldiff constraint. For $c2$ and Alldiff constraints, two levels of consistency are considered, i.e., Forward Checking (denoted by FC) and Arc Consistency (denoted by AC). Propagation of $c3$ follows [LV02]. All CP models have been implemented in Gecode using CP(Graph) and CP(Map).

Table 4.2 compares all these approaches and shows us that, except for easy instances of class D which are best solved by `vflib`, all other classes of instances are best solved by Approx./k=4. When comparing the different CP models, we note that adding redundant $c3$ constraints significantly improves the solution process except for the easy instances of class D which are better solved with simpler models.

	Solved instances (%)						Average time						Average failed nodes					
	vflib	c2		c2+c3		App.	vflib	c2		c2+c3		App.	vflib	c2		c2+c3		App.
		FC	AC	FC	AC	k=4		FC	AC	FC	AC	k=4		FC	AC	FC	AC	k=4
A	35	100	100	100	100	100	251.4	57.1	38.7	26.9	22.3	**1.9**	-	165239	19	67	0	0
B	0	0	0	0	0	**100**	-	-	-	-	-	**28.7**	-	-	-	-	-	0
C	0	0	0	0	0	**100**	-	-	-	-	-	**110.4**	-	-	-	-	-	0
D	100	100	100	5	0	100	**0.8**	7.9	81.7	542.7	-	24.7	-	2402	0	0	0	0
E	0	0	5	33	20	**80**	-	-	362.0	319.5	397.6	**18.8**	-	-	154	21	7	13
F	0	0	0	10	5	**63**	-	-	-	381.7	346.5	**10.5**	-	-	-	52	14	481

Table 4.2: Comparison of state-of-the-art approaches. Each line successively reports the percentage of instances that have been solved within a CPU time limit of 600s on an Intel Xeon 3,06 Ghz with 2Go of RAM; the average run time for the solved instances; and the average number of failed nodes in the search tree for the solved instances.

4.5 Conclusion

We introduced a new filtering algorithm for the subgraph isomorphism problem that exploits the global structure of the graph in order to achieve a stronger partial consistency. This work extends a filtering algorithm for graph isomorphism [SS06] where a total order defined on some graph property labelling is strengthened until a fixpoint. The extension to subgraph isomorphism has been theoretically founded. The order is partial and can also be iterated until a fixpoint. However, using such a partial order is ineffective. Instead, one can map this partial order to a total order. Performing such a mapping is hard, and can be efficiently approximated through a heuristic algorithm. Experimental results show that our propagators are efficient against state-of-the-art propagators and algorithms.

There are several open research interesting issues. A dynamic termination criteria for the iterative labeling should be designed. For now the number of iterations has to be fixed. The algorithm could be stopped whenever the gain of pruning is low. Moreover, the lower bound of G is ignored in the algorithm, and we could use this information in the algorithm. Some theoretical issues also exit. It concerns the exact level of consistency of our algorithm when the fixpoint is reached. It is not clear for instance if the $alldiff$ and morphism constraints can be discarded when the algorithm goes to the fixpoint.

5

Symmetries[1]

5.1 Introduction

This chapter aims at developing symmetry breaking techniques for subgraph isomorphism. Our first goal is to develop specific detection techniques for the classical variable symmetries and value symmetries, and to break such symmetries when solving subgraph isomorphism. Our second goal is to develop local symmetry detection and breaking techniques that can be easily handled for subgraph isomorphism. Symmetries arise naturally in graphs as automorphisms. However, although many graph problems have been tackled [BFL05, CB04, Sel03] and a computation domain for graphs has been defined [DDD05], and despite the fact that symmetries and graphs are related, little has been done to investigate the use of symmetry breaking for graph problems in constraint programming.

Contributions The contributions are the followings:

- We show that *all* global variable symmetries can be detected by computing the set of automorphisms of the pattern graph, and how they can be broken.
- We show that *all* global value symmetries can be detected by computing the set of automorphisms of the target graph, and how they can be broken.

[1] Part of this work has been published in [ZDD06] [ZDD07].

- Experimental results show that global symmetry breaking is an effective way to increase the number of tractable instances of the subgraph isomorphism problem.

- We show that local symmetries can be detected by computing the set of automorphisms on various subgraphs of the target graph.

- Experimental results show that local symmetries solve more difficult instances compared to global symmetries.

Applications Regarding subgraph isomorphism, the direct application is to use symmetries to speed up the search, especially when searching for all solutions. The efficiency of symmetry breaking depends on the number of symmetries in the pattern and target graph in a given instance. Thus there is no guarantee that even a single symmetry exists in a given instance. This pushes symmetry breaking subgraph isomorphism toward fields where graphs are naturally symmetric.

It was recently discovered ([GK07]) that symmetry breaking subgraph isomorphism finds an application in motif discovery. The problem of motif discovery is to find frequent subgraphs of a given graph, that is found more frequently than in a random graph. Former techniques (such as [Wer06]) enumerated the subgraphs of the given graph, leading to network centric algorithms. The authors of [GK07] obtained the best motif discovery algorithm by using a Ullmann subgraph isomorphism algorithm together with symmetry breaking. They generate all possible subgraphs of a given size and match the generated subgraph to the given graph, leading to matching centric algorithms. Using this method, they beat all former techniques. Their technique however uses only global variable symmetries. The methods developed in this chapter find thus a direct application in motif discovery.

Related Works A symmetry in a Constraint Satisfaction Problem (CSP) is a bijective function that preserves CSP structure and solutions. Symmetries are important because they induce symmetric subtrees in the search tree. If the instance has no solution, failure has to be proved for equivalent subtrees regarding symmetries. If the instance has solutions, many symmetric solutions will have to be enumerated in symmetric subtrees. The detection and breaking of symmetries can thus speed up the solving of a CSP.

Handling symmetries to reduce search space has been a subject of research in constraint programming for many years. Crawford *et al.* [CGLR96] showed that computing the set of predicates breaking the symmetries of an instance is NP-hard in general. Different approaches exist for exploiting symmetries. Symmetries can be broken during search either by posting additional constraints (SBDS) [GS01] or by pruning the tree below a state symmetrical to a previous one (SBDD) [GHK03]. Symmetries can be broken by taking into account the symmetries into the search heuristic [MT01]. Symmetries can be broken by adding constraints to the initial problem at its root node [CGLR96]. Symmetries can also be broken by remodeling the problem [Smi01].

Dynamic detection of value symmetries (also called local value symmetries or conditional value symmetries) and a general method for detecting them has been proposed in [Ben94]. The general case for such a detection is difficult. However in binary CSPs made of non-equal constraints, dominance detection for value symmetries can be performed in linear time [BS06].

Lately research efforts has been triggered towards defining, detecting and breaking symmetries. Cohen *et al.* [CJJ+05] defined two types of symmetries, solution symmetries and constraint symmetries and proved that the group of constraint symmetries is a subgroup of solution symmetries. Puget [Pug05d] showed how to detect symmetries automatically, showed that all variable symmetries can be broken with a linear number of constraints for injective problems [Pug05c], and also propose to apply this idea to SIP [Pug04]. Gent *et al.* [GKL+05] rediscovered local symmetries defined in [Ben94] and evaluated several techniques to break local symmetries. However the detection of local symmetries remains a research topic, as it not clear whether the trade off between local symmetry detection and tree search reduction is worth.

Outline Sections 5.2 provides the necessary background in symmetry breaking. Sections 5.3 and 5.4 present variable symmetries and value symmetries in subgraph isomorphism. Section 5.5 describes local symmetries for subgraph isomorphism. Experiments are discussed in Section 5.6. Finally, Section 5.7 concludes this chapter.

5.2 Background

Recall that a CSP instance is a triple $< X, D, C >$ where X is the set of variables, D is the universal domain specifying the possible values for those variables, and C is the set of constraints. In the sequel, $n = |V_p|$, $d = |D|$, and $D(x_i)$ is the domain of x_i. A symmetry over a CSP instance P is a bijection σ mapping solutions to solutions, and hence non solutions to non solutions [Pug05d]. Since a symmetry is a bijection where domain and target sets are the same, a symmetry is a permutation. A *variable symmetry* is a bijective function $\sigma : X \rightarrow X$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $\sigma s = ((\sigma(x_1), d_1), \ldots, (\sigma(x_n), d_n))$. A *value symmetry* is a bijective function $\sigma : D \rightarrow D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $\sigma s = ((x_1, \sigma(d_1)), \ldots, (x_n, \sigma(d_n))$. A *value and variable symmetry* is a bijective function $\sigma : X \times D \rightarrow X \times D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $\sigma s = (\sigma(x_1, d_1), \ldots, \sigma(x_n, d_n))$. A *global symmetry* of a CSP is a symmetry holding on the initial problem. A *local symmetry* of a CSP P is a symmetry holding only in a sub-problem $P^{'}$ of P. The conditions of the symmetry are the constraints necessary to generate $P^{'}$ from P [GKL^{+}05] [Ben94]. A *group* is a finite or infinite set of elements together with a binary operation (called the group operation) that satisfies the four fundamental properties of closure, associativity, the identity property, and the inverse property. An *automorphism of a graph* is a graph isomorphism with itself. The set of automorphisms $Aut(G)$ defines a finite group of permutations.

5.3 Variable Symmetries

In this section, we show that the set of global variable symmetries of a subgraph isomorphism CSP is the set of automorphisms of the pattern graph. Moreover, we show how existing techniques can be used to break all global variable symmetries.

5.3.1 Detection

This subsection shows that, in subgraph isomorphism, global variable symmetries are the automorphisms of the pattern graph and do not depend on the target graph. It has been shown that the set of variable

symmetries of the CSP is the automorphism group of a *symbolic graph* [Pug05d]. The pattern G_p is transformed into a symbolic graph $S(G_p)$ where $Aut(S(G_p))$ is the set of variable symmetries of the CSP.

Definition. A CSP P modeling a subgraph isomorphism instance (G_p, G_t) can be transformed into the following symbolic graph $S(P)$:

1. Each variable x_i is a distinct node labelled i.
2. If there exists a morphism constraint between x_i and x_j, then there exists an arc between i and j in the symbolic graph.
3. The constraint alldiff is transformed into a node typed with label 'a'; an arc (a, x_i) is added to the symbolic graph for each x_i.

Figure 5.1 shows a pattern transformed into its symbolic graph. If we do not consider the extra node and arcs introduced by the alldiff constraint, then the symbolic graph $S(P)$ and G_p are isomorphic by construction. Given the labelling of nodes representing constraints, an automorphism in $S(P)$ maps the alldiff node to itself and the nodes corresponding to the variables to another node corresponding to the variables. Each automorphism in $Aut(G_p)$ will thus be a restriction of an automorphism in $Aut(S(P))$, and an element in $Aut(S(P))$ will be an extension of an element in $Aut(G_p)$. Hence the two following theorems.

Theorem 1. Let (G_p, G_t) be a subgraph isomorphism instance, P its associated CSP. We have :

- $\forall\, \sigma \in Aut(G_p)\ \exists\, \sigma^{'} \in Aut(S(P)) :\ \forall\, n \in V_p : \sigma(n) = \sigma^{'}(n)$
- $\forall\, \sigma^{'} \in Aut(S(P))\ \exists\, \sigma \in Aut(G_p) :\ \forall\, n \in V_p : \sigma(n) = \sigma^{'}(n)$

Theorem 2. Let (G_p, G_t) be a subgraph isomorphism instance, P its associated CSP. The set of variable symmetries of P is the set of bijective functions $Aut(S(P))$ restricted to V_p, which is equal to $Aut(G_p)$.

The above theorem states that only $Aut(G_p)$ has to be computed in order to get all variable symmetries.

5.3.2 Breaking

Two existing techniques are relevant to our particular problem. The first technique is an approximation and consists in breaking only the generators of the symmetry group [CGLR96]. Those generators are obtained

Figure 5.1: Example of symbolic graph for a square pattern.

by an automorphism detection software such as NAUTY [McK81] . For each generator σ, an ordering constraint $s \leq \sigma s$ is posted.

The second technique breaks all variable symmetries of an injective problem by using a Schreier-Sims algorithm, provided that the generators of the variable symmetry group are known [Pug05b]. Puget showed that the number of constraints to be posted is linear with the number of variables. The Schreier-Sims algorithm computes a base and a strong generating set of a permutation group. Let G be the group, S_g the symmetry group of g elements containing G, and t the number of generators, then its complexity is in $O(g^2 log^3|G| + t.g.log|G|)$.

5.4 Value Symmetries

In this section we show how all global value symmetries can be detected and how existing techniques can be extended to break them.

5.4.1 Detection

In subgraph isomorphism, global value symmetries are automorphisms of the target graph and do not depend on the pattern graph.

Theorem 3. Let (G_p, G_t) be a subgraph isomorphism instance and P be its associated CSP. The set of global value symmetries of P is equal to $Aut(G_t)$.

Proof Suppose that $\sigma \in Aut(G_t)$, that f is a subgraph isomorphism between G_p and G_t, and that $f(i) = v_i$ for $i \in V_p$. Consider the subgraph $G = (V, E)$ of G_t, where $V = \{v_1, \ldots, v_n\}$ and $E = \{(i, j) \in E_t \mid (f^{-1}(i), f^{-1}(j)) \in E_p\}$. This means that there exists an isomorphic function $f^{'}$ matching G_p to σG. Hence $((x_1, \sigma(v_1)), \ldots, (x_n, \sigma(v_n)))$ is a solution. Suppose σ is a value symmetry of P and suppose $\sigma \notin Aut(G_t)$. This means that a solution can be mapped to a non solution, or a non

solution to a solution, which is impossible by definition of a value symmetry. ■

5.4.2 Breaking

Breaking global value symmetries can be performed by using the GE-Tree technique [RDGKL04]. The idea is to modify the labelling by avoiding symmetrical value assignments. Suppose a state S is reached, where $x_1, \ldots, x_k$ are assigned to $v_1, \ldots, v_k$ respectively, and $x_{k+1}, \ldots, x_n$ are not assigned yet. The variable x_{k+1} should not be assigned to two symmetrical values, since two symmetric subtrees would be searched. For each value $v_i \in D(x_{k+1})$ that is symmetric to a value $v_j \in D(x_{k+1})$, only one state S_1 should be generated with the new constraint $x_{k+1} = v_i$.

A convenient way to compute those symmetrical values uses the Schreier-Sims algorithm. Algorithm Schreier-Sims outputs the sets $U_i = \{k \mid \exists\, \sigma \in Aut(G_t) : \sigma(i) = k \wedge \sigma(j) = j\ \forall\ j < i\}$. A set U_i gives the images of i by the automorphisms of G mapping $0, \ldots, i-1$ to themselves. If values are assigned in an increasing order, assigning symmetrical values can be avoided by using those sets U_i. Using symmetry breaking constraints together with GE-Tree is complete and correct as shown in [Pug05b].

5.5 Local Symmetries

Global symmetries may hide symmetries arising during search. During search, variables are assigned and new variable symmetries may arise. As values are removed from domains, new value symmetries are created and can be exploited. In this section, we focus on detecting those symmetries for the subgraph isomorphism problem.

Local symmetries for subgraph isomorphism can be found through subgraphs of the initial pattern and target graphs. During the search, subgraphs of the pattern and target graph define variable and value local symmetries. We first show how to define those subgraphs, and then we explain local variable symmetry detection and local value symmetry detection.

5.5.1 Partial dynamic graphs

We first introduce partial dynamic graphs. Those graphs are associated to a state in the search and correspond to the unsolved part of the

problem. This can be viewed as a new local problem to the current state.

Definition. Let S be a state in the search.
The **partial dynamic pattern graph** $G_p^- = (V_p^-, E_p^-)$ induced by S is a subgraph of G_p such that :

- $V_p^- = \{i \in V_p \mid \exists\, j : (i,j) \in E_p \wedge \exists\, a \in D(x_i) \wedge \exists\, b \in D(x_j) \wedge (a,b) \notin E_t\}$
- $E_p^- = \{(i,j) \in E_p \mid i \in V_p^- \wedge j \in V_p^-\}$

The **partial dynamic target graph** $G_t^- = (V_t^-, E_t^-)$ is a subgraph of G_t such that :

- $V_t^- = \cup_{i \in V_p^-} D(x_i)$
- $E_t^- = \{(a,b) \in E_t \mid a \in V_t^- \wedge b \in V_t^-\}$

Those partial dynamic graphs define the local CSP corresponding to the local state.

Figure 5.2 shows an example where circled nodes are assigned to each other. The domain of the variables are $D(1) = \{k\}$, $D(2) = D(4) = \{f, j\}$, and $D(3) = \{g, c, e, i\}$. In the pattern graph, plain nodes and edges represent G_p^-. Regarding morphism constraints, dashed edges are entailed morphism constraints and plain edges are non entailed morphism constraints. In the target graph, plain nodes and edges represent G_t^- assuming a forward checking propagation for the morphism constraints.

One general way to compute local symmetries of binary CSPs is to use the microstructure of the CSP [CJJ+05]. The set of nodes of the microstructure graph is the product set of the variables and the domain. In our particular problem of subgraph isomorphism, the variables are the nodes of G_p^- and the domain is the set of nodes of G_t^-. Hence the order of the microstructure is $|G_p^- \times G_t^-|$ and can be very large. But in subgraph isomorphism, local symmetries can be computed directly in the graphs G_p^- and G_t^-, without using the microstructure.

5.5.2 Local variable symmetries

Local variable symmetries must map variables having the same domain. This fact follows directly from the definition of a variable symmetry.

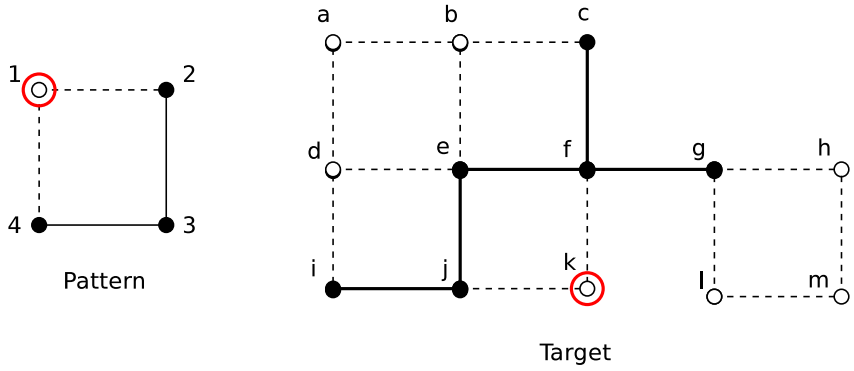

Figure 5.2: Example of local subgraphs.

This problem was not present for global variable symmetries as the initial domains are V_t. The set of automorphisms of the partial dynamic pattern graph has to be redefined.

Definition. Given a partial dynamic pattern graph G_p^-, $Aut'(G_p^-)$ is the set of automorphisms mapping a node i to a node j if and only if $D(x_i) = D(x_j)$.

The following theorem states that local variable symmetries can be obtained by computing $Aut'(G_p^-)$.

Theorem 4. Let (G_p, G_t) be a subgraph isomorphism instance, L be a state in the search space, G_p^- the partial dynamic pattern graph associated with L, and P' be the CSP associated with L. The set of variable symmetry of $P^{'}$ is $Aut'(G_p^-)$.

Proof Suppose $\sigma \in Aut'(G_p^-)$. Consider the symbolic graph $S(P')$ of P'. Recall that the alldiff constraint has no influence on $Aut(S(P'))$. All automorphisms β of $Aut(S(P'))$ are not variable symmetry of P' since domains of variables may be different in the local subproblem P'. Since $\sigma \in Aut(S(P'))$ and is restricted to map only variables with the same domains, σ is a variable symmetry of P'. Suppose that σ is a variable symmetry of $P^{'}$ and that $\sigma \notin Aut'(G_p^-)$. This means that a solution can be mapped to a non solution, or a non solution can be mapped to a solution, which is impossible by definition of a local variable symmetry. ■

Computing $Aut'(G_p^-)$ can be done as usual by using automorphism

detection software. The initial partition is refined into ordered sets containing variables having the same domain.

Breaking

Local variable symmetries can be broken by using the same technique as for global variable symmetries (Section 5.3.2). This ensures that all detected local variable symmetries are broken. However, adding breaking constraint of the form $x_i < x_j$ modifies the local symbolic graph $S(P)$. This may introduce or remove new local variable symmetries. The detection presented in the previous section is however valid. Indeed, the additional constraints $x_i < x_j$ ensure that $D(x_i) \neq D(x_j)$. Any automorphism between x_i and x_j is excluded from $Aut'(G_p^-)$.

5.5.3 Local value symmetries

The following theorem states that value symmetries of the local CSP P' can be obtained by computing $Aut(G_t^-)$ and that these symmetries can be exploited without losing or adding solutions to the initial problem.

Theorem 5. Let (G_p, G_t) be a subgraph isomorphism instance, P' be the local CSP associated with a state during the search. The set of value symmetry of $P^{'}$ is $Aut(G_t^-)$.

Proof This follows directly from Theorem 3 and the fact that (G_p^-, G_t^-) is a subgraph isomorphism instance. ■

When using graph variables inside our declarative framework, the dynamic target graph G_t^- can be computed dynamically. Given a constraint $BijMC(P, T, M)$, the dynamic target graph G_t^- is the upper bound of variable T and can be obtained in $O(1)$.

Speeding up detection

Computing directly $Aut(G_t^-)$ is correct but this computation can be fasten. Actually, all value symmetries are not possible in a local instance (G_p^-, G_t^-). Only nodes that are all present in at least one domain can be mapped to each other in a value symmetry of P'. The search tree of the automorphism algorithm can be pruned when such nodes are mapped together.

Breaking

In this subsection, we show how to modify the GE-Tree method to handle local value symmetries. Before distribution, the following actions are triggered :

1. Compute the partial dynamic target graph G_t^-.
2. The NAUTY and Schreier-Sims algorithms are called to produce the new $U_i^{'}$ sets.
3. Given a state S, a new variable and value selection can be used such that local value symmetries are broken :
   (a) a new state S_1 with a constraint $x_k = v_k$
   (b) a new state S_2 with constraints : $x_k \neq v_k$ and $x_k \neq v_j \ \forall \ j \in U_{k-1} \cup U_{k-1}^{'}$.

The only difference with the original GE-Tree method is the addition of the U'_{k-1} during the creation of the second branch corresponding to the state S_2.

An issue is how to handle the global and local structures U. In the Gecode system (http://www.gecode.org), in which the actual implementation is made, the states are copied and trailing is not needed. Thus the global structure U must not be updated because of backtracking. A single global copy is kept during the whole search process. In a state S where local values symmetries are discovered, structure U is copied into a new structure $U^{''}$ and merged with $U^{'}$. This structure $U^{''}$ shall be used for all states $S^{'}$ having S in its predecessors.

5.6 Experimental results

The objectives in this section are to assess performances of global symmetries, and performance of local symmetries against global symmetries. For local symmetries, we study the overhead of computing local symmetry information and their ability to solve more difficult instances. Moreover, we would like to know whether local symmetries can be applied on the whole search space.

The CSP model for subgraph isomorphism has been implemented in Gecode, using CP(Graph) and CP(Map) [DDD05] [DDZD05] . The

CP(Graph) framework provides graph domain variables and CP(Map) provides function domain variables. All the software is implemented in C++. The standard implementation of the NAUTY algorithm is used. We also implemented Schreier-Sims algorithm. The computation of the constraints for breaking injective problems is implemented, and GE-Tree method is also incorporated. All local symmetry techniques presented are also implemented.

Instances - The data graphs used to generate instances are from the GraphBase database containing different topologies (see Chapter 3) and has been used in [LV02]. Experiments are performed on the first 50 undirected graphs from GraphBase. The undirected set was selected because it holds potentially more symmetries than the directed graphs. This undirected set contains graphs ranging from 10 nodes to 138 nodes. All those graphs are tested for isomorphism with one another. Only subgraph isomorphism instances with a pattern graph smaller than the target graph are kept. There are 1225 instances.

Setup - All runs were performed on a dual Intel(R) Xeon(TM) CPU 2.66GHz with 2 Go of RAM. In our tests, we look for all solutions. This ensures that we measure the whole tree search reduction, and we avoid strong influence of the heuristic. As shown later in this section, the number of solved instances stabilizes for all instances after a couple of minutes. Hence a run time limit is set. A run is solved if it finishes in less than 5 minutes, unsolved otherwise. Detecting the local symmetries on the whole search space tends to be time-consuming. Hence local symmetry detection is seen as an extension of global symmetries. No detection is made when more than 3 variables are instantiated. Breaking is performed over the whole search space.

Automorphism detection time - A main concern is how much time it takes to compute the symmetries of the graphs. Regarding global symmetries, NAUTY processed each undirected graph in less than 0.02 second. All undirected graphs were processed by Schreier-Sims in less than one second, except two of them, with 4 seconds and 8 seconds. This shows a negligible time regarding symmetry detection on this set of instances.

Models - Depending on the symmetry breaking techniques, various models are selected for these experiments :

- vflib : state of the art dedicated C++ algorithm [CFSV01]
- light : simple CP model

	solved	tot.T	av.T	av.M	av.T com.	av.M com.
		min	sec	kb	sec	kb
vflib	47.1%	3329	9.35	115	9.35	92
light	51.4%	3162	17.91	2028	14.89	1391
heavy	58.94%	2584	6.57	7892	5.81	3103
global var	64%	2318	8.72	7744	2.75	2933
global value	61.2%	2479	8.45	7820	3.06	3014
global varvalue	65.7%	2197	7.35	7545	2.50	2983

Table 5.1: Detailed results for global symmetries.

  - Forward checking constraints
  - No redundant constraint (that is only the morphism constraints are considered).

- heavy : advanced CP model
  - Arc consistency
  - Redundant constraint [LV02]
- global var : heavy + global variable symmetry
- global value : heavy + global value symmetry
- global varvalue : heavy + global variable and value symmetry
- local var : heavy + local variable symmetry
- local value : heavy + local value symmetry
- glocal var : heavy + global *and* local variable symmetry
- glocal value : heavy + global *and* local value symmetry

Vflib and the light model are considered as basic models since they perform only forward checking. We call easy instances those instances that are quickly solved by vflib and the light model. Those instances do not require any arc consistent or redundant constraint.

Detailed results - We study first experimental results for global symmetries. Figure 5.3 shows the number of solved instances against time. This Figure justifies the choice of a time limit of 5 minutes, as most

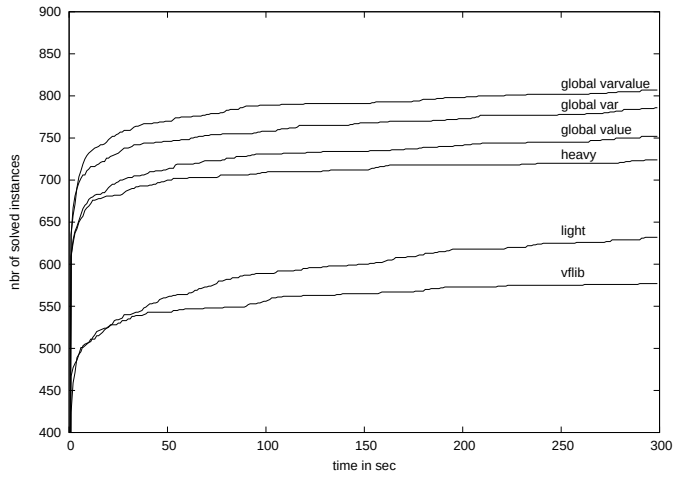

Figure 5.3: Results for global symmetries

	solved	tot.T	av.T	av.M	av.T com.	av.M com.
		min	sec	kb	sec	kb
heavy	58.94%	2584	6.57	7892	5.81	3103
glocal var	60.82%	2473	5.93	19201	4.40	3182
local var	58.45%	2615	5.88	18666	3.30	3096
glocal value	58.78%	2601	6.37	17839	3.88	3684
local value	57.14%	2682	4.96	7812	3.29	3243

Table 5.2: Detailed results for local symmetries over all instances.

of the solved instances are solved during the first 100 seconds. Hence only the percentage of solved instances is relevant. Table 5.1 shows the detailed results. The total time is the time to solve all instances, the mean time is the mean time over all solved instances, the common mean time(memory) is the mean time(memory) over instances solved by vflib. Global symmetries clearly outperforms light, heavy and vflib and improve time on easy instances and all instances. Thanks to global variable and value symmetries, 18% more instances are solved compared to vflib and all instances are solved much more efficiently.

We now study the experimental results for local symmetries. Table 5.2 shows the detailed results. The common time is still reduced,

	solved	total time	mean time	mean mem
global var	64,73%	2203 min.	4.27 sec.	11371 kb
glocal var	65,96%	2150 min.	3.26 sec.	11503 kb
local var	63,10%	2300 min.	3.15 sec.	4105 kb
global value	63,92%	2256 min.	2.94 sec.	11475 kb
glocal value	64,49%	2234 min.	3.78 sec.	28610 kb
local value	63,18%	2301 min.	3.77 sec.	4330 kb

Table 5.3: Detailed results for local symmetries with coroutining.

but local symmetries achieve the same performance as the heavy model without any symmetry technique, with the exception of local and global variable symmetries. Those results for local symmetries are due to the time needed to compute local symmetries. Actually, some easy instances are not solved with local symmetries.

In order to assess efficiency of local symmetries for difficult problems, we performed the following experiment. The light model is ran for 30 seconds, and if the instance is not solved, local symmetry models are used for 270 seconds. This coroutining setup ensures that easy instances are solved. Results for this new setup are shown in Table 5.3. We compare the results of local symmetries against global symmetries. Local symmetries slightly outperform global symmetries. Inside local symmetry models, the models combining global symmetries outperform pure local symmetry models. This is mainly because some instances contain a lot of symmetries that disappear during search. Local symmetry has a high cost but reduces time for difficult instances. Not surprisingly, local symmetries performance are poor on easy instances, but outperform global symmetry on difficult instances.

To the best of our knowledge, subgraph isomorphism with symmetry breaking achieves the best percentage of solved instances over the GraphBase benchmark proposed by [LV02].

5.7 Conclusion

In subgraph isomorphism, both global variable and value symmetries can be computed on the initial instance. Indeed, this computation can be made directly on the pattern graph and the target graph. Moreover,

all variable and value symmetries can be broken by computing a base and a strong generating set of the permutation groups thanks to the Schreier-Sims algorithm. Local variable and value symmetries can be found in a similar way. A suitable definition of the local pattern and target graphs makes the computation of local symmetries as direct as for global symmetries.

Experimental results suggest that breaking all variable and value symmetries is an efficient way to solve difficult instances. Global symmetries together with arc consistency and redundant constraints were able to solve 65% of the instances, which makes constraint programming the most efficient technique for this data set. Local symmetries achieve also good results on difficult instances. However, computing local symmetries may not be the good tradeoff between search and symmetries, especially for easy instances. Computing local symmetries during the whole search is inefficient.

Interesting directions include experiments on faster but weaker detection methods. One could search for and break generators, as the Schreier-Sims tends to be time consuming. Other weaker forms of detection could also be used. Experiments should be conducted on real-world class of graphs such as scale-free networks. Finally, it would be interesting to reproduce motif discovery experiments from [GK07], but using the whole range of global and local symmetries.

6

Decomposition

6.1 Introduction

In this chapter we study the limits of the direct application of state-of-the-art (static and dynamic) decomposition techniques for problems with global constraints; we show that such a direct application is useless for SIP. We develop a hybrid decomposition approach for such problems and design specific search heuristics for SIP, exploiting the structure of the problem to achieve decomposition. We show that the CP approach using the proposed decomposition techniques outperforms the state-of-the-art algorithms, and solves more instances on some classes of problems (sparse instances with many solutions).

Decomposition techniques are an instantiation of the divide and conquer paradigm to overcome redundant work for independent partial problems. A constraint problem (CSP) can be associated with its constraint network, which represents the active constraints together with their relationship by means of shared variables. During search, the constraint network looses structure as variables are instantiated and constraints entailed by domain propagation. The constraint network can possibly consist of two or more independent components, leading to redundant work due to the repeated computation and combination of the corresponding independent partial solutions. The key to solve this is *decomposition* that consists of two steps. The first step detects the possible problem decompositions, by examining the underlying constraint network for independent components. The second step exploits these independent components by solving the corresponding partial CSPs in-

dependently, and combines their solutions without redundant work. Decomposition can occur at any node of the search tree, i.e. at the root node or dynamically during search. In constraint programming, decomposition techniques have been studied through the concept of AND/OR search [Mat07]. AND/OR search is sensitive to problem decomposition, introducing search subtree combining AND nodes as an extension to classical OR search nodes. The size of the minimal AND/OR search tree is exponential in the tree width while the size of the minimal OR search tree is exponential in the path width, and is never worse than the size of the OR tree search.

Checking for decomposition is usually done in one of two ways. Either, only the initial constraint network is statically analyzed, resulting in a so called pseudo-tree. This structure encodes both, the static search heuristic and the information when a subproblem is decomposable [DM04]. Another possibility is to consider the dynamic changes of the constraint network by analyzing it at each node during the search [DM07]. Such a dynamic approach is better suited if a strong constraint propagation (e.g. by AC) is present but obviously to the cost of additional computations.

A major problem of decomposition techniques are their problem specificity. Without good heuristics, decomposition may occur seldom or very late such that the computational overhead for checking etc. is too high for an efficient application. Nevertheless, some approaches have been shown to be more general by applying dedicated algorithms, e.g. graph separators or cycle cutset conditioning [KK98, Mat07, MD05].

However, those (usually static) algorithms fail to compute good heuristics on problems with global constraints, which have an initially complete constraint graph. Indeed, such algorithms presuppose a sparse constraint graph. In the subgraph isomorphism problem (SIP), for example, the initial constraint graph is complete due to the presence of a global `alldiff`-constraint. This prevents cycle cutset and graph separator algorithms to be applied. A further drawback of a static analysis is the non-predictable decomposability of the constraint network achieved by constraint entailment through propagation. To exploit this, a dynamic analysis of the problem structure during the search is necessary. This is of high importance for SAT- [LvB04] and CSP-solving [DM07]. Unfortunately, a dynamic analysis requires significant additional work that slows down the search process once more.

In this chapter we show how to overcome those shortcomings by com-

bining static and dynamic decomposition approaches to take advantage of decomposition for the hard problem of SIP. A combination yields a balance between the fast static analysis and the needed full propagation exploited by dynamic search strategies in the presence of global constraints. The underlying idea is to follow the static ordering until a first problem decomposition is available (or likely) and to switch afterwards to a full propagated decomposing search. For the later, we consider only a binary constraint representation inside the constraint network in order to compute a good decomposition-enforcing heuristic. As shown in the experiments, this idea is a key point for an efficient application of a decomposing search (as AND/OR) for the SIP.

Regarding decomposition of graph matching, Valiente and al. [VM97] have shown how to use decomposition techniques in order to speed up subgraph homomorphism. [VM97] states that, if the initial pattern graph is made of several disconnected components, then matching each component separately is equivalent to matching all of them together. Specific algorithms are also demonstrated. Our work can be seen as an extension to this work. We consider the subgraph isomorphism problem instead of the subgraph homomorphism problem. The latter case is easier as the constraint graph is made only of the initial pattern graph. Moreover, we apply the decomposition dynamically when [VM97] decomposes only statically on the initial pattern graph.

Outline - The chapter is structured as follows. Section 6.2 introduces a decomposition method able to detect decomposition at any stage during the search. In Section 6.3, the proposed decomposition method is applied and specialized to SIP. Experimental results assessing the efficiency of our approach are presented in Section 6.4. Section 6.5 concludes the chapter.

6.2 Decomposition

In this section we show how to define and detect decomposition during search. Sections 6.2.1 and 6.2.2 define a decomposition method able to detect decomposition at any state during search, considering that we do not know a priori when decomposition occurs. Section 6.2.3 shows that our method is able to compute the same decompositions than the AND/OR search framework [Mat07], where the search is precomputed on a graph representation of the constraint network, and decomposition events are known in advance. The AND/OR search method has shown to

be very attractive for a large number of classes of constraint networks. But as we shall see in Section 6.3, our method is suited for the SIP while the AND/OR method is not applicable because the decomposition events cannot be precomputed.

6.2.1 Preliminary

A *Constraint Satisfaction Problem (CSP)* P is a triple $(X, \mathcal{D}, C)$ where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of domains (i.e. a finite set of values), each variable x_i is associated with a domain D_i, and C is a finite set of constraints with $scope(c) \subseteq X$ for all $c \in C$, where $scope(c)$ is the set of variables involved in the constraint c. A constraint c over a set of variables defines a relation between the variables. A *solution* of the CSP is an assignment of each variable in X to one value in its associated domain so that no constraint $c \in C$ is violated. We denote $Sol(P)$ the set of solutions of a CSP P.

A *partial CSP* $\hat{P}$ of a CSP $P \equiv (X, \mathcal{D}, C)$ is a CSP $(\hat{X}, \hat{\mathcal{D}}, \hat{C})$ where $\hat{X} \subseteq X$, $\forall \hat{D}_k \in \hat{\mathcal{D}} : \hat{D}_k \subseteq D_k$ and $\hat{C} \subseteq C$. Note that since $\hat{P}$ is a CSP, we have $scope(\hat{c}) \subseteq \hat{X}$ for all $\hat{c} \in \hat{C}$.

6.2.2 Decomposing CSPs and graphs

This subsection defines the notion of decomposition for a CSP. A CSP is decomposable into partial CSPs if the CSP and its decomposition have the same solutions.

Definition. A CSP P is *decomposable* in partial CSPs $P_1, \ldots, P_n$ iff :

- $\forall\ s \in Sol(P) : \exists\ s_1, \ldots, s_k \in Sol(P_1), \ldots, Sol(P_k) : s = \cup_{i \in [1,k]} s_i$
- $\forall\ s_1, \ldots, s_k \in Sol(P_1), \ldots, Sol(P_k) : \exists\ s \in Sol(P) : s = \cup_{i \in [1,k]} s_i$.

This general definition of decomposition can be instantiated to two practical cases. The first definition corresponds to the direct intuition of a decomposition: a CSP is decomposable if it can be split into disjoint partial CSPs. It is called 0-decomposability as no variable are shared between the partial CSPs.

Definition. A CSP $P = (X, \mathcal{D}, C)$ is *0-decomposable* in partial CSPs $P_1, \ldots, P_n$ with $P_i = (X_i, \mathcal{D}_i, C_i)$ iff $\forall\ 1 \leq i < j \leq n : X_i \cap X_j = \emptyset$, $\cup_{i \in [1,k]} X_i = X$, $\cup_{i \in [1,k]} \mathcal{D}_i = \mathcal{D}$, $\cup_{i \in [1,k]} C_i = C$.

The second definition finds more decompositions by allowing the partial CSPs to have instantiated variables in common. It is called 1-decomposability as variables shared between the partial CSPs have a domain of size 1.

Definition. A CSP $P = (X, \mathcal{D}, C)$ is *1-decomposable* in partial CSPs $P_1, \ldots, P_k$ with $P_i = (X_i, \mathcal{D}_i, C_i)$ iff $\forall\ 1 \leq i < j \leq n : x \in (X_i \cap X_j) \Rightarrow |D_x| = 1$, $\cup_{i\in[1,k]} X_i = X$, $\cup_{i\in[1,k]} \mathcal{D}_i = \mathcal{D}$, $\cup_{i\in[1,k]} C_i = C$.

The relationship with the general definition is direct. If a CSP P is *0-decomposable* or *1-decomposable* in partial CSPs $P_1, \ldots, P_k$, then P is decomposable in partial CSPs $P_1, \ldots, P_k$. From Definitions 6.2.2 and 6.2.2, it follows further :

Property 3. If a CSP $P = (X, \mathcal{D}, C)$ is 0-decomposable in $P_1, \ldots, P_k$, then P is 1-decomposable in $P_1, \ldots, P_k$. Further P might be 1-decomposable in $P'_1, \ldots, P'_{k'}$ with $k' \geq k$ via overlapping partial problems P'_i.

Redundant computation during CSP-solving is performed whenever a CSP is 0- or 1-decomposable into k partial CSPs $P_1, \ldots, P_k$. For instance, if the solutions of P_1 are computed first, then for each solution of P_1 repeatedly all solutions of $P_2, \ldots, P_k$ are computed. Therefore, $P_2, \ldots, P_k$ are solved $|Sol(P_1)|$ times and this overhead can be exponential in the size of the CSP. This can be avoided by solving the partial problems independently. The necessary detection of the CSP-decomposition into independent partial CSPs can be performed through the concept of constraint graphs.

Definition. The *constraint graph* of a (partial) CSP $P = (X, \mathcal{D}, C)$ is an undirected graph $G^P = (V, E)$ where $V = X$ and $E = \{(x_i, x_j) \mid \exists\ c \in C : x_i, x_j \in scope(c)\}$.

Note that all variables in the scope of one constraint form a clique in G^P. This constraint graph is also called the *primal graph* [Dec03]. There is a standard syntactic way of decomposing a CSP, based on its constraint graph.

Definition. A graph $G = (V, E)$ is *decomposable* into k subgraphs $G_1, \ldots, G_k$ iff $\forall_{1\leq i<j\leq k} : V_i \cap V_j = \emptyset$, $\cup_{i\in[1,k]} V_i = V$, and $\cup_{i\in[1,k]} E_i = E$.

Property 4 shows that one has to compute disjoint components of the constraint graph to detect independent CSPs. This can be done in linear time by a simple BFS.

Property 4. Given a CSP $P = (X, \mathcal{D}, C)$ with its constraint graph G, for all $k \geq 1$, the constraint graph G of P is decomposable in $G_1, \ldots, G_k$, iff P is 0-decomposable in $P_1, \ldots, P_k$ iff P is 1-decomposable in $P'_1, \ldots, P'_m$ with $m \geq k$.

Proof - The first iff is straightforward. For the second iff, we can construct a 1-decomposition $P_1, \ldots, P_m$ of P from a decomposition $G_1, \ldots, G_k$ of G, with $m \geq k$. The construction is described for the case $k = 1$ (i.e. $P_1 = P$), and can be easily generalized. Let $G = (V, E)$ be the graph constraint of P. Let $V_s = \{x \in V \mid |D_x| = 1\}$. Transform G into G' where G' is the graph G without variables with a singleton domain. More formally, $G' = (V', E')$ with $V' = V \setminus V_s$ and $E' = (V' \times V') \cap E$. Suppose G' is decomposable into $G'_1, \ldots, G'_m$ $(m \geq 1)$. Then, nodes associated to variables with a singleton domain and their associated edges are added to the G'_i, giving $G^1_i = (V^1_i, E^1_i)$. More formally $G^1_i = (V^1_i, E^1_i)$ where $V^1_i = V'_i \cup V_s$ and $E^1_i = (V^1_i \times V^1_i) \cap E$. The graphs $G^1_1, \ldots, G^1_m$ are the constraint graphs of the partial CSPs P_i of the 1-decomposition of P. ■

The above property is especially useful when $k = 1$. In this case, the 0-decomposition does not decompose the CSP, while 1-decomposition may decompose it.

6.2.3 Relationship with AND/OR search tree

Another approach to define decomposable CSPs is to use the concept of AND/OR search spaces defined with pseudo-trees [Mat07].

Definition. Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V, E')$ defined on all its nodes is called *pseudo-tree* of G if any arc of E which is not included in E' is a back-arc, namely it connects a node to an ancestor in T.

Definition. Given a CSP $P = (X, \mathcal{D}, C)$, its constraint graph G^P and a pseudo-tree T^P of G^P, the associated AND/OR search tree has alternating levels of OR nodes and AND nodes. The OR nodes are labeled x_i and correspond to variables. The AND nodes are labeled $< x_i, v_k >$ and correspond to assignment of the values v_k in the domains of the variables. The root of the AND/OR search tree is an OR node, labeled with the root of the pseudo-tree T^P. The children of an OR node x_i are AND nodes labeled with assignments $< x_i, v_k >$, consistent along the path from the root. The children of an AND node $< x_i, v_k >$ are OR nodes labeled with the children of variable x_i in T^P.

Semantically, the OR states represent alternative solutions, whereas the AND nodes represent the problem decompositions into independent partial problems, all of which need to be solved. When the pseudo-tree is a chain, the AND/OR search tree coincides with the regular OR search tree.

Following the ordering induced by the given pseudo-tree T^P of the constraint graph of a CSP P, the notion of 1-decomposability coincides with the decompositions induced by an AND/OR search.

Property 5. Given a CSP $P = (X, \mathcal{D}, C)$, a pseudo tree T^P over the constraint graph of P and a path p of length l ($l \geq 1$) from the root node of T^P to an AND node p_l, the CSP P where all variables in the path p are assigned is 1-decomposable into $P_1, \ldots, P_k$ where k is the number of OR successors in T^P of the end node p_l.

Proof - Let $y_1, \ldots, y_k$ ($k \geq 1$) be the OR successor nodes of the end node p_l in T^P. We note $tree(y_i)$ the tree rooted at y_i in T^P. Let $X_s = \{v \in X | v \in p\}$. Then build the partial CSPs $P_i = (X_i, \mathcal{D}_i, C_i)$ ($i \in [1, k]$):

$$\begin{aligned} X_i &= X_s \cup \{v \in X \mid v \in tree(y_i)\} \\ \mathcal{D}_i &= \{D_x \in \mathcal{D} \mid x \in X_i\} \\ C_i &= \{c \in C \mid scope(c) \subseteq X_i\}. \end{aligned}$$

It is clear that $\cup_{i \in [1,k]} C_i = C$ since there exists no constraint between two different $tree(y_i)$ in T^P, by definition of a pseudo tree. ■

As will be explained in the next section, neither static nor dynamic AND/OR search is suited for our particular problem. In SIP, the constraint graph is complete, and thus the pseudo tree is a chain, leading to an AND/OR search tree equivalent to an OR search tree. However the CSP P becomes 1-decomposable during search and a *dynamic* framework is needed in order to check decomposition on any state during the search. But this is computationally very expensive as we will show in Section 6.4.

6.3 Applying decomposition to SIP

The CSP model $P = (X, \mathcal{D}, C)$ of subgraph isomorphism should represent a total function $f : N_p \rightarrow N_t$. This total function can be modeled

with $X = x_1, \ldots, x_n$ with x_i a FD variable corresponding to the ith node of G_p and $D(x_i) = N_t$. The injective condition is modeled with an `alldiff`$(x_1, \ldots, x_n)$ global constraint. The isomorphism condition is translated into a set of n constraints $MC_i \equiv (x_i, x_j) \in E_t$ for all $x_i \in N_p$. Given the above modelling, the constraint graph of the CSP, called the SIP constraint graph, is the graph $G^P = (N^P, E^P)$ where $N^P = X$ and $E^P = E_p \cup E_{\neq}$. Note, E_p is representing all propagations of the MC_i constraints while $E_{\neq}$ depicts the global `alldiff`-constraints, i.e. a clique ($E_{\neq} = N_p \times N_p$). Therefore, the SIP-CSP consists of global constraints only that would prevent decomposition using a static AND/OR search. Implementation, comparison with dedicated algorithms, and extension to subgraph isomorphism and to graph and function computation domains can be found in [ZDD05].

6.3.1 Decomposing SIP

This subsection explains how to decompose the SIP problem. We first show why static AND/OR search fails by studying the SIP constraint graph.

Static AND/OR Search: Because of the `alldiff`-constraint, the SIP constraint graph corresponds to the complete graph $K_{|N_p|}$. The pseudo-tree computed on the constraint graph of any SIP instance is a chain, detecting no decomposition at all. Moreover, the initial SIP constraint graph is not 1-decomposable. Therefore a static analysis of the SIP-CSP yields no decomposition at all and is not applicable.

Decomposition seems difficult to achieve. However, as variables are assigned during search, 1-decomposition may occur at some nodes of the search tree. A dynamic detection of 1-decomposition at different nodes of the search tree gives a first way of detecting decomposition for the SIP.

Dynamic AND/OR Search: A dynamic analysis of the SIP constraint graph, as done for dynamic AND/OR search, takes care of possible constraint entailments and propagation results. It is therefore very useful for a strongly propagated CSP. The main drawback is the slow down due to the additional propagation and dynamic decomposition checks. Further, the SIP constraint graph is still a complete one and does not allow for decomposition.

Our 1-decomposition removes assigned variables in the decomposition process. One could also remove entailed constraints, leading thus to more decomposition. This can easily be done for the `alldiff`-constraint by removing an edge $(x_i, x_j) \in E^P$ representing $x_i \neq x_j$ when $D_i \cap D_j = \emptyset$ $(i \neq j)$. In the following, we redefine the constraint graph of a SIP as a constraint graph for the morphism constraints together with a dynamic constraint graph of the `alldiff`-constraint.

Definition. Given the CSP $P = (X, \mathcal{D}, C)$ of a SIP instance, its *SIP constraint graph* is the undirected graph $G = (V, E^{MC} \cup E^{\neq})$, where $V = X$, $E^{MC} = \{(x_i, x_j) \in E_p \mid x_i, x_j \in X\}$ and $E^{\neq} = \{(i, j) \in X \times X \mid D_i \cap D_j = \emptyset\}$.

Given the particular structure of a SIP constraint graph, it is possible to specialize and simplify the detection of 1-decomposition.

Property 6. Let $P = (X, \mathcal{D}, C)$ be a CSP model of a SIP instance, and let $G = (V, E^{MC} \cup E^{\neq})$ be its SIP constraint graph. Let $M = (V', E')$ be the constraint graph without assigned variables, i.e. with $V' = \{x \in X \mid |D_x| > 1\}$ and $E' = (V' \times V') \cap E^{MC}$. Then P is 1-decomposable into $P_1, \ldots, P_m$ iff M is decomposable into $M_1, \ldots, M_m$ and $D(M_i) \cap D(M_j) = \emptyset$ $(1 \leq i < j \leq m)$ with $D(M_i)$ the union of the domains of the variables associated to the nodes of M_i.

The above property states that the decomposition of M is a necessary condition. We can therefore design heuristics leading to the decomposition of M, hence in some cases in the decomposition of P.

A direct approach consists in detecting 1-decomposition at each node of the search tree. When the CSP becomes 1-decomposable in partial CSPs, those are computed separately in AND nodes. As show in the experimental section, this strategy proves to be much slower than a standard OR search tree. The reason is twofold:

1. Decomposition is tested at every node of the search tree. Starting from the root node is useless, as a lot of computation time is lost.

2. There is no guarantee that a decomposition will occur.

Based on this observation, we present a *hybrid approach* combining the best of the static and dynamic strategies.

The Hybrid Approach: As stated before, even a dedicated dynamic AND/OR search, checking for decomposition on the reduced constraint graph only, is not fast enough to compete with state-of-the-art SIP-solvers as implemented in the `vflib` library. Therefore, we suggest a hybrid approach in order to fix this. The idea is as follows:

1. calculate a static pseudo tree heuristic on the reduced constraint graph
2. apply a forward checking search following the pseudotree up to the first branching or until a fixed number of variables is assigned
3. switch the strategy to dynamic AND/OR search with full AC-propagation

This ensures, that the expensive dynamic approach is first used when a decomposition is available or at least likely after full propagation. Up to that moment, a cheap forward checking approach is used for a fast inconsistency check and a strong reduction of the reduced constraint graph.

In the following, we will give two dedicated heuristics we have applied in the preliminary forward checking procedure.

6.3.2 Heuristics

We now present two heuristics based on Property 6 aiming at reducing the number of decomposition tests, and favoring decomposition. The general idea is to first detect a subset of variables disconnecting the morphism constraint graph into disjoint components as it is a necessary condition for 1-decomposability. The search process will first distribute over these variables. The test of 1-decomposition is performed when all these variables are instantiated. It is also performed at the subsequent nodes of the search tree.

The cycle heuristic (h1)

The objective of the cycle heuristic is to find a set of nodes S in the morphism graph $CG^{MC} = (X, E^{MC})$ (see Def. 6.3.1) such that the graph without those nodes is simply connected (i.e. a tree). When the variables associated to S are assigned, any subsequent assignment will decompose the morphism graph. Finding the minimal set of nodes is known as the minimal cycle cutset problem and is a NP-Hard problem

[FL06]. We propose here a simple linear approximation that returns the nodes of the cycles of the graph. Algorithm 6 runs in $O(|V_p|)$. The effectiveness of such a procedure on different classes of problems is shown in the experimental section. One of the main advantage is its simplicity.

Algorithm 14: Selection of the body variables.

input : $G = (X, E)$ the CG^{MC}
output: The nodes of the cycles of G

```
1 All ← X
2 T ← ∅
3 while (∃ n ∈ X | Degree(n) == 1) do
4 |   T ← T ∪ {n}
5 |   remove node n from G
6 return All \ T
```

Using graph partitioning (h2)

Graph partitioning is a well-known technique that allows hard graph problems to be handled by a divide and conquer approach. In our context, it can be used to separate the morphism constraint graph into two graphs of equal size.

Definition. Given a graph $G = (V, E)$, a k-graph partitioning of G is a partition of V in k subsets, $V_1, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $\cup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized (called the *edgecut*).

Based on the edgecut of the morphism constraint graph, we can easily deduce a subset variables.

Definition. Given a 2-graph partitioning of G, a *nodecut* is a set of nodes containing one node of each edge in the cutset.

Finding a minimum edgecut is a NP-Hard problem for $k \geq 3$, but can be solved in polynomial time for $k = 2$ by matching (see [GJ90], page 209). However we use a fast local search approximation [KK98], as the exact minimum subset is not needed.

6.4 Experimental Results

Goals

The objective of our experiments is to identify the class of graphs where decomposition is effective. We compare our decomposition method on different classes of SIP with standard CSP models as well as `vflib`, the standard and reference algorithm for subgraph isomorphism [CFSV01]. We also compare our decomposition method with standard direct decomposition. The different heuristics presented in Section 3.5 are also tested.

Instances

The instances are taken from the `vflib` graph database described in [FSV01b]. There are several classes of randomly generated graph, random graphs, bounded graphs and meshes graphs. The target graphs has a size n and the relative size of the pattern is noted α. For random graphs, the target graph has a fixed number of nodes n and there is a directed arc between two nodes with a probability η. The pattern graph is also generated with the same probability η, but its number of nodes is αn. If the generated graph is not connected, further edges are added until the graph is connected. For random graphs, n takes a value in $[20, 40, 80, 100, 200, 400, 800, 1000]$, η in $[0.01, 0.05, 0.1]$, and α in $[20\%, 40\%, 60\%]$. There are thus 69 classes of randomly connected graphs. In a class of instances denoted as `si2-r001-m200`, we have $\alpha = 20\%$, $\eta = 0.01$, and $n = 200$ nodes.

Mesh-k-connected graphs are graphs where each node is connected with its k neighborhood nodes. Irregular mesh-k-connected graphs are made of a regular mesh with the addition of random edges uniformly distributed. The number of added branches is ρn. For random graphs, n can take a value in $[16, \ldots, 1096]$, k in $[2, 3, 4]$, and ρ in $[0.2, 0.4, 0.6]$. In an irregular mesh-connected class of instances denoted as `si2-m4Dr6-m625`, we have $\alpha = 20\%$, $k = 4$, $\rho = 0.6$ and $n = 625$ nodes.

One hundred graphs are generated for each class of instances. For random graphs, we also generated 100 additional instances where the target graph has 1600 nodes, for each possible value of η and α. We used the generator freely available from the graph database, following the methodology described in [FSV01b].

Models

Several models were considered for the experiments. First of all, we use the available implementation of `vflib`. Then classical CP models are used, called `CPFC` and `CPAC`. The model `CPFC` is a model where all the constraints use forward checking and the variable selection selects the first variable which is involved in the maximum number of constraints (called `maxcstr`) using minimal domain size as tiebreaker. The model `CPAC` is similar except it uses an arc consistent version of the `MC` constraint, plus n arc-consistent alldiff constraints over the neighbors of each pattern node, as proposed in [Rég95]. Similar results are observed with a single global arc-consistent alldiff, but with slightly worse performance.

The model `CP+Dec` waits for 30% of the variables to be instantiated following a variable selection policy, called `minsize`, selecting the non instantiated variable with the smallest domain. It then tests at each node of the search tree if decomposition occurs using a `maxcstr` variable selection. The model `CP+Dec+h1` uses the cycle heuristics; once the nodes belonging to the cycles of the pattern graph are instantiated using a `minsize` variable selection policy (up to 30% of the size of the pattern), decomposition is tested at each node of the search tree and follows a `maxcstr` variable selection. The model `CP+Dec+h2` uses the graph partitioning heuristics; once the variables belonging to the nodecut set are instantiated (up to 30% of the size of the pattern), decomposition is tested at each node of the search tree and follows a `maxcstr` variable selection.

Setup

All experiments were performed on a cluster of 16 machines (AMD Opteron(tm) 875 2.2Ghz with 2Gb of RAM). All runs are limited to a time bound of 10 minutes. In each experiment, we search for all solutions. Experiments searching for one solution have also been done but are not reported here for lack of space. These experiments lead to the same conclusions.

Description of the tables

Table 6.1 shows the results for random graphs and Table 6.2 for irregular mesh-connected graphs. Each line describes the execution of 100 instances from a particular class. The column N indicates the mean

number of solutions among the solved instances. The column % indicates the number of instances that were solved within the time bound of 10 minutes. The column μ indicates the mean time over the solved instances and the column σ indicates the corresponding standard deviation. The column D indicates the number of instances that used decomposition among the solved instances. The column $\#D$ indicates the mean number of decomposition that occurred over all solved instances. The column S indicates the mean size of the initial variable set computed by the heuristics `h1` or `h2`. Table 6.3 gives the mean degree and its variance for the different instances classes. For each class of instances in Tables 6.1 and 6.2, the results of the best algorithms are in bold.

Analysis

We start the analysis by looking at random graphs (see Table 6.1). We compare first the `vflib` with the CP models `CPFC` and `CPAC`. For all `si2-*` and `si6-*` instances, the `CPAC` model is the best in mean time and % of the solved instances except for `si2-r001-m200`, where `CPFC` is the best.

We now look at the comparison of decomposition methods for random graphs (second table in Table 6.1).

First, we will focus on the `si2-r001-*` classes. The models `CP+Dec+h1` and `CP+Dec+h2` achieve better decompositions than the `CP+Dec` model. Even though `CP+Dec` tends to induce more decompositions, the number of instances using decomposition (see column D) is higher for `CP+Dec+h1` and `CP+Dec+h2` than for `CP+Dec`. This visualizes the computational overhead of a pure dynamic decomposition approach. However, the number of instances using decomposition tends to be zero for `m1600` instances. This is due to the fact that the graphs have higher degrees as their size increases (see Table 6.3). This can be observed by looking at the column S: the size of the initial subset of variable to instantiate becomes closer to 100% as size increases. Indeed, our decomposition method is beaten by the `CPAC` model for `si2-r001-m800` and `m1600`.

We now focus on the `si6-r01-*` classes. As stressed earlier, those instances have denser graphs. The initial set of variables to instantiate is the whole set of pattern nodes for `CP+Dec+h1` and `CP+Dec+h2`. No decomposition occurs. Why then `CP+Dec+h*` models outperform all other methods in those classes? Because in the class `si6-r01-*`, the

Table 6.1: Randomly connected graphs, searching for all solutions.

Bench		vflib			CPAC			CPFC		
	N	%	μ	σ	%	μ	σ	%	μ	σ
si2-r001-m200	61E+6	72	74	115	83	56	109	85	41	76
si2-r001-m400	17E+8	2	248	118	10	106	156	7	288	177
si2-r001-m800	28E+7	0	-	-	**14**	**166**	**133**	1	153	-
si2-r001-m1600	2500	16	203	202	**81**	**224**	**93**	0	-	-
si6-r01-m200	1	100	2	3	100	9	11	100	12	17
si6-r01-m400	1	66	99	133	89	156	116	50	190	137
si6-r01-m800	1	7	235	153	0	-	-	5	389	125
si6-r01-m1600	1	0	-	-	0	-	-	39	499	51

Bench		CP+Dec					CP+Dec+h1						CP+Dec+h2					
	N	%	μ	σ	D	#D	%	μ	σ	D	#D	S	%	μ	σ	D	#D	S
si2-r001-m200	61E+6	94	49	100	91	9244	**98**	**6**	**40**	98	1834	0.2	87	23	48	71	909	0.2
si2-r001-m400	17E+8	15	160	177	15	35655	**75**	**68**	**125**	75	2268	0.4	29	212	218	22	196	0.3
si2-r001-m800	28E+7	0	-	-	0	12	4	227	254	4	21	0.6	12	256	239	8	0	0.6
si2-r001-m1600	2500	0	-	-	0	0	7	165	199	1	0	0.8	0	-	-	0	0	0.9
si6-r01-m200	1	94	148	153	0	0	**100**	**0**	**0**	0	0	1	100	0	0	0	0	1
si6-r01-m400	1	2	179	220	0	0	**100**	**2**	**1**	0	0	1	100	4	6	0	0	1
si6-r01-m800	1	0	-	-	0	0	**100**	**46**	**35**	0	0	1	100	46	39	0	0	1
si6-r01-m1600	1	0	-	-	0	0	**74**	**479**	**71**	0	0	1	54	435	79	0	0	1

Table 6.2: Irregular meshes, searching for all solutions.

Bench		vflib			CPAC			CPFC		
	N	%	μ	σ	%	μ	σ	%	μ	σ
si2-m4Dr6-m625	88E+5	89	23	50	94	21	38	95	6	27
si2-m4Dr6-m1296	17E+7	16	135	137	33	178	123	38	107	154
si6-m4Dr6-m625	3.31	100	7	43	100	29	4	100	9	4
si6-m4Dr6-m1296	10.38	**100**	**13**	**55**	100	233	30	100	113	65

Bench		CP+Dec					CP+Dec+h1						CP+Dec+h2					
	N	%	μ	σ	D	#D	%	μ	σ	D	#D	S	%	μ	σ	D	#D	S
si2-m4Dr6-m625	88E+5	35	223	151	35	0.7	**100**	**6**	**22**	96	5.4	0.5	94	6	21	88	5.5	0.3
si2-m4Dr6-m1296	17E+7	3	120	36	3	0.1	**63**	**67**	**109**	63	4	0.5	49	163	170	49	3.9	0.5
si6-m4Dr6-m625	3.3	8	105	32	0	0	**100**	**7**	**3**	6	0.1	0.8	100	22	26	6	0.1	0.7
si6-m4Dr6-m1296	10.3	0	-	-	0	0	100	65	20	41	0.6	0.7	77	223	161	29	0.4	0.7

Table 6.3: Mean degree for the tested graph set.

Bench	degree	
	μ	σ
si2-r001-m200	2.30	0.14
si2-r001-m400	2.89	0.14
si2-r001-m800	3.99	0.18
si2-r001-m1600	6.80	0.19
si6-r01-m200	3.29	0.14
si6-r01-m400	5.27	0.16
si6-r01-m800	9.76	0.15
si6-r01-m1600	19.20	0.17
si2-m4Dr6-m625	3.51	0.26
si2-m4Dr6-m1296	3.53	0.20
si6-m4Dr6-m625	5.12	0.16
si6-m4Dr6-m1296	5.19	0.14

`CP+Dec+h1` approach reduces to a hybrid level of consistency between `CPFC` and `CPAC` with a minsize variable selection policy during the forward checking phase.

For random graphs, the decomposition method with heuristics is especially useful for sparse graphs with many solutions, while an hybrid model beginning with forward checking and switching to arc consistency at some point seems the best choice for denser graphs and there are few solutions. The `vflib` is clearly outperformed on all these classes of instances. Experiments on the other classes of random graphs, not reported here for lack of space, confirmed this analysis.

We now analyze irregular mesh-connected graphs. We observe in Table 6.3 that the mean degree of the `si2-m4Dr6-*` classes is higher than for the `si6-m4Dr6-*` classes. We first compare the `vflib` and CP models without decomposition. For sparser `si2-m4Dr6-*` classes, `CPFC` is the best method, while for denser `si6-m4Dr6-*` classes, `vflib` is the best. We have no particular explanation for this behavior and this is an open question. Regarding decomposition methods, the same remarks than for random graphs apply. The `CP+Dec` model tends to produce less decomposition than the `CP+Dec+h*` models. Moreover, `CP+Dec+h*` models are the best models for sparser instances with many solutions. As the mean degree of the instances increases (see Table 6.3) and the

number of solutions decreases, the decomposition methods become less efficient. Indeed, for `si6-m4Dr6-m1296`, the best method is `vflib`, but our decomposition approach also solves all the instances and helps CP at diminishing the mean time.

Summary

The application of standard direct decomposition methods `CP+Dec` leads to performances worse than the direct application of standard CP models (`CPFC`, `CPAC`) and `vflib`. On most classes, the cycle heuristic (`h1`) is better than the graph partitioning heuristic (`h2`). On sparse randomly connected graphs with many solutions, and on sparse irregular meshes, our decomposition method outperforms standard CP approaches as well as `vflib`. For denser connected graphs, hybrid CP models between `CPAC` and `CPFC` with a `minsize` policy) is the best choice. For denser irregular meshes, `vflib`, the standard CP models and our decomposition method solve all the instances, but `vflib` is more efficient.

6.5 Conclusion

Our initial question was to investigate the application of decomposition techniques as AND/OR search for problems with global constraints, in particular for the SIP. We showed that it is indeed possible using a hybrid approach of static and dynamic techniques and a dedicated problem structure analysis. For the SIP, one can derive a decomposition enforcing static heuristic that is used by a cheap forward checking approach. As soon as the problem gets (likely) decomposable, the search process is switched to a fully propagated, dynamically decomposed search. This exploits the non-predictable reduction of the constraint graph structure via constraint propagation and entailment but reduces the huge computational effort of a completely propagated search. We showed that our hybrid decomposition approach is able to beat the state-of-the-art VF-algorithm for sparse graphs with high solution numbers. As future work, we would like to investigate more heuristics for SIP as it influences the quality of decomposition. Moreover, we intend to investigate the use of our decomposition method for motif discovery where solving SIP is used as an enumeration tool [GK07].

7

Conclusion

The goal of this thesis is to propose an efficient and declarative framework for graph matching. We propose such a framework in Chapter 3. The idea it to use graph and map variables, instead of ground graph and function objects. Map variables where the domain and the codomain are not ground were introduced. A specific map constraint that prunes the array by taking into account the available information from the domain and the codomain is developed. Together with graph variables, this allows to express graph matching problems, using optional nodes and optional arcs. Morphism constraints can be extended to this framework. The experiments have shown that such a framework is efficient for subgraph isomorphism compared to dedicated state-of-the-art algorithms and compared to a direct CP model. Hence embedding graph matching problems in our framework is efficient, but we also inherit from the expressiveness of the various constraints defined over the graph computation domain.

Chapter 4 turned to the problem of designing stronger propagators for the subgraph isomorphism problem. We show how to extend the idea of iterative labelling in the context of subgraph isomorphism, instead of the graph isomorphism [SS08]. Such an extension implies the computation of a partial order on the labelings. The idea was proved to be well-founded and efficient. Indeed, benchmarks on difficult problems - that is instances leading to a large search space with few or no solutions - show great benefits in using the proposed propagator, outperforming all previous CP models. Our framework is thus able to deal with difficult problems. It remains to show that common constraint programming

techniques can be incorporated in our framework and that our framework can beat dedicated state-of-the-art algorithm vflib for problems with many solutions.

Chapter 5 studies the use of symmetries in the context of subgraph isomorphism. The symmetries of the subgraph isomorphism depend on the instance, and we show how it is indeed possible to detect automatically all global variable and value symmetries, and how to detect local variable and value symmetries. Moreover, experiments show the benefits of global and local symmetries.

Chapter 6 studies the limits of the direct application of decomposition to the subgraph isomorphism problem. Even though the initial constraint graph is complete, decomposition can be achieved by computing a static heuristic over the pattern graph and by using forward checking in a first stage of the search. Afterwards full arc-consistent propagation is triggered and decomposition is checked. Such an approach has been proved to be effective on random sparse graphs containing a lot of solutions. Hence, not only our framework is able to deal with difficult problems, it is also able to deal with problems with a high density of solutions.

The significance of this thesis lies in the fact that, even though our framework is expressive, CP can be considered as the state-of-the-art for subgraph isomorphism, outperforming the dedicated state-of-the-art vflib algorithm for difficult instances - where CP is known to be effective - but also on the vflib random graph benchmarks, especially on sparse instances. Together with decomposition, it offers moreover the automatic detection of symmetries, which have been shown to be important in motif discovery [GK07].

Memory consumption is however a clear advantage for vflib. As shown in this thesis, the CP approach consumes more memory than the vflib algorithm. As stressed in the conclusion of the chapter 2, this is a consequence of the CP approach that considers the way to reduce the domain of computation, whereas the vflib algorithm uses a constructive approach. For this reason, we think a challenge is to overcome the memory gap between vflib and the CP approach.

In addition to this open issue, we would like to stress some interesting future works.

Regarding the matching framework, it would be interesting to integrate the techniques proposed in [SS08] and [Rég03], for graph isomorphism and maximum common subgraph respectively. Path constraints

can also be integrated to the framework, enabling a feature available in some matching algorithms [FGP+07].

Regarding the subgraph isomorphism constraint, there remains two open questions: the first one concerns theoretical study on the level of consistency achieved when the fixpoint is reached, even though reaching the fixpoint has been showed to be ineffective in practice. Moreover, a criteria should also be designed to stop the iterations of the labelling process.

Can symmetry techniques be successfully applied in a context where computational difficulty is proportional to the number of solutions ? [GK07] showed that global variable symmetries are essential to solve the motif discovery problem using subgraph isomorphism, where subgraphs have to be enumerated and enumeration is important. This is mainly because there are a lot of solutions, and finding the unique solutions regarding symmetries is very efficient. It would be interesting to reproduce the experiments conducted in [GK07], to see if global variable symmetries are more efficient. Moreover, global value symmetries as well as local symmetries could be used. Decomposition is useful for sparse graphs with a lot of solutions, and hence could also be applied and evaluated in this context, which leads to the question of knowing how to combine decomposition and symmetry breaking.

Bibliography

[Aze07] Francisco Azevedo. Cardinal: A finite sets constraint solver. *Constraints*, 12(1):93–129, 2007.

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

[BC94] N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.

[BE05] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer, 2005.

[Bel00] Nicolas Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report T2000/01, SICS, 2000.

[Ben94] Belaid Benhamou. Study of symmetry in constraint satisfaction problems. In Alan Borning, editor, *Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Proceedings*, volume 874 of *Lecture Notes in Computer Science*, pages 246–254, Seattle, USA, May 1994. Springer.

[Bes06] Christian Bessière. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 3, pages 29–70. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[BFL05] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In Barták and Milano [BM05], pages 64–78.

[BHH+05a] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the nvalue constraint. In Barták and Milano [BM05], pages 79–93.

[BHH+05b] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The range and roots constraints: Specifying counting and occurrence problems. In Kaelbling and Saffiotti [KS05], pages 60–65.

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[BM05] Roman Barták and Michela Milano, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*, volume 3524 of *Lecture Notes in Computer Science*. Springer, 2005.

[BS06] Belaïd Benhamou and Mohamed Réda Saïdi. Reasoning by dominance in not-equals binary constraint networks. In Benhamou Frédéric, editor, *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming, CP2006*, Lecture Notes in Computer Science, pages 670–674, Cité des Congrès - Nantes, France, September 2006. Springer.

[CB04] Hadrien Cambazard and Eric Bourreau. Conception d'une contrainte globale de chemin. In *10e Journées nationales sur la résolution pratique de problèmes NP-complets*, pages 107–121, 2004.

[CDPP04] A. Chabrier, E. Danna, C. Le Pape, and L. Perron. Solving a network design problem. *Annals of Operations Research*, 130:217–239, 2004.

[CFS+98] Luigi Cordella, Pasquale Foggia, Carlo Sansone, F.Tortella, and Mario Vento. Graph matching: A fast algorithm and

its evaluation. In *ICPR '98: Proceedings of the 14th International Conference on Pattern Recognition-Volume 2*, page 1582, Washington, DC, USA, 1998. IEEE Computer Society.

[CFSV99] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*, page 1172, Washington, DC, USA, 1999. IEEE Computer Society.

[CFSV01] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159. Cuen, 2001.

[CGLR96] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[CH07] Irène Charon and Olivier Hudry. A survey on the linear ordering problem for weighted or unweighted tournaments. *4OR*, 5(1):5–60, 2007.

[CJJ+05] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E.Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. In van Beek [vB05], pages 17–31.

[CKC82] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et sciences informatiques*, 2(4), 1982.

[CPSV99] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *LNCS*, 1551:16–30, 1999.

[CS03a] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In Kevin D. Ashley and Derek G. Bridge, editors, *5th Int. Conf. On Case-Based Reasoning (ICCBR 2003)*, LNAI, pages 80–95. Springer, June 2003.

[CS03b] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In Kevin D. Ashley and Derek G. Bridge, editors, *ICCBR*, volume 2689 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2003.

[DBDK04] Peter J. Dickinson, Horst Bunke, Arek Dadej, and Miro Kraetzl. Matching graphs with unique node labels. *Pattern Anal. Appl.*, 7(3):243–254, 2004.

[DBL06] *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 2006.

[DC01] Daniel Diaz and Philippe Codognet. Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.

[DCV07] Pasquale Foggia Donatello Conte and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[DDD05] Grégoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *Principles and Pratice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225, 2005.

[DDZD05] Yves Deville, Grégoire Dooms, Stéphane Zampelli, and Pierre Dupont. Cp(graph+map) for approximate graph matching. *1st International Workshop on Constraint Programming Beyond Finite Integer Domains, CP2005*, pages 33–48, 2005.

[Dec03] Rina Dechter. *Constraint Processing (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, May 2003.

[DK03] Fred DePiero and David Krout. An algorithm using length-r paths to approximate subgraph isomorphism. *Pattern Recogn. Lett.*, 24(1-3):33–46, 2003.

[DK06] G. Dooms and I. Katriel. The minimum spanning tree constraint. In *Proc. of the 12th International Conference on Principles and Practice of Constraint Programming*, pages 152–166, 2006.

[DK07] G. Dooms and I. Katriel. The "not-too-heavy" spanning tree constraint. In *Proceedings of CPAIOR 2007*, 2007.

[DM04] Rina Dechter and Robert Mateescu. The impact of AND/OR search spaces on constraint satisfaction and counting. In *Proc. of the CP'2004*, 2004.

[DM07] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[DPBT99] Paul J. Durand, Rohit Pasari, Johnnie W. Baker, and Chun-Che Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 1999.

[DvH87] Mehmet Dincbas and Pascal van Hentenryck. Extended unification algorithms for the integration of functional programming into logic programming. *J. Log. Program.*, 4(3):199–227, 1987.

[DZD08] Yves Deville, Stéphane Zampelli, and Grégoire Dooms. Combining two structured domains for modeling various graph matching problems. In F. Fages, F. Rossi, and S. Soliman, editors, *Recent Advances in Constraint Programming*. Springer-Verlag, 2008.

[FGP+07] A. Ferro, R. Giugno, G. Pigola, A. Pulvirenti, D. Skripin, G. D. Bader, and D. Shasha. Netmatch. *Bioinformatics*, 23(7):910–912, 2007.

[FHK01] Pierre Flener, Brahim Hnich, and Zeynep Kiziltan. Compiling high-level type constructors in constraint programming. In *PADL '01: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, pages 229–244, London, UK, 2001. Springer-Verlag.

[FJHM05] Alan M. Frisch, Chris Jefferson, Bernadette Marinez Hernandez, and Ian Miguel. The rules of constraint modelling. In *Proceedings of IJCAI 2005*, 2005.

[FL06] François Fages and Akash Lal. A constraint programming approach to cutset problems. *Comput. Oper. Res.*, 33(10):2852–2865, 2006.

[FSV01a] P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations*, 2001.

[FSV01b] P. Foggia, Carlo Sansone, and Mario Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarcking. *CoRR*, cs.PL/0105015, 2001.

[Ger93] C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *Third Workshop on Constraint Logic Programming (WCLP'93)*, Marseille, 1993.

[Ger97] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[Ger06] Carmen Gervet. Constraints over structured domains. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 17, pages 605–638. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[GH06] Carmen Gervet and Pascal Van Hentenryck. Length-lex ordering for set csps. In *AAAI* [DBL06].

[GHK03] I.P. Gent, W. Harvey, and T. Kelsey. Generic sbdd using computational group theory. In *Proceedings of CP'03*, pages 333–346, 2003.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GK07] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In Terence P. Speed and Haiyan Huang, editors, *RECOMB*, volume 4453 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[GKL+05] Ian .P. Gent, Tom Kelsey, Steve A. Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In van Beek [vB05], pages 256–270.

[GS01] I.P. Gent and B.M. Smith. Symmetry breaking during search in constraint programming. In *Proceedings of CP'01*, pages 599–603, 2001.

[GS02] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR (2)*, pages 112–115, 2002.

[Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, Cambridge, MA, USA, 1989.

[Hen02] Pascal Van Hentenryck. Constraint and integer programming in opl. *INFORMS J. on Computing*, 14(4):345–372, 2002.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[HN02] Mohammad Taghi Hajiaghayi and Naomi Nishimura. Subgraph isomorphism, log-bounded fragmentation and graphs of (locally) bounded treewidth. In *MFCS '02: Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, pages 305–318, London, UK, 2002. Springer-Verlag.

[Hni03] B. Hnich. *Function variables for Constraint Programming*. PhD thesis, Uppsala University, Department of Information Science, 2003.

[HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184, New York, NY, USA, 1974. ACM.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[Knu93] Donald E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM, New York, NY, USA, 1993.

[KS05] Leslie Pack Kaelbling and Alessandro Saffiotti, editors. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*. Professional Book Center, 2005.

[Lau78] J. L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–128, 1978.

[Luk80] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *FOCS*, pages 42–49. IEEE, 1980.

[LV02] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, 2002.

[LvB04] Wei Li and Peter van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *Proc. of the 16th IEEE International Conference on Tools with Artificial Intelligence*, 2004.

[Mat07] R. Mateescu. *AND/OR Search Spaces for Graphical Models*. PhD thesis, 2007.

[MB98] Bruno T. Messmer and Horst Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(5):493–504, 1998.

[McG79] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3):229–250, 1979.

[McG82] James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software - Practice and Experience*, 12(1):23–34, 1982.

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

[MD05] Robert Mateescu and Rina Dechter. AND/OR cutset conditioning. In *Proc. of the IJCAI'2005*, 2005.

[MT01] P. Meseguer and C. Torras. Exploiting symmetries within the constraint satisfaction search. *Artificial intelligence*, 129(1-2):133–163, 2001.

[PGPR98] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[PTDM04] K. A. Sakallah P. T. Darga, M. H. Liffiton and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proc. Design Automation Conference (DAC)*, pages 530–534. IEEE/ACM, June 2004.

[Pug92] J.-F Puget. Pecos a high level constraint programming language. In *Proceedings of Spicis 92*, 1992.

[Pug94] Jean-Francois Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.

[Pug03] Jean-François Puget. Using constraint programming to compute symmetries. *Thirdth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03)*, 2003.

[Pug04] Jean-François Puget. Breaking symmetries in all different problems. *Fourth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'04)*, 2004.

[Pug05a] Jean-François Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[Pug05b] Jean-Francois Puget. Breaking symmetries in all different problems. In Kaelbling and Saffiotti [KS05], pages 272–277.

[Pug05c] Jean-Francois Puget. Elimination des symétries dans les problèmes injectifs. In *Proceedings des Journées Francophones de la Programmation par Contraintes*, 2005.

[Pug05d] Jean-François Puget. Automatic detection of variable and value symmetries. In van Beek [vB05], pages 477–489.

[RDGKL04] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 211–215. IOS Press, 2004.

[Reg94] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[Rég03] Jean-Charles Régin. Using constraint programming to solve the maximum clique problem. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 634–648. Springer, 2003.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[RKH05] A. Robles-Kelly and E.R. Hancock. Graph edit distance from spectral seriation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27-3:365–378, 2005.

[Rud98a] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Theory and Application of Graph Transformations*, number 1764 in Lecture Notes in Computer Science, pages 238–252. Springer, 1998.

[Rud98b] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Theory and Application of Graph Transformations*, number 1764 in Lecture Notes in Computer Science, pages 238–252. Springer, 1998.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[Rég95] Jean-Charles Régin. Developpement d'outils algorithmiques pour l'intelligence artificielle : application a la chimie organique. *PhD Thesis*, 1995.

[Rég03] Jean-Charles Régin. Using constraint programming to solve the maximum clique problem. In *Principles and Practice of Constraint Programming, 9th International Conference, CP 2003*, volume 2833 of *LNCS*, pages 634–648. Springer, 2003.

[SC06] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[Sch02] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[Sel03] Meinolf Sellmann. Cost-based filtering for shorter path constraints. In *Proceedings of the 9th International Conference on Principles and Practise of Constraint Programming (CP)*, volume LNCS 2833, pages 694–708. Springer-Verlag, 2003.

[Smi87] D.R. Smith. Structure and design of global search algorithms. Technical Report Tech. Report KES.U.87.12, Kestrel Institute, Palo Alto, Calif., 1987.

[Smi01] B. Smith. Reducing symmetry in a combinatorial design problem. *Proc. CP-AI-OR'01, 3rd Int. Workshop on Integration of AI and OR Techniques in CP*, 2001.

[Smo95] Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[SS04a] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.

[SS04b] Sébastien Sorlin and Christine Solnon. A global constraint for graph isomorphism problems. In *6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004)*, volume 3011 of *LNCS*, pages 287–301. Springer-Verlag, 2004.

[SS05] Sébastien Sorlin and Christine Solnon. Reactive tabu search for measuring graph similarity. In Mario Vento Luc Brun, editor, *5th IAPR-TC-15 workshop on Graph-based Representations in Pattern Recognition*, pages 172–182. Springer-Verlag, April 2005.

[SS06] Sébastien Sorlin and Christine Solnon. A new filtering algorithm for the graph isomorphism problem. *3rd International Workshop on Constraint Propagation and Implementation, CP2006*, 2006.

[SS08] Sébastien Sorlin and Christine Solnon. A parametric filtering algorithm for the graph isomorphism problem. *Constraints*, 13(4), December 2008.

[SSJ07] Sébastien Sorlin, Christine Solnon, and Jean-Michel Jolion. A Generic Graph Distance Measure Based on Multivalent Matchings, April 2007. Chapter of the book "Applied

Graph Theory in Computer Vision and Pattern Recognition", Vol. 52, Series "Studies in Computational Intelligence", Springer, pages 151-182.

[SSSG06] Olfa Sammoud, Sébastien Sorlin, Christine Solnon, and Khaled Ghedira. A Comparative Study of Ant Colony Optimization and Reactive Search for Graph Matching Problems. In Günther Raidl and Jens Gottlieb, editors, *6th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2006)*, LNCS, pages 287–301. Springer, April 2006.

[ST06] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.

[Thi04] Sven Thiel. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. PhD thesis, University of Saarbrucken, 2004.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[Val02] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.

[vB05] Peter van Beek, editor. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, Augustus 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*. Springer, 2005.

[VM97] Gabriel Valiente and Conrado Martínez. An algorithm for graph pattern-matching. In *Proc. 4th South American Workshop on String Processing*, volume 8 of *International Informatics Series*, pages 180–197. Carleton University Press, 1997.

[Wer06] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):347–359, 2006.

[WNS97] M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, Imperial College, 1997.

[WZC95] Jason T. L. Wang, Kaizhong Zhang, and Gung-Wei Chirn. Algorithms for approximate graph matching. *Inf. Sci. Inf. Comput. Sci.*, 82(1-2):45–74, 1995.

[ZDD05] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate constrained subgraph matching. In *Principles and Pratice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 832–836, 2005.

[ZDD06] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Symmetry breaking in subgraph pattern matching. *Sixth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'06)*, 2006.

[ZDD07] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Symmetry breaking in subgraph pattern matching. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, pages 203–218. ISTE Hermes, 2007.

[ZDS+07] Stéphane Zampelli, Yves Deville, Christine Solnon, Sébastien Sorlin, and Pierre Dupont. Filtering for subgraph isomorphism. In *Proc. 13th Conf. of Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 728–742. Springer, 2007.

[Öst02] Patric R. J. Östergaard. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, 2002.