Université catholique de Louvain
Faculté des Sciences Appliquées

The CP(Graph) Computation Domain in Constraint Programming

Grégoire Dooms

Dissertation présentée en vue de l'obtention du titre de Docteur en Sciences Appliquées.

Composition du jury:

Pr. Y. Deville, UCL-INGI (Co-Promoteur),
Pr. P. Dupont, UCL-INGI (Co-Promoteur),
Pr. P. Van Roy, UCL-INGI (Examinateur),
Pr. L. Wolsey, UCL-CORE (Examinateur),
Pr. C. Schulte, KTH-IMIT, Sweden (Examinateur),
Pr. P. Prosser, University of Glasgow, U.K. (Examinateur),
Pr. A. Van Lamsweerde, UCL-INGI (Président)

Louvain-la-Neuve
Année académique 2005-2006
Département d'Ingénierie Informatique

Acknowledgments

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

I would like to thank Yves Deville and Pierre Dupont, my advisers, for the support, numerous advices, ideas and insights about my research and this thesis. These years at UCL have not just been very interesting on a scientific basis but also on a personal basis.

I would like to thank all my colleagues at the Département d'Ingénierie Informatique and in the Faculty of Applied Sciences of UCLouvain for the nice atmosphere and the interesting discussions we had. Thank you to all the administrative and technical staff of the department. Thank you to all the people with which I spent noons eating or hanging out at the cafeteria.

I would like to thank Yves Deville, Stéphane Zampelli, Raphaël Collet, Luis Quesada and Peter Van Roy for helping me in my study of constraint programming.

I would like to thank Stéphane Zampelli and Raphaël Collet for the support and advices they provided me with when I struggled with the Mozart and Gecode systems. We also had pretty good times discussing bugs, fighting language wars and discussing syntax down at the cafeteria.

I would like to thank Jérome Callut, Pierre Dupont, and Sébastien Vast for the enlightening discussions we had about the random walk approaches to biochemical network analysis. I would also like to thank Philippe Delsarte for his help and interesting discussions about discrete mathematics and teaching thereof.

I would like to thank all my colleagues of the third floor of the Reaumur building: Christophe Damas, Renaud De Landtsheer, David Janssens, Bernard Lambeau of the KAOS group, and Jérome Callut, Jean-Noël Monette, Pierre Schaus, Sébastien Vast and Stéphane Zampelli of the AI and Constraints group. A special atmosphere mixing fun and work and especially fun about work helped me during the good and the bad days. Working late has been pretty fun a few times.

I would like to thank Christian Schulte, Guido Tack, Michael Lagerkvist of the Gecode team and the rest of the staff at IMIT, KTH for their great hosting and advices during the preparation of the release of the CP(Graph) prototype.

I would like to thank Irit Katriel for interesting discussions about graph algorithmics, constraint programming, politics, sports and slavery and for her support during the preparation of this thesis.

I would like to thank Christian Schulte, Patrick Prosser, Peter Van Roy, and Lawrence Wolsey for the numerous comments and suggestions they gave me about a draft of this thesis.

I would like to thank Marc Lobelle, Yves Deville and the Walloon Region for financial support during these years at UCLouvain. This work has been done while I was funded by the WIST project BioMaze. I would also like to thank Yves Deville for giving me the opportunity of teaching Python at Technofutur ASBL. I also thank him for the opportunity to lecture this course at the IFI in Hanoi and Alain Pirotte for the good time we had there.

I would like to thank all my friends for the support and happiness they feed me with and in particular Stéphane Bataille, Thomas Goor, David Janssens, Antony Lesuisse, Olivier Martin, Benjamin Michiels, Julien Thewys and Patrick Watrin for the fun time we had days and nights in Louvain-la-Neuve. Thank you also to all those which although not cited will recognize themselves.

I would finally like to thank my parents, my family and Julie Cailleau. Thank you Julie for your support, intelligence, courage, empathy and love without which I would never have accomplished this work.

Contents

List of Figures

Chapter 1

Introduction

Constraint Programming (CP) is the science and technology for dealing with constraints. It is at the crossing of Artificial Intelligence (AI), Operations Research (OR), Algorithmics and programming languages. One of the main field in CP is that of solving constraint satisfaction problems (CSP) by combining constraint propagation with search. A constraint satisfaction problem is specified by a set of variables each with a set of possible values called their domain and a set of constraints to be satisfied. An assignment of a value from its domain to each variable is a solution to the CSP if it satisfies all constraints.

To find exact solutions to CSPs, the search space is explored by interleaving a domain filtering step with a choice step. The role of the filtering phase also called constraint propagation is to reduce the search space by identifying values which belong to no solution and can be safely removed from a variable's domain. The choice step splits the problem into two or more easier subproblems. These steps are interleaved until the subproblems are easy to solve. The shape of the search tree and the way it is explored is under the control of the user which can choose which type of choices are used and in which order the search tree is explored.

Higher Level Programming in CP Over the years, some works have been dedicated towards providing higher level constructs to model and solve constraint satisfaction problems within CP: global constraints, constraint combinators, modeling languages and global variables.

Global constraints are constraint aggregating a common pattern of constraints and using a dedicated filtering algorithm for this conjunction of simpler constraints. They ease the modeling process by providing a high-level construct which can be applied to a wide range of problems. They also speed-up the resolution by providing an efficient filtering algorithm which outperforms the uninformed combination of the filtering algorithms of the basic constraints.

Constraint combinators were presented in [84] and allowed the user to declaratively specify higher level constraints by combining basic constraints with disjunction, implication and cardinality operators.

Modeling languages such as OPL [120] allow to express CSPs and search strategies using high-level algebraic and set notations which are then translated to a lower level constraint language and engine using integer variables and constraints.

Finally, higher-level variable types such as sets [55] allow to model the problem with higher level variables and constraints which are directly handled by the solver without being translated to integers. This allows the solver to better exploit the structure inherent to the variable type.

Graphs in Constraint Programming Graphs are used in several areas of Constraint Programming. First, binary CSPs – CSPs whose constraints involve two variables – have been represented as graphs called constraint networks and these graphs have been used to study the properties of these CSPs. These networks have also been used to describe global constraints as networks of simple constraints in the global constraint catalog [3]. A generic filtering algorithm for global constraints based on this graph representation has also been introduced [5].

Graph problems have been modeled using integer CSPs. Several different integer models have been used to represent graphs and several global constraints have been designed to model graph properties.

Finally, global constraints such as the global cardinality constraint (GCC) or the all different constraint (alldiff which holds if its arguments are pairwise distinct) use graphs in their filtering algorithm.

These subjects and their close relation with graph intervals are presented in more depth in section 3.6.

Contributions

- We introduce graph variables and basic constraints that open a high-level declarative framework for expressing and solving graph based CSPs.

- We show that this framework integrates with the former models and provides an high level interface to apply graph constraints on different underlying graph models.

- We show that applications for graph intervals extend beyond graph variables in CP. In particular, we show that classical filtering algorithms for global constraints such as GCC or alldiff can be interpreted as filtering algorithms for simple graph properties.

- The set of basic graph constraints is enriched with a set of global graph constraints which model classical graph properties and relations such as subgraph, connected or spanning tree. Filtering algorithms for these constraints are designed, presented and implemented. We analyze, integrate, extend, and implement the simple path [37, 21] and shorter path [110, 109, 49] constraints.

- Two novel graph constraints for weighted spanning tree problems are introduced[1]. Their filtering algorithms are based on state of the art graph algorithmic components. They provide two compact modeling tools which generalize the problems of minimum spanning tree verification, minimum spanning tree sensitivity, robust spanning tree design and inverse parameter optimization.

- We design, implement and evaluate a practical framework based on the Gecode library. This proof of concept provides a new variable type, reusable models for dealing with graphs and constraints over these models and variable type. This software has been released as an open-source contribution to Gecode.

- This implementation of CP(Graph) is applied to a constrained path finding problem arising in Bioinformatics. Several experiments are presented and discussed to evaluate this implementation of CP(Graph) on top of Gecode and further developments are proposed.

Outline The next chapter is dedicated to background on graph theory and constraint programming. We describe and formalize important aspects of constraint programming such as the execution model which is assumed

[1] This work is joint work with Irit Katriel. It has been conducted over nearly a year and is the result of several iterative refinements of the algorithms, proofs and presentation. My personal contribution resides in the original idea for several of the algorithms, lemmas and proofs; particularly for the MST constraint. We both contributed significantly to every parts of this work.

in the rest of the thesis. We also define some notions of graph theory which are used in the rest of this thesis.

Chapter 3 describes the bases of the CP(Graph) computation domain. The lattice structure of sets of graphs is described and used to define graph intervals. We then present a small set of kernel constraints which connect graphs with other computation domains of CP. This set of constraints was designed as a compact set of constraints to be introduced along with graph variables in order to allow the modeling of graph-based problems using graph variables in CP. We illustrate this by using these constraints to express higher-level graph constraints and classical graph problems. We also show that some constraints cannot be decomposed into basic constraints in a compact way.

The second part of this thesis concerns global graph constraints. In Chapter 4, we describe a set of graph constraints for enforcing several graph properties in a graph CSP. We present and prove consistency levels and filtering algorithms for these constraints.

In Chapter 5, we present two novel global graph constraints for weighted spanning trees: the minimum spanning tree constraint and the weight bounded spanning tree constraint. Filtering algorithms are designed and their level of consistency is proved.

In Chapter 6, we describe an implementation of the CP(Graph) computation domain as an extension to the Gecode library. We describe a basic API for graph domains and show how graph models based on sets or integers can be integrated in this model. We describe a graph variable type implementation and features to help develop graph constraints and graph CSPs. We stress the integration of the CP(Graph) computation domain with other domains.

In Chapter 7, we first present an application of CP(Graph) to constrained path finding in biochemical networks. Another application to graph matching by Deville et al. [30] is described. It consists in the use of a single graph monomorphism constraint and graph variables to model problems ranging from graph monomorphism to approximate subgraph isomorphism. We then evaluate the Gecode implementation of the CP(Graph) computation domain. Some experiments are presented to compare the efficiency of the different graph models for different classes of problems in Gecode. Two other experiments on the connected constraint and the Knight's tour highlight the benefits of using global graph constraints. Another experiment highlights some shortcomings of the chosen execution model when dealing with many basic constraints over a graph variable instead of higher level constraints such as those presented in Chapters 4 and 5. Optimizations and an adaptation of the execution model are sketched in this chapter too.

We finally conclude this thesis and discuss future works.

Chapter 2

Background

Before the presentation of the CP(Graph) computation domain in chapter 3, we go through a few preliminary notions about constraint programming and graph theory. The constraint programming section covers basic notions but also the execution model, constraint services and notation which are used in the rest of this thesis. The graph theory part defines notions which are used to express graph based CSPs or design filtering algorithms for graph constraints.

2.1 Constraint Programming

2.1.1 Constraint Satisfaction Problems

Let $\mathcal{X} = \{X_1, ..., X_k\}$ be a set of variables. Let $\mathcal{U}$ be the universal set of all possible values. Each variable $X \in \mathcal{X}$ has a set of possible values $D(X) \subseteq \mathcal{U}$ which can be assigned to the variable and which is called the *domain* of the variable. In the rest of this section, we consider that the set $\mathcal{X}$ of variables is enough for all CSPs.

An *assignment* s is the instantiation of each variable $X \in \mathcal{X}$ to a single value $s(X)$: $D(X) = \{s(X)\}$. In this case, the variables are said to be *assigned* or *fixed*. On the other hand, when the domain of a variable contains more than one value, the variable is said to be *unassigned* or *non-fixed*.

As the elements of $2^{\mathcal{U}}$ – each possible set of values of $\mathcal{U}$ – might require large amounts of memory to represent, it can be more convenient to only allow some specific sets of values (which are more compact to represent). This was formalized as *approximate domains* in [11].

A *domain approximation* A is a subset of $2^{\mathcal{U}}$ which is closed under intersection and contains at least these elements: the empty domain $\emptyset$, a complete domain $V \subseteq \mathcal{U}$ and singletons $\{v\}$ for each value v in V (these mandatory elements constitute an unit approximate domain in [11]).

Using this notion of domain approximation, we define a *store* as an element of A^k where A is the domain approximation used in the CSP. A store is a tuple of variable domains: A store D defines a domain $D(X)$ for each variable X of the CSP. A store also constitutes a set of assignments.

It is convenient to be able to denote the smallest approximate domain enclosing any given set of values; The function $apx_A : 2^{\mathcal{U}} \to A$ does it. This function is also extended element-wise to stores: $(2^{\mathcal{U}})^k \to A^k$. If D is a store with domains in $2^{\mathcal{U}}$ (no approximation, $D \in (2^{\mathcal{U}})^k$) then $\forall X \in \mathcal{X}$: $apx_A(D)(X) = apx_A(D(X))$.

Example 2.1. For integers ($\mathbb{Z}$), the enumerated domain is $2^{\mathbb{Z}}$ denoting all subsets of the set $\mathbb{Z}$, and an integer variable X can have the following domain $D(X) = \{-1, 3, 5\}$. A common domain approximation is the integer intervals $A = \{[i..j] | i, j \in \mathbb{Z}\}$[1]. The set {-1,3,5} is not part of the integer intervals and cannot be the approximate domain of X if this approximation is used. Instead, the smallest enclosing interval $[-1..5] = apx_A(\{-1, 3, 5\})$ would be its domain $D(X)$.

A CSP over the variable set $\mathcal{X}$ can be formalized as a pair (C, D) where C is a set of constraints and D is a store.

Let C denote a set of constraints c_i each dealing with a subset $scope(c_i)$ of the variables in $\mathcal{X}$. A constraint can be viewed as a relation, or a set of tuples of values which satisfy the constraint. As in [11], to simplify this formal presentation, we consider that all constraints deal with the whole set $\mathcal{X}$ of variables and can be considered as a set of valid assignments instead of a set of tuples. Indeed, any constraint can be viewed as a constraint with an extended scope $\mathcal{X}$: if a variable is not part of the scope of the original constraint, it can take any value. We write $s \in c$ to state that s is an assignment satisfying the constraint c.

An assignment s is a *solution* to the CSP if it satisfies all the constraints. More formally, $\forall c \in C : s \in c$. Or, equivalently, $s \in (\cap C)$. We denote the set of solutions of a CSP (C, D) by $Sol(C, D)$; it is the set of assignments among D which satisfies all the constraints: $Sol(C, D) = (\cap C) \cap D$

Two CSPs (C, D) and (C', D') over the same set $\mathcal{X}$ of variables are *equivalent* if their set of solutions is equal: $Sol(C, D) = Sol(C', D')$.

A CSP (C, D) is *failed* if it has no solutions ($Sol(C, D) = \emptyset$). A common case of failed CSP is when $D(X) = \emptyset$ for one variable X of the CSP.

[1]If $i > j$, $[i..j] = \emptyset$. If $i = j$, $[i..j] = \{i\}$.

A constraint c is *entailed* by the CSP (C, D) if it is a logical consequence of the CSP: $Sol(C, D) \subseteq c$. Detecting entailment might be as hard as solving the CSP. Another more common notion is that of entailment by the store (called domain entailment in [84]) : c is entailed by the store D iff $Sol(\emptyset, D) \subseteq c$. In practice, it is not constraint entailment but propagator entailment which is used; This notion is defined below.

2.1.2 Filtering Algorithms

Each constraint is associated with a filtering algorithm (also called propagator) aiming at reducing the domain of the variables in its scope without removing solutions. The process of reducing the domains is called filtering, pruning or propagation. It can be formalized by a filtering function which transforms a CSP into an equivalent CSP while reducing the domains.

Let (C, D) be a CSP over the set of variables $\mathcal{X}$ with a domain approximation A. A filtering function $f \in A^k \to A^k$ must have the following properties:

Property 2.2. *[11] Let D_1 and D_2 be stores and let f denote a filtering function associated with the constraint c. f must satisfy the following properties:*

- *Contractant:* $f(D_1) \subseteq D_1$
- *Monotone:* $D_1 \subseteq D_2 \Rightarrow f(D_1) \subseteq f(D_2)$
- *Correct:* $D_1 \cap c \subseteq f(D_1)$

Example 2.3. Consider the constraint $X_1 < X_2$ with domains $D(X_1) = [0..5]$ and $D(X_2) = [-2..3]$. As no solution exists with $X_1 \geq 3$ or $X_2 \leq 0$, we can filter the domains of the two variables to $D(X_1) = [0..2]$ and $D(X_2) = [1..3]$. This example of filtering is contractant and correct.

A filtering function f is *entailed* by the store D if every smaller store is a fixed point of the function: $\forall D' \subseteq D : f(D') = D'$. Programs implementing filtering functions are called *propagators* and entailment of the implemented filtering function is also called *propagator subsumption.*

Local Consistency

In the previous example, any stronger filtering of the variable domains would remove solutions to this constraint and would thus not be correct. This filtering is optimal. The notion of *optimal filtering* ca be formalized in the following way: Let f_c be a filtering function for constraint c for a domain approximation A. f_c is optimal iff $\forall D, f_c(D) = apx_A(Sol(\{c\}, D))$.

The amount of filtering done by a propagator can be characterized by properties relating the set of solutions and the set of values in the domains. Such a set of properties is called the level of local consistency of a CSP. The two most used local consistency notions in CP are arc-consistency (also called domain consistency) and bounds consistency (sometimes called interval consistency and for which several distinct and incompatible definitions have been presented [108]).

A CSP with enumerated domains is said to be generalized arc consistent (GAC [14]) if for each constraint, each value of each variable domain appears in at least one tuple of the constraint. In other words, a CSP is generalized arc consistent if the store is a fixed point of an optimal filtering function for each individual constraint. The term arc consistency comes from the modeling of a binary CSP – a CSP containing only constraints on two variables – as a graph. For CSPs containing N-ary constraints the terms hyper-arc consistency, generalized arc consistency or domain consistency are used instead. For unary constraints the term node-consistency is used.

Example 2.4. Consider the constraint $X_1 = X_2 - 1$ with domains $D(X_1) = \{0, 1, 7\}$ and $D(X_2) = \{1, 4, 8\}$. Computing arc consistent domains for this constraint leads to $D(X_1) = \{0, 7\}$ and $D(X_2) = \{1, 8\}$.

Arc consistency can also be modeled as an optimization problem: Let (C, D) denote a CSP, create another CSP (C', D') which contains the following additional Boolean variables $B_{i,j}$ with $D'(B_{i,j}) = [0, 1]$ for each pair of variable i and value j. The set C' of constraints of the new CSP contains the additional constraints $B_{i,j} \Leftrightarrow X_i = j$ which link these variables to the original variables of the CSP. Computing arc consistent domains for (C, D) then amounts to find the maximum value for each $B_{i,j}$ variable subject to C' and D'. For instance, if the solution to the problem of maximization of variable $B_{1,7}$ for the CSP (C', D') is 0 then 7 must be excluded from the domain of X_1 in order for the CSP (C, D) to be arc consistent.

Another notion of local consistency, bound consistency for an arithmetic constraint c corresponds to the following property: for each variable X_i, for each value b in $\{\min(D(X_i)), \max(D(X_i))\}$ there exists an real assignment s, a solution to the continuous relaxation of c, with $b = s(X_i)$ and for each other variable X_j, $s(X_j) \in [\min(D(X_j)), \max(D(X_j)]$.

Example 2.5. Consider the same constraint $X_1 = X_2 - 1$ with the same domains $D(X_1) = \{0, 1, 7\}$ and $D(X_2) = \{1, 4, 8\}$. This constraint is already bounds consistent. The min and max of both domains can be extended to a real solution of the constraint. On the other hand, for a constraint $X_1 = 2X_2$, and the domains $D(X_1) = [0, 7]$ and $D(X_2) = [1, 8]$, bounds consistent domains are respectively $[1, 4]$ and $[2, 8]$. Indeed, value 1 for X_2 can be extended to a solution with $X_1 = 0.5$ but value 0 for X_1 cannot. Hence the minimum of X_1 must be set to at least 1, excluding the solution with $X_1 = 0.5$. For the upper bound, the values 5, 6 and 7 of X_1 correspond to assignments outside the domain of X_2 and must be removed.

2.1.3 Search and Execution Model

The domains of variables in a CSP define a search space, the space of all potential assignments of values to variables. The goal is to find an assignment satisfying all the constraints (or all of them or the best one according to some optimization criterion). The process is called solving the CSP.

In global search, the CSP is solved by decomposing the problem into smaller subproblems until the subproblems are solvable in a straightforward manner. This process could explore the search space exhaustively. But as the size of this space is exponential in the number of variables, only a small part of it can be explored in practice. The reduction of domains done by filtering algorithms shrinks the search space that needs to be explored while guaranteeing that no solution is lost.

Execution Model

The general schema of the search algorithm is described in algorithm 1. It maintains a list of available unexplored subproblems (called *Fringe* in the algorithm), picks one and runs the constraint propagation step on it. Then, if the space is not solved or failed, it applies a choice step to compute additional subproblems (called alternatives). Notice that the way alternatives are computed and the order in which they are explored is not specified by this search skeleton. It is up to the user to choose an appropriate search algorithm and heuristic.

In the constraint propagation step, filtering algorithms are run according to algorithm 2 until the domains are reduced to a fixed point of all filtering functions associated to the constraints of the CSP. This algorithm is taken from [11] and is an adaptation of classical arc consistency algorithms such as AC-3 [79]. It is the type of constraint propagation algorithm used in many solvers.

```
Data: $(C, D)$ a CSP, $F$ a set of filtering functions $f_c$ for each $c \in C$.
      Let $S = Sol(C, D)$ be the set of solutions to this CSP.
Result: $S$ is returned
begin
    $Fringe \longleftarrow \{(C, D)\}$
    $S \longleftarrow \emptyset$
    while $Fringe \neq \emptyset$ do
        pop $(C', D')$ from $Fringe$
        $<$ Apply Constraint Propagation on $(C', D')$ $>$
        if $(C', D')$ is solved then
            $S \longleftarrow S \cup D'$
        else if $(C', D')$ is not failed then
            $Fringe \longleftarrow Fringe \cup <$ Compute alternatives of
            $(C', D') >$
        end
    end
end
```

Algorithm 1: Global Search Skeleton

```
Data: $(C, D)$ a CSP, $F$ a set of filtering functions $f_c$ for each $c \in C$.
      Let $S = Sol(C, D)$ be the set of solutions to this CSP.
Result: $D$ is updated such that $(C, D)$ is a CSP equivalent to the
        original ($S = Sol(C, D)$ after the update) and such that
        each $f_c$ is at fixed point :$\forall c \in C : f_c(D) = D$.
begin
    $W \longleftarrow C$
    while $W \neq \emptyset \wedge \forall X \in \mathcal{X} : D(X) \neq \emptyset$ do
        pop $c$ from $W$
        $D' \longleftarrow D$
        $D \longleftarrow f_c(D)$
        if $D \neq D'$ then
            $W \longleftarrow W \cup \{c | \exists X \in scope(c) \wedge D(X) \neq D'(X)\}$
        end
    end
end
```

Algorithm 2: Simple Constraint Propagation Algorithm (adapted from [11])

Assumed Services

The implementation of CSPs is assumed to provide the following services:

- read/write access to the domain representation (D) of the variables. The write operation is assumed to be contractant: only a smaller domain can be reassigned to a variable.
- operators to manipulate domain representation (at least the computation of the intersection of two domains)
- the ability to add new constraints to the set of constraints.

The feature of removal of subsumed constraint propagators is not mandatory but helpful and subsumption detection will be considered when designing filtering algorithms. We do not use constraint rewriting operators [11] which allow the replacement of a subset of the constraints of the CSP by another set of constraints.

Notation

Syntactically, when we write

$$D(X_1) \longleftarrow D(X_1) \cap D(X_2) \tag{2.1}$$

we mean that the $\cap$ operation is applied to the current value of the domains of variables X_1 and X_2 and that the result of this operation is assigned as the new domain of X_1 in the store. Also when referring to a domain abstraction using intervals (*e.g.* intervals of integers, intervals of real values and set or graph intervals) we use the following notation for the bounds: $\underline{D}(X_1)$ for the lower bound of the domain and respectively $\overline{D}(X_1)$ for the upper bound; $D(X_1) = [\underline{D}(X_1), \overline{D}(X_1)]$.

When a filtering rule must only be applied in some circumstances, we use the following notation:

$$\frac{\mathit{Guard}}{\mathit{Rule}}$$

where *Guard* is a Boolean expression over the current value of the domains and *Rule* is a filtering rule – a domain assignment such as the rule 2.1.

Search Heuristics

In the problem decomposition step, several alternatives are considered, each consisting in adding a constraint to the CSP. For the search to be exhaustive, the disjunction of these choice constraints must be entailed by the constraint store.

In practice, the constraints added are often simple constraints which only affect the domains and need not be added to the set of constraints. There can be any number of alternatives at each choice point. Three common strategies are binary split (where the two alternatives are $X_i \leq a$ and $X_i > a$), enumeration (the two alternatives are $X_i = a$ and $X_i \neq a$) or labeling where each value of the domain of a variable is tried (the alternatives are $X_i = a$, $X_i = b$, $\ldots$). In these cases, the search tree is also determined by the order of variables. For instance, it is possible to branch on values of variable X_1 then X_2 or the other way around. At each choice point the value ordering can be chosen too *e.g.* $X_1 = a$ or $X_1 = b$ could be tried first. This choice of variable and value ordering can be done beforehand or dynamically during the search. The latter alternative can take into account the current state of the domains and constraints at each choice point.

When the shape of the search tree (*i.e.* the alternative computation function) is determined, there remains to decide the way to explore this search tree: It can be explored in any order. The most common order is depth-first but other search strategies have been proposed such as A* [61] or limited discrepancy search [62]. In the case of depth first search, the fringe is implemented as a stack on which new alternatives are pushed and from which the latest pushed alternative is popped for constraint propagation.

Note that it is not necessary to actually represent the CSPs contained in the fringe in memory, it is only necessary to be able to reconstruct one in order to be able to apply the constraint propagation on it.

2.1.4 Beyond Finite Domains

A computation domain is first determined by a type of values, *e.g.* integers, sets, multisets or tuples. Then it is determined by a set of constraints needed to express and solve problems with this computation domain. Finally, it is determined by the domain abstraction: a practical model for representing the variable domains.

The design of a computation domain needs to consider first a set of mathematical properties of the values it models. These properties are exploited to design the domain abstraction and to design efficient constraint propagators. As variable domains can be seen as a communication channel between filtering algorithms, the domain abstraction characterizes the kind of information that can transit in this channel and it must be designed to provide useful information to the filtering algorithms.

There are several reasons for adding new domains in CP. The first is the need for an efficient handling of a class of problems which were either not modelable or not easy to model without the new domain. The best example is the continuous domain, the reals. The domain of a real variable is an interval bounded by floats. To deal with undetermined real values, the traditional calculus is extended to an interval calculus in which operators deal with intervals instead of values. Constraints are built on these operators and narrow domains while always enclosing the actual solutions. Branching is done by splitting intervals.

In theory, for expressing any discrete domain, all we need is integer domains (sets or graphs can be describe with Booleans; see section 3.2). But in a practice, abstractions ease the design of smaller and more robust models. Another advantage is the possibility to tune the constraint solver to use effectively the structure inherent to these higher level variables. Finally, when designing filtering algorithms, high level abstractions ease the mathematical reasoning and clarify the proofs of properties about these high-level structures (see chapters 4 and 5).

Existing Domains

Apart from integers and Booleans (which are usually encoded as 0/1 integers), the most common discrete domain is sets. They have been introduced by Gervet in [55] which integrated them in ECLIPSE under the name CONJUNTO. They had been implemented in ILOG [90] and were later implemented in Mozart [83]. In [55], the domain abstraction used for sets is set intervals. Set intervals are intervals of the partial order defined by set inclusion. As depicted in figure 2.1, set intervals are defined by two bounds: the greatest lower bound (glb), the set of all elements which *must* belong to the value of the set variable and the least upper bound (lub), the set of all elements which *can* belong to the value of the set variable. Obviously, the glb is part of the lub.

More formally, a set interval $[s_L, s_U]$ defines the following (possibly empty) set of sets : $\{s | s_L \subseteq s \subseteq s_U\}$. Set intervals define a domain abstraction as they allow to represent single sets ($[s, s]$) and are closed under intersection : $[s^1_L, s^1_U] \cap [s^2_L, s^2_U] = [s^1_L \cup s^2_L, s^1_U \cap s^2_U]$.

The basic constraints over set variables and their filtering rules are detailed below. We present the filtering rules for three representative constraints, the interested reader is referred to [55] for the filtering rules of other constraints.

Interval $S \in [s_1, s_2]$ The set variable S is constrained to belong to the set

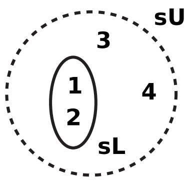


Figure 2.1: A set interval defines a set of sets. The lub set s_U defines the set of all possible values and the glb set s_L defines the values known to be included in every set. The set interval $[\{1,2\},\{1,2,3,4\}]$ represents the set of sets $\{\{1,2\},\{1,2,3\},\{1,2,4\},\{1,2,3,4\}\}$.

interval $[s_1, s_2]$ where s_1 and s_2 are fixed sets. As the domain abstraction is a set interval, this constraint is directly encoded in the variable domain (by intersecting the current interval domain $[\underline{D}(S), \overline{D}(S)]$ of the variable S with $[s_1, s_2]$). The constraint is then entailed.

Intersection $S_1 = S_2 \cap S_3$ This constraint links three set variables S_1, S_2 and S_3 and holds if S_1 is the intersection of S_2 and S_3. Its optimal filtering rules are the following:

1. The values which are for sure in S_2 and S_3 must be included in S_1
$$\underline{D}(S_1) \longleftarrow \underline{D}(S_1) \cup (\underline{D}(S_2) \cap \underline{D}(S_3))$$
2. The values which are for sure in S_1 must be included in S_2
$$\underline{D}(S_2) \longleftarrow \underline{D}(S_2) \cup \underline{D}(S_1)$$
3. The values which are definitely not in S_2 or not in S_3 cannot belong to S_1
$$\overline{D}(S_1) \longleftarrow \overline{D}(S_1) \cap \overline{D}(S_2) \cap \overline{D}(S_3)$$
4. The values which are in S_3 for sure and which are certainly not in S_1 cannot belong to S_2
$$\overline{D}(S_2) \longleftarrow \overline{D}(S_2) \setminus \big(\underline{D}(S_3) \setminus \overline{D}(S_1)\big)$$
5. The rules for updating $D(S_3)$ are symmetric to those for S_2.

If one of these rules computes an empty domain, the constraint fails. The propagator is entailed only when all domains are fixed. To make an optimal filtering algorithm out of these rules, one has to apply them until they reach a fixed point. It can be shown that a single application of these updating rules in the order they are presented (first prune lower bounds then prune upper bounds) leads to a fixed point.

Union $S_1 = S_2 \cup S_3$ This constraint links three set variables S_1, S_2 and S_3 and holds if S_1 is the union of S_2 and S_3.

Difference $S_1 = S_2 \setminus S_3$ This constraint links three set variables S_1, S_2 and S_3 and holds if S_1 is S_2 minus S_3.

Cardinality $I = \#S$ This constraint holds if the integer variable I is equal to the cardinality of the set S. Its optimal filtering rules are the following:

1. The values of I must be enclosed by the cardinality of the set bounds

$$D(I) \longleftarrow D(I) \cap [\max(\underline{D}(I), \#\underline{D}(S)), \min(\overline{D}(I), \#\overline{D}(S))]$$

2. If the cardinality is fixed and corresponds to one of the bounds then the set is that bound:

$$\frac{D(I) = \{i\} \wedge \#\underline{D}(S) = i}{\overline{D}(S) \longleftarrow \underline{D}(S)}$$

$$\frac{D(I) = \{i\} \wedge \#\overline{D}(S) = i}{\underline{D}(S) \longleftarrow \overline{D}(S)}$$

These rules, applied in this order, compute a fixed point. The constraint is entailed when the variables are fixed.

Membership $I \in S$, $I \notin S$ The first constraint holds if the value of the integer variable I belongs to the set value of the set variable S. The second is its negation. The filtering rules for the membership constraint are the following:

1. The values which are for sure not in S cannot be the value of I

$$D(I) \longleftarrow D(I) \cap \overline{D}(S)$$

2. If I has a fixed value, this value must belong to S

$$\frac{D(I) = \{i\}}{\underline{D}(S) \longleftarrow \underline{D}(S) \cup \{i\}}$$

The constraint is entailed if all possible values of I already belong to S for sure: $D(I) \subseteq \underline{D}(S)$

Multiple domain abstractions have been described for sets. Set cardinality (number of elements) also defines a partial order among sets and has been used in conjunction with sets intervals in [2]. The total lexicographic order has also been investigated with an hybrid lattice-lexicographic representation [104]. Recently, a domain abstraction based on the length-lexicographic total order[2] has also been proposed in [56]. Finally, an enumerated domain based on binary decision diagrams (BDD) has been used in [63].

Other domains such as multisets (bags) or sequences – strings, tuples, lists – have also been proposed. In CLPS [77], sets, multisets and lists domains are represented with an enumerated domain. Otherwise, they are most of the time implemented as a vector of integer variables. Then, these domains are characterized by a set of integer constraints with a semantic dedicated to strings, list or multisets. For instance the period [10] constraint is an integer constraint which holds when the list of integer variables in its scope models a periodic list of values. The Same [9] constraint holds when two lists of integer variables model the same multiset. In the same spirit, [13] presents disjoint, partition and intersection constraints for multisets.

In chapter 3, we describe the graph intervals which have a lattice structure similar to that of sets.

Towards the Introduction of a Graph Domain

Graphs were already present in ALICE [76], a seminal constraint programming system, with the constraints *Hamiltonian path* and *circuit*. A set of predicates and rules to handle graphs in Constraint Handling Rules (CHR) [44] is described in [39]. They have been mentioned as first class objects in CLP as a special kind of relation in [53]. In that work, relations are finite sets of pairs and graphs are modeled as relations between a set of nodes and itself. The absence of nodes in that domain representation has some shortcomings for the communication between constraints (see section 3.2.3). Path variables were mentioned in [78] to solve a network design problem. We presented the concept of graph domain variable in [29] and published it

[2]The order of words in a crossword dictionary

in [35, 37]. The path variables of [78] became the digraph variables of ILOG and were presented in [103].

Many practical problems can be modeled using graphs, and CP has been applied to many such problems. For instance, vehicle routing [19], the traveling salesman problem (TSP) [42, 22, 85], network design [78, 1] and bio-informatics [7, 50]. However these problems were modeled using integer and Boolean variables. Many efforts have been spent on discovering and implementing pruning rules for graph properties and many such rules have been rediscovered over the years, mostly for paths and circuits. We believe that one cause for this fact is the co-existence of several possible models for representing a graph using integers, the absence of a thorough study of graph constraints and of a rich collection of graph constraints in available constraint solvers.

We show in this thesis how the introduction of a graph computation domain can ease the development of such pruning rules, serve as a corpus of such rules and present graph models, graph variables and filtering algorithms for constraint programming over graphs.

2.1.5 Prerequisites and CSP Notation

In the rest of this thesis and for the CP(Graph) implementation, we assume the availability of several constraints over Boolean, integer and set variables. We also consider that Boolean variables are encoded as 0/1 integer variables.

Classical Constraints

We assume the availability of the following basic integer constraints:

- $I_0 = I_1$: equality
- $I_0 \neq I_1$: difference
- $I_0 \leq I_1$: order
- $I_0 = I_1 + I_2$: sum
- $I_0 = I_1 * I_2$: product

We assume the availability of a linear arithmetic constraint of the following form with b and $a_i (0 \leq i \leq k)$ a set of integer constants, and $X_i (0 \leq i \leq k)$ a set of integer variables.

$$\sum_{0 \leq i \leq k} a_i X_i \leq b$$

We also use the `alldiff` constraint which holds if all its parameters are pairwise distinct.

Classical Boolean constraints:

- $B_0 = \neg B_1$ the “not” relation.
- $B_0 = B_1 \vee B_2$ the “or” relation.
- $B_0 = B_1 \wedge B_2$ the “and” relation.
- $B_0 = B_1 \Rightarrow B_2$ the “implication” relation.
- $B_0 = B_1 \Leftrightarrow B_2$ the “equivalence” relation.
- $B_0 = B_1 \not\Leftrightarrow B_2$ the “XOR” relation.

Reified integer constraints (a reified constraint is obtained by identifying the truth value of a constraint with a Boolean variable):

- $B \Leftrightarrow (I_0 = I_1)$: reified equal ((B, I_0, I_1) can be assigned the values $(0, 3, 4)$ or $(1, 3, 3)$ for instance).
- $B \Leftrightarrow (I_0 \leq I_1)$: reified no greater than.

The set constraints presented in the previous section and their reified version are assumed to be available too.

CSP Notation

To alleviate the notation, we express the CSPs in a formal mathematical syntax rather than a programming language syntax. The logic we use is a subset of monadic second order logic. That logic allows quantification over elements and sets of elements. The constraints are expressed by using either constraint predicates or relational symbols. When the context allows no ambiguity, we can use a relational symbol to denote a reified constraint. For instance, we might use $\subseteq$ to denote either the subset constraint over two sets or its reification $Subset(S_1, S_2, B)$ which holds if B, a Boolean variable, is equal to the truth value of the relation $S_1 \subseteq S_2$. In a CSP expression, all free variables are existentially quantified. We use the symbol $\equiv$ to denote a decomposition of the constraint on the left side into a conjunction of constraints on the right side.

We naturally use the symbol $\forall$ for universal quantification. As universal quantification is not supported in most CP frameworks, we give a translation of this expression into two semantically equivalent models using constraints available in most of the frameworks. Let S be a set variable and C a constraint with an integer parameter x, or its decomposition in a logical expression with a free variable x. We use the expression $\forall x \in S : C(x)$ to express that the constraint C must hold for all elements in the set variable S.

Two translations of this notation are presented, the first has a consistency level higher than the second. The first translation uses reified constraints; these are constraints augmented with a Boolean parameter which value is true if and only if the relation modeled by the constraint holds. For instance, $X_0 \leq_B X_1) \equiv X_0 \leq X_1 \Leftrightarrow B$. A filtering algorithm for a constraint C must be augmented with a filtering algorithm for the negation of this constraint to obtain a reified propagator.

Our first translation of the notation $\forall x \in S : C(x)$ amounts to posting, for each $x_i \in \overline{D}(S)$, the constraint $x_i \in S \Rightarrow C(x_i)$ where $\Rightarrow$ is the Boolean implication constraint and $\in$ and C are reified. In a pseudo programming language syntax, each of the constraints $x_i \in S \Rightarrow C(x_i)$ would be:

```
declare SetVar S
declare integer xi
forall xi in upper_bound_values(S):
    new BooleanVar Bri, Bli
    implies(Bli,Bri)
    reified_is_in(xi,S,Bli)
    reified_C(xi,Bri)
```

This translation is only possible if a reified version of the constraint exists in the language or constraint library. Otherwise, an alternate translation can be used: as soon as $x_i \in S$ is known to be entailed, post the constraint $C(x_i)$.

```
declare SetVar S
declare integer xi
forall xi in upper_bound_values(S):
    new BooleanVar Bi
    reified_is_in(xi,S,Bi)
    as_soon_as(Bi,1):
        post(C,xi)
```

While this second translation has the same semantics for fixed variables, that is the same set of solutions, it does not provide the same amount of filtering as the first translation which filters each element x_i from $D(S)$ when it is known that $C(x_i)$ does not hold.

2.1.6 Constraint Programming Systems

Constraint logic programming, that is logic programming augmented with syntactic constraints was first developed in Prolog II [23] with disequations. Finite domains [64] were introduced in the CHIP [33] constraint logic programming system. Other Prolog systems such as GNU Prolog [31] and ECLIPSE [124] were then developed. OPL [65] 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 [89].

Constraint solvers such as ILOG Solver provide a constraint programming API for procedural languages such as C or C++. The language Oz [113, 97] 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 [105]. The Gecode [48] constraint development environment is an open-source C++ library. Its architecture is described in [106, 107] and in section 6.1. The implementation of the CP(Graph) computation domain on top of Gecode is described in chapter 6. A Mozart implementation of a prototype of our computation domain was described in [36].

2.2 Graph Theory

In this section we present graph theory notions which are necessary for the understanding of the rest of this thesis. These notions are used extensively in chapters 4 and 5 which deal with graph constraints. For chapter 3, which introduces the CP(Graph) domain, only basic notions such as graph, path, connected graph or tree are necessary. An index of these graph theory definitions is available at the end of this thesis.

2.2.1 Graphs

A *directed graph*, denoted by $G = (V, E)$ is defined as two sets: V the set of its vertices (or nodes) and $E \subseteq (V \times V)$, the set of its edges (or arcs). Such a directed graph is depicted in figure 2.2. For an edge (u, v), u is called the *source* or *tail* of the edge and v is called the *target* or *head* of the edge.

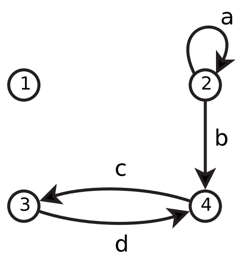


Figure 2.2: Directed graph. Edge (or arc) a is the pair of vertices (or nodes) $(2, 2)$. Edge c is $(3, 4)$ and d is $(4, 3)$.

On the other hand, an *undirected graph* is also denoted by $G = (V, E)$ but its edges are sets of two nodes: $E \subseteq \{\{u, v\}|u \in V \wedge, v \in V \wedge u \neq v\}$. One is presented in figure 2.3.

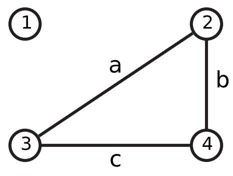


Figure 2.3: Undirected graph. Edge a is $\{2, 3\}$ and b is $\{2, 4\}$.

A *symmetric* directed graph is a graph in which $(u, v) \in E \Leftrightarrow (v, u) \in E$.

Note that an undirected graphs corresponds to a unique symmetric directed graph: Each edge $\{u, v\}$ of the undirected graph corresponds to two edges (u, v) and (v, u) of the directed graph.

The *reverse* of an edge (u, v) of a directed graph is the edge (v, u). The reverse G' of a directed graph G is the directed graph composed of all reverse edges of G. A symmetric graph is its own reverse graph.

Finally, every directed graph can be viewed as an undirected graph by forgetting the orientation of its edges, removing loops and merging all multiple resulting edges (see figure 2.4 for an example). We call this undirected graph the *undirected version* of the directed graph.

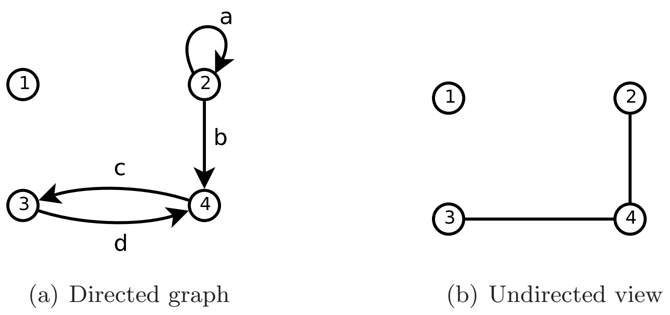
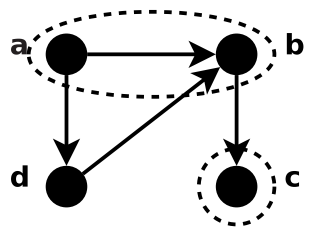


Figure 2.4: Undirected view of a directed graph

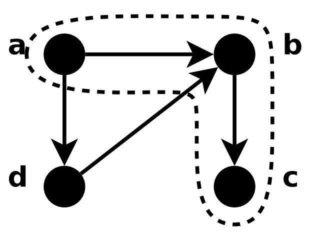
Here are some important notions about graphs: The *order* of a graph is its number of vertices, the *size* of a graph is its number of edges. In an undirected graph $G = (V, E)$, the *neighbors* of a vertex u is the set of all other vertices forming an edge with u : $\{v|\{u, v\} \in E\}$. Such vertices are said to be *adjacent* to u. The number of neighbors of a vertex u is called the *degree* of u.

In a directed graph, the *out-neighbors* of a vertex u in a graph $G = (V, E)$ is the set of vertices to which an arc emerges from u: $\{v|(u, v) \in E\}$. Symmetrically, the *in-neighbors* of a vertex v in the graph G is the set of vertices from which there is an arc incident to v: $\{u|(u, v) \in E\}$. Similarly to the notion of *degree* of a vertex in an undirected graph, the *out-degree* and *in-degree* of a vertex in a directed graph are the cardinality of the respective sets of neighbors.

We denote the *subgraph* relation using the same symbol as the subset relation: $\subseteq$. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \subseteq G_2 \iff V_1 \subseteq V_2 \wedge E_1 \subseteq E_2$. Figure 2.5(a) presents a graph and one of its subgraphs.



(a) A subgraph with nodes $\{a, b, c\}$

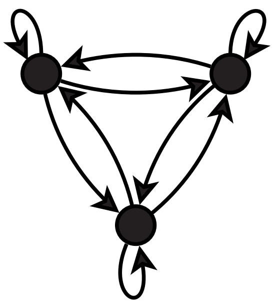


() The subgraph induced by nodes $\{a, b, c\}$

Figure 2.6: Subgraph VS induced subgraph

This simple notion of subgraph should not be confused with the *induced subgraph*: G_1 is an induced subgraph of G_2 if all edges of G_2 incident to vertices of G_1 also belong to G_1: $\forall (u, v) \in E_2 : u \notin V_1 \vee v \notin V_1 \vee (u, v) \in E_1$. Figure 2.6 presents the same graph with an induced subgraph corresponding to the same set of vertices.

The *complete graph* of order n is a graph with n nodes and every possible edges connecting these nodes. Complete directed and undirected graphs of order 3 are presented in figure 2.6.


(a) Directed complete graph of order 3

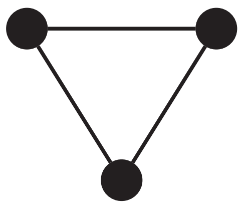

(b) Undirected complete graph of order 3

Figure 2.6: Complete graphs

The *complement* of a graph $G = (V, E)$ is a graph $\bar{G} = (V, \bar{E})$ which contains all edges which are not present in E : $\bar{E} = (V \times V) \setminus E$. Figure 2.7 presents a graph and its complement.

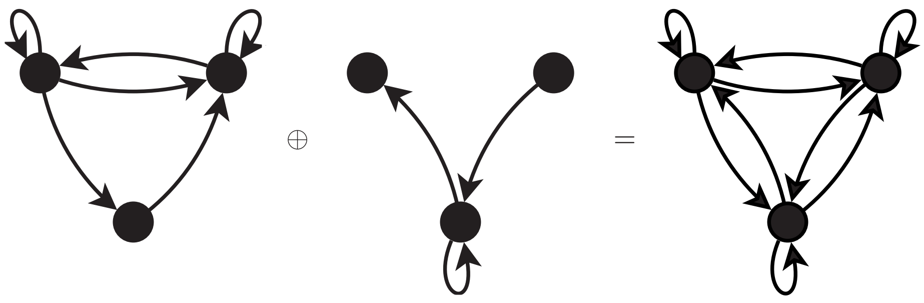


Figure 2.7: The two graphs on the left are complement of each other. They partition a complete graph.

A graph $G = (V, E)$ is *bipartite* if its vertices can be partitioned into two non empty sets V_1 and V_2 such that no edge belongs to $V_1 \times V_1$ or $V_2 \times V_2$. A bipartite graph is presented in figure 2.8.

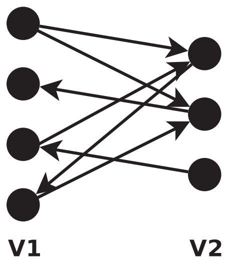


Figure 2.8: A directed bipartite graph and its two sets of nodes V_1 and V_2.

2.2.2 Paths

A *walk* in a graph is a alternating sequence of vertices and edges starting and ending in a vertex; a single vertex is an empty walk. In the graph presented in figure 2.9, (3), $(3, c, 4, d, 3)$ and $(2, a, 2, a, 2, b, 3)$ are walks.

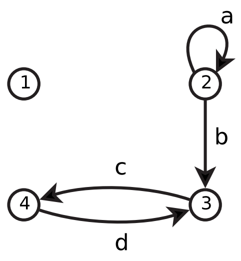


Figure 2.9: Walks in a directed graph. (3), $(3, c, 4, d, 3)$ and $(2, a, 2, a, 2, b, 3)$ are walks.

A walk in which there is no repeated vertex is called a *path*. In figure 2.9, $(2, b, 3, c, 4)$ is the longest path. Some authors allow paths to start and end in the same vertex, we prefer to call that case a *circuit* or *cycle*; $(4, d, 3, c, 4)$ is an example. In this thesis, circuit will be preferred when referring to directed graphs, and cycle used for undirected graphs.

A directed graph with no circuits is called a *directed acyclic digraph* or *DAG*. A dag is presented in figure 2.10. The figure also shows the *topological order* of the nodes in a DAG. In a topological order, every vertex n is associated with a value $o(n)$ and for each arc (u, v), $o(u) < o(v)$.

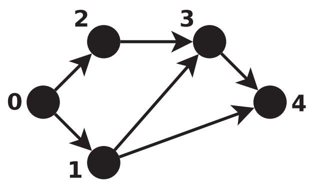


Figure 2.10: Directed acyclic graph and topological order

2.2.3 Connectedness in Undirected Graphs

Two nodes of an undirected graph are said to be *connected* if there exists at least one path joining the two nodes. An undirected graph is said to be *connected* if for every pair of nodes, the nodes are connected.

The *connected components* of an undirected graph are the maximum connected subgraphs of this graph (see figure 2.11).

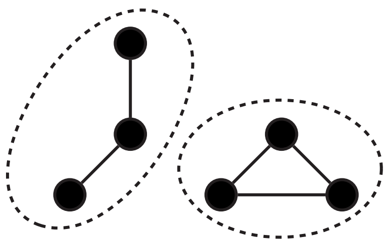


Figure 2.11: Connected components of an undirected graph.

In a connected component some nodes and arcs are necessary for the connectedness property to hold:

A *cutnode* of an undirected graph is a node whose removal disconnects the graph. Similarly, a *bridge* is an arc whose removal disconnects the graph. On figure 2.12, c_1, c_2 and c_3 are cutnodes and b is a bridge.

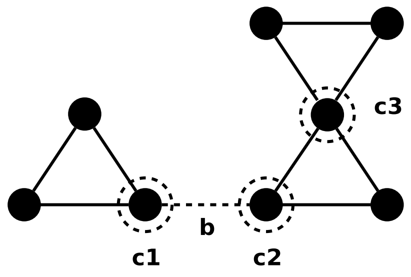
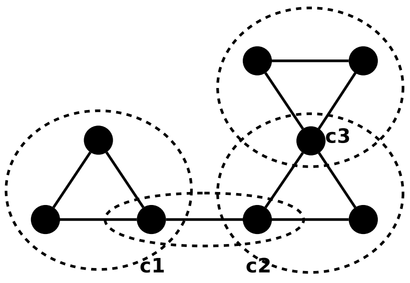
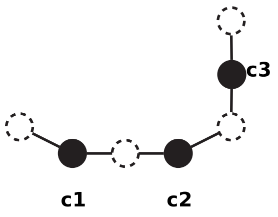


Figure 2.12: Bridge b and cutnodes c_1, c_2 and c_3.

A *biconnected* graph is a graph containing no cutnode and a *2-edge-connected* graph is a graph containing no bridge. As with connected components, the notions of *biconnected components* and *2-edge-connected components* refer to the largest subgraphs having these properties (see figures 2.13(a) for biconnected components and 2.14(a) for 2-edge-connected components).

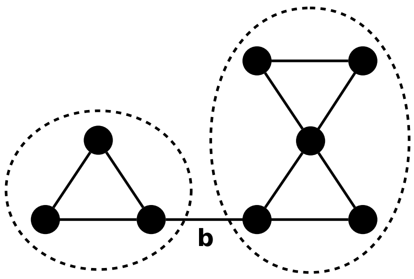


(a) Biconnected components

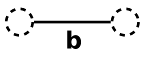


(b) Cutnode-reduced graph

Figure 2.13: Biconnectivity



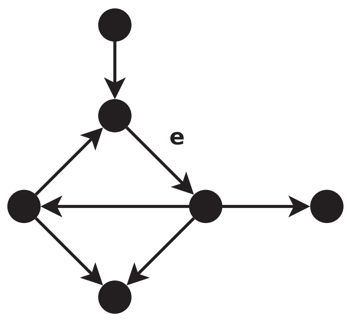
(a) 2-edge-connected components



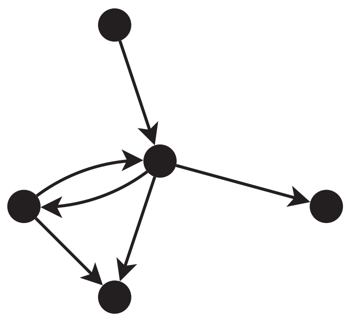
(b) Bridge-reduced graph

Figure 2.14: 2-edge-connectivity

The *contraction* of an edge (u, v) in a graph results in a graph where vertex v is removed and all edges incident to v are replaced by edges incident to u. $G = (V, E)$ becomes $G' = (V \setminus \{v\}, E \cup \{(x, u) | (x, v) \in E\} \cup \{(u, x) | (v, x) \in E\} \setminus \{(x, y) \in E | x = v \vee y = v\})$. This process is illustrated in figure 2.15.



(a) Before contraction



(b) After contraction

Figure 2.15: Contraction of edge e.

A *bridge-reduced forest* t of an undirected graph g results from the contraction of all edges belonging to a 2-edge-connected component of g. It is a forest t whose nodes correspond to each 2-edge-connected component of g and whose edges correspond to bridges linking different components in g. An example is given in figure 2.14(b) where the two 2-edge-connected components of figure 2.14(a) are contracted.

A *cutnode-reduced forest* t of an undirected graph g results from the contraction of all edges belonging to a biconnected component of g which are not incident to a cutnode. It is a forest t whose nodes correspond to each biconnected component and cutnode of g and whose edges link cutnodes to the biconnected components they belong to. An example is given is figure 2.13(b) where every biconnected component of figure 2.13(a) is contracted into one additional node.

It might not be obvious why the cutnode-reduced graph t is a forest. If there is a cycle in t then all of the nodes which belong to the cycle are in the same biconnected component, a contradiction.

2.2.4 Connectedness in Directed Graphs

In a directed graph, we say that u *reaches* v if there exists a directed path from u to v in the graph. We can still say that u and v are connected as there is also an undirected path linking u and v.

A directed graph is said to be *weakly connected* if its undirected version obtained by forgetting the orientation of arcs is connected. In other words, a directed graph is weakly connected if for each pair (u, v) of its nodes there is a path from u to v regardless of the direction of the arcs along the path.

A directed graph is said to be *strongly connected* if every node reaches every other nodes. In other words, a directed graph is strongly connected if for each pair (u, v) of its nodes there is a directed path from u to v.

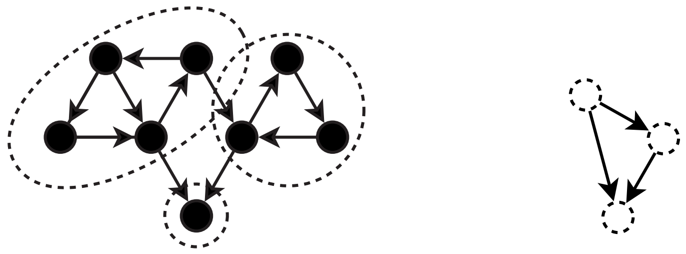

(a) Strongly connected components (b) SCC-reduced graph

Figure 2.16: Strong connectivity

As with other notions these properties allow to define components: *weakly connected components* and *strongly connected components* are the maximum subgraphs with these properties. Figure 2.16(a) presents the strongly connected components of a weakly connected graph.

If we contract all edges in the strongly connected components, we obtain a graph with no circuit which we call the *SCC-reduced graph*. Some authors just call it the *reduced-graph* but as we use several different reduced graphs we use SCC-reduced graph for this one. A SCC-reduced graph of the graph in figure 2.16(a) is presented in figure 2.16(b).

The following notion is the directed graph counter-part of the notion of bridges and cutnodes. A node d is said to *dominate* t from s iff all directed paths from s to t go through d. Node d is then said to be a *dominator* of t from s. This notion also exists for arcs: Arc e is said to dominate t from s iff all directed paths from s to t go through e. Among the dominators of a directed graph all bridges and cutnodes of its undirected version are present. But as illustrated in figure 2.17 there are also other dominators: from source s, node e, a cutnode, dominates c and d. On the other hand, g which is not a cutnode dominates all other nodes and arc (g, b) dominates a.

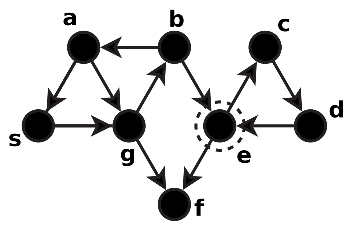


Figure 2.17: Dominators

The *transitive closure* of a directed graph $G = (V, E)$ is a graph $G^+ = (V, E^+)$ such that $(u, v) \in E^+$ if and only if there is a path from u to v in G. In figure 2.18 a graph and its transitive closure are presented. Note that for a undirected graph, the transitive closure would just consist in cliques for each connected component of the original graph.

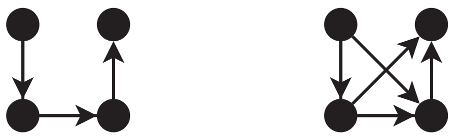


Figure 2.18: A graph (left) and its transitive closure (right).

2.2.5 Trees

A *forest* is an undirected graph with no cycle. A *tree* is a connected forest. Therefore each connected component of a forest is a tree. A forest is depicted in figure 2.19.

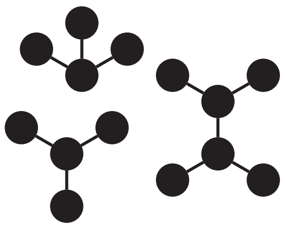


Figure 2.19: A forest made of three trees

A *spanning tree* of a connected undirected graph $G = (V, E)$ is a tree $T = (V, E')$ with $E' \subseteq E$ (see figure 2.20). When dealing with a non connected undirected graph, its *spanning forest* is composed of a spanning tree of each of its connected components. When weights are assigned to each edge of the graph, one might be interested in the *minimum spanning tree*, a spanning tree minimizing the sum of its edge weights.

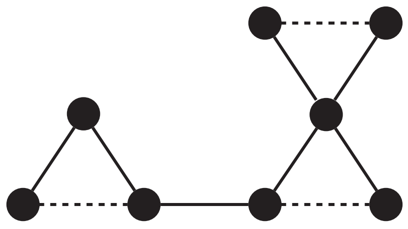


Figure 2.20: A spanning tree of a connected graph (spanning tree with solid edges, non-tree edges dashed)

A *directed tree* is a directed graph whose undirected version is a tree. Specials kinds of directed trees arise when all arcs are directed from a *root vertex* or to this root vertex (see figure 2.21).

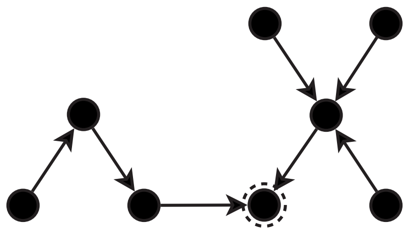


Figure 2.21: A directed tree and its root (circled)

The *shortest path tree* of a graph G is a directed spanning tree whose every edges are oriented from a source vertex s and such that for each node u in the graph, the unique path $s - u$ from the source to u in the tree is a shortest path from s to u in G (see figure 2.22).

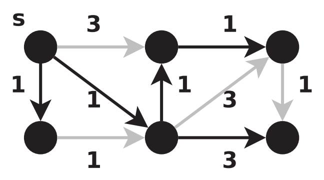


Figure 2.22: Shortest path tree from s. The tree edges are black, others are gray.

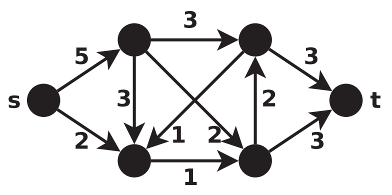
2.2.6 Flows, Cuts and Matchings

Let $G = (V, E)$ be a directed graph and s and t be distinct vertices called the source and the sink. Let $cap : E \to \mathbb{N}_0$ be a function assigning a positive capacity to all edges of the graph. A *(s,t)-flow* (or simply *flow*) is a function $f : E \to \mathbb{N}_0$ assigning a flow value to each edge such that the capacity is respected and the flow is conserved at each vertex other than s and t:

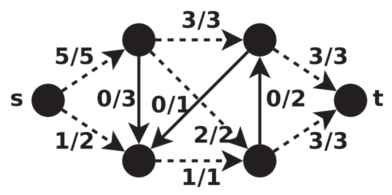
$$0 \leq f(e) \leq cap(e) \qquad \text{for each } e \in E$$

$$\sum_{e|\exists x:e=(u,x)} f(e) = \sum_{e|\exists x:e=(x,u)} f(e) \qquad \text{for each } u \in V \setminus \{s,t\}$$

The value of the flow is the amount of flow leaving the source s and entering the sink t. A capacited graph is presented in figure 2.23(a). In figure 2.23(b) a maximum flow is presented in the following format: an edge label f/c where f is the flow value of the edge and c is the capacity.



(a) Capacited graph



(b) Maximum flow

Figure 2.23: A capacited graph and a maximum flow

A *cut* C is a connected subset of the nodes of the flow graph such that $s \in C$. Edges *crossing* the cut are edges whose source is in C and target is not. The capacity of the cut is the sum of the capacities of the edges crossing the cut. Figure 2.24 highlights a cut and the corresponding crossing edges.

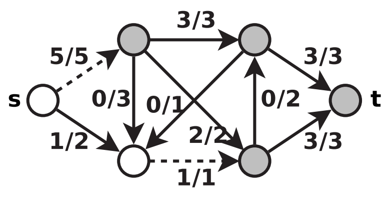


Figure 2.24: A cut (white nodes) and crossing edges (dashed)

The Ford-Fulkerson max-flow-min-cut theorem states that the value of a maximum flow in a graph corresponds to the value of a minimum capacity cut in the graph. The flow in figure 2.23(b) is a maximum flow and the cut in 2.24 is a minimum cut. We use a max-flow-min-cut computation in section 4.5 to compute the minimum weight subgraph of a graph with edge and vertices weights.

In a graph $G = (V, E)$, a *matching* is a subset $M \subseteq E$ of the edges such that every vertex $v \in V$ belongs to at most one edge in M. A node incident to a edge of the matching is said to be *covered* by the matching, otherwise it is said to be *free* (see figure 2.25; the matching is dashed and the circled node is free). Many constraints such as *alldiff* use a *maximum matching* computation, that is either maximum cardinality matching or maximum weight matching when a weight is assigned to each edge.

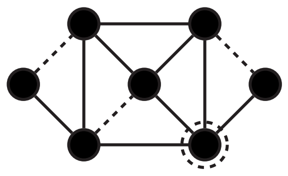


Figure 2.25: A maximum matching (dashed edges). The circled node is free. The others are covered.

Chapter 3

The CP(Graph) computation domain

This chapter describes the CP(Graph) computation domain which introduces graph variables, their domain abstraction, and constraints over these variables in constraint programming. These not only allow a high-level modeling of graph problems as CSPs but are also an useful abstraction when designing a filtering algorithm based on graphs for any type of constraints.

Section 3.1 describes graph intervals used as the domain abstraction for graph variables in CP(Graph) and their properties. Section 3.2 covers CSP models which have been used to express graph problems. These models are compared with graph intervals. Then section 3.3 describes the variable types used in the CP(Graph) framework. They include nodes, arcs and sets of them. Section 3.4 describes three basic constraints which link graph variables with other types of variables in CP(Graph). These constraints are sufficient to express graph problems together with the other variables of CP(Graph) and basic set and finite domain constraints as described in section 3.5.

A top down approach is used in that section to show how combinatorial graph problems can be decomposed into high level graph constraints and how these constraints can be decomposed into kernel graph constraints. A shortcoming of these decompositions is that graph properties related to transitive closure such as connectedness require a large number of constraints in their decomposition.

To conclude this chapter, in section 3.6, we show how CP(Graph) extends beyond graph variables and also applies to filtering algorithms for constraints over integers. We first discuss other models for graphs as well as existing graph constraints defined for these models. Then we show that the use of graph algorithms in global constraints such as GCC or alldiff amounts to applying a filtering algorithm for a graph constraint over a graph interval. Finally, we discuss the generic graph-based filtering algorithm recently proposed in [5] to filter all global constraints of the global constraints catalog [3] and show that the use of graphs in that algorithm directly corresponds to graph intervals.

3.1 Preliminaries on Graph Intervals

As graph variables take graphs as values, the domain of such a variable is a set of graphs. Since an enumeration of all possible graphs for each variable might take a lot of space, a domain abstraction is used instead. Graph intervals are suitable as a domain abstraction as the subgraph relation defines a partial order and a lattice structure. In the rest of this section, we describe formally the properties of graph intervals.

Definition 3.1. A partially ordered set (poset) is a set S together with a binary relation $\leq$ that is

- Reflexive : $\forall x \in S : x \leq x$
- Antisymmetric : $\forall x, y \in S : x \leq y \wedge y \leq x \Rightarrow x = y$
- Transitive : $\forall x, y, z \in S : x \leq y \wedge y \leq z \Rightarrow x \leq z$

Definition 3.2. [16] A lattice is a poset P whose any two elements have a greatest lower bound or "meet" denoted by $x \cap y$ and a least upper bound or "join" denoted by $x \cup y$.

These meet and join operators obey the following laws:

- Idempotent: $x \cap x = x = x \cup x$
- Commutative: $x \cup y = y \cup x$ (and dually for $\cap$)
- Associative: $x \cup (y \cup z) = (x \cup y) \cup z$ (and dually for $\cap$)
- Absorption: $x \cup (x \cap y) = x \cup y$ and $x \cap (x \cup y) = x \cap y$

Definition 3.3. In a lattice L, the *null element* O is the element such that each other element is greater than O: $O \in L \wedge \forall x \in L : O \leq x$. The *universe element* I is the element such that all elements are smaller than I: $I \in L \wedge \forall x \in L : x \leq I$.

Example 3.4. The power set $2^S = \{S' | S' \subseteq S\}$ is a lattice with set intersection as the meet operation and set union as the join operation. It has the empty set $\emptyset$ as the null element, and S itself as the universe element.

Theorem 3.5. *[16] The direct product $L \times M$ of any two lattices is a lattice with the following partial order: $(l_1, m_1) \leq (l_2, m_2) \Leftrightarrow l_1 \leq l_2 \wedge m_1 \leq m_2$.*

As a result of this theorem, the product $\mathbf{N} \times \mathbf{E}$ of two power set lattices $\mathbf{N} = 2^N$ and $\mathbf{E} = 2^E$, the sets of all subsets of N and E respectively, is a lattice. The elements of that lattice are the pairs of sets (N', E') with $N' \subseteq N \wedge E' \subseteq E$. The order relation in this lattice is $(N_1, E_1) \subseteq (N_2, E_2) \Leftrightarrow N_1 \subseteq N_2 \wedge E_1 \subseteq E_2$.

The set of all subgraphs of a given graph $G = (N, E)$ which we denote by the *power graph* 2^G is a subset of the lattice $2^N \times 2^E$. It is the set of all pairs (N', E') of $2^N \times 2^E$ satisfying the constraint $E' \subseteq N' \times N'$.

We now show that the power graph is a lattice by using graph intersection as the meet operation and graph union as the join operation.

Definition 3.6. Graph union is defined as: Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$

Definition 3.7. Graph intersection is defined as: Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$

Proposition 3.8. *For graphs, union and intersection are internal relations: The union of two graphs is a graph and the intersection of two graphs is a graph.*

Proof. The proof consists in seeing that

$$E_i \subseteq (V_i \times V_i),\ i \in (1, 2) \Rightarrow E_1 \cup E_2 \subseteq (V_1 \cup V_2) \times (V_1 \cup V_2)$$

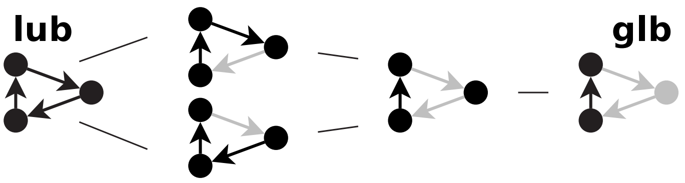
and dually for intersection. □

In the power graph 2^G, the null element O is the empty graph $(\emptyset, \emptyset)$ and the universe element I is the graph G itself.

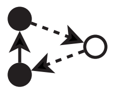
Definition 3.9. [98] If $x > y$ but $x > z > y$ is not satisfied by any z in the lattice, we say that *x covers y*. An element which covers O is called an *atom* and an element which is covered by I is called an *anti-atom*.

In the power graph lattice, atoms are graphs made of a single node while anti-atoms are the universe graph without one of its edges or without one of its isolated nodes if it has some.

Definition 3.10. If $G_1 \subseteq G_2$, $[G_1, G_2]$ is a graph *interval*. It is the set of all graphs G such that $G_1 \subseteq G \subseteq G_2$. A graph interval is a lattice, a lattice of graphs corresponding to a graph interval is presented in figure 3.1.



(a) A graph lattice



(b) The corresponding graph interval

Figure 3.1: The graph lattice shows the set of graphs contained in the graph interval. The graphic codes for the picture of the graph interval is that the lower bound is composed of solid nodes and edges and the rest of the upper bound is composed of empty nodes and dashed edges.

We give a few properties of lattices and intervals:

- if $G_1 \not\subseteq G_2$ then $[G_1, G_2] = \emptyset$.
- $[G_1, G_2] = \{G\}$ iff $G_1 = G_2 = G$.
- $2^G = [\emptyset, G]$

In a finite lattice, all sets S of elements admit a greatest lower bound (glb) $\cap S$ and a least upper bound (lub) $\cup S$ (by associativity of $\cup$ and $\cap$). Note that for any S, $\cap S \subseteq \cup S$. Hence, a given set S of graphs is bounded by the interval $[\cap S, \cup S]$.

$$S \subseteq [\cap S, \cup S] \wedge \nexists G_1, G_2 : S \subseteq [G_1, G_2] \subset [\cap S, \cup S]$$

Graph intervals will be used to model the domain of graph variables.

3.2 Classical Graph Models

Some graph problems and constraints have been solved with CP, with most models based on the finite domain (integers). In this section, we present common CSP models which were used to express graph CSPs and compare them with graph intervals.

We show that these models are very similar to graph intervals and that the most notable difference is the ability to constrain nodes of the graph variable when using graph intervals. We show that the bounds of the smallest enclosing graph interval can be easily extracted from these models. This has two consequences: first, the filtering algorithms which have been designed for these models can be easily adapted to graph intervals by extending them to also deal with nodes. Second, the filtering algorithms presented in this thesis for graph intervals can also be used for these models by abstracting nodes or adding variables to model nodes.

3.2.1 The Models

We call "single successor model" the first graph model which is used for many existing graph constraints such as the `circuit` constraint of OPL [120], the TSP (traveling salesman problem) and TSPTW (TSP with time windows) in [22, 85, 42], and VRP (vehicle routing problem) in [19]. In [7] this model is combined with a tree constraint to model phylogenetic tree construction problems. This model consists in a vector of integer variables. Its graph semantic is that the variable at index i with an assigned value j represents the presence of the arc (i, j) in the graph. In that model every node has exactly one out-neighbor (possibly itself). Any assignment of values to the variables in that model produces a graph composed of cycles and anti-arborescences (trees directed towards a root). Hence only some specific graphs can be represented in this model (see figure 3.2 for an example of such a graph). A

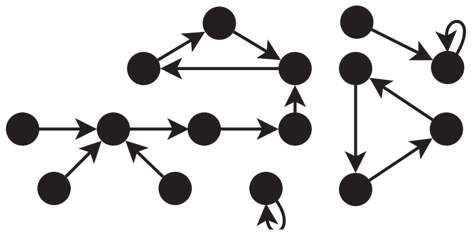


Figure 3.2: An assignment of the single successor model

second model is the Boolean adjacency matrix [36] or edge presence Boolean model. In that model a Boolean variable is used for each potential arc. For some path constraints [110, 21] a graph is implicit in the constraint and only one Boolean variable is used per arc of the original graph (instead of one per pair of nodes). This is also the classical integer formulation of the constrained shortest path problems [38, 42].

This model can also be implemented by using one set variable for each line of the Boolean adjacency matrix. In that case one set is used for each node and it models the out-neighbors of this node.

3.2.2 Smallest Enclosing Interval

These models can be interpreted as a representation of a graph interval. An instantiation of the variables in the models encodes a graph and the set of instantiations allowed by the domains models a set of graphs. These sets of graphs are however only subsets of graph intervals.

The Boolean model allows to represent any graph except graphs containing isolated nodes. It suffices to add a vector of Boolean variables to model the nodes and add adjacency constraints to have a complete graph model as we did in [36]. On the other hand, the single successor model only contains graphs made of disjoint cycles and anti-arborescences which cover a fixed set of nodes.

These models are close to graph intervals and the smallest enclosing graph interval can be built easily. In the Boolean model, the graph composed of the arcs assigned to 1 is the greatest lower bound and the graph composed of all arcs not set to 0 (that is non-fixed or fixed to 1) is the least upper bound. In that model, both bounds of the graph interval can be represented by the Boolean model.

On the other hand, with the single successor model, the least upper bound is the graph composed of all the arcs present in the domain of each integer variable and the greatest lower bound is the graph composed of all integers already assigned, each representing a single arc. In this second model, the lower bound of the graph interval is not representable by the integer model: the integers cannot model that graph as some nodes have no outneighbors in that graph.

3.2.3 On Constraining Nodes

In the graph interval model, nodes can be constrained: They can be added to the lower bound or removed from the upper bound. In many previous graph models, either the set of nodes is fixed (as in some uses of the single successor model) or it is defined implicitly by the set of arcs. In any case, it is not possible to use nodes to adjust the bounds of the model.

Because of the adjacency constraint inherent to every graph ($E \subseteq V \times V$), the smallest element which can be added to the lower bound is an isolated node (or an arc if its endnodes are already present). For the upper bound, the removal of an arc or an isolated node is the finest change that can be made.

In a generic graph model, it is important to be able to enclose the set of solutions as closely as possible in order to provide filtering algorithms with the finest information about the current domains. Therefore, the presence of nodes in the graph model is important. For instance, in the $Connected(G)$ constraint filtering algorithm (section 4.3.2), some cutnodes can be included in the lower bound. Including nodes can result in filtering on the edges: in a path problem, even if the original problem does not care about nodes and does not constrain nodes, edges are linked to nodes via a relation ($\#E = \#V - 1$) which can trigger filtering on the edges.

In conclusion, we showed that these two graph models based on finite domains are close to a graph interval and allow to easily extract the bounds of the smallest graph interval enclosing them. This allows to easily adapt filtering algorithms to deal with one of these models or a graph interval. We also explained why being able to represent nodes in a graph variable domain is important.

3.3 Variables in CP(Graph)

When modeling a graph CSP, it is necessary to be able to reason about nodes and edges of the graphs. Hence, CP(Graph) uses variable types not only for graphs but also for nodes, edges and sets of nodes or edges. As many graph problems are related to weighted graphs, an additional variable type for weight functions is introduced too. All these types and their notation are depicted in table 3.1. The last row of this table describes weight functions. In that row, $\mathcal{N}$ and $\mathcal{A}$ are the universal sets of nodes and arcs. Weight functions are all functions from arcs and nodes to integers. For all CSPs notations in this thesis, small case letters are used to denote fixed, constant values while capital letters denote non-fixed variables, variables whose domain is not a singleton. The type of variables is indicated by the name of the variable according to table 3.1: for instance SA is a set of arcs while SN is a set of nodes.

Type	Representation	Constraint	Constants	Variables
Integer	$0, 1, 2, ...$		$i_0, i_1, ...$	$I_0, I_1, ...$
Node	$0, 1, 2, ...$		$n_0, n_1, ...$	$N_0, N_1, ...$
Arc	$(0, 1), (2, 4), ...$		$a_0, a_1, ...$	$A_0, A_1, ...$
Set	$\{0, 1, 2\}, \{3, 5\}$ ...		$s_0, s_1, ...$	$S_0, S_1, ...$
Set of nodes	$\{0, 1, 2\}, \{3, 5\}$ ...		$sn_0, sn_1, ...$	$SN_0, SN_1, ...$
Set of arcs	$\{(0, 3), (1, 2)\}, ...$		$sa_0, sa_1, ...$	$SA_0, SA_1, ...$
Graph	(SN, SA)[a]	$SA \subseteq SN \times SN$	$g_0, g_1, ...$	$G_0, G_1, ...$
Weight functions	$\mathcal{N} \cup \mathcal{A} \to \mathbb{Z}$		$w_0, w_1, ...$	W_0, W_1

[a] SN a set of nodes and SA a set of arcs

Table 3.1: Variable types of CP(Graph)

3.3.1 Nodes, Arcs and Sets

In table 3.1, nodes are identified with integers. For a given CSP, each node of the universal graph is numbered and is referred to by its number. When describing constraints, we still make the distinction between the node type and the integer type. We assume that nodes are represented by integer variables with an enumerated domain. As in general there is no natural order between nodes in a graph, we did not consider the possibility to represent node domains as intervals but there are probably applications for which this could be beneficial.

By definition, in a directed graph, arcs are pairs of nodes. Hence we consider arcs are encoded as pairs of integers. The domain of an arc variable is a enumerated set of pairs of integers. As constraint solvers might not support pairs as values, these pairs can also be encoded as integers. But this is considered an implementation detail and is abstracted here.

Sets of nodes and arcs are also supported and it is assumed that the set membership constraint can be consistently applied to these variables: the constraints $A \in SA$ and $N \in SN$ must be supported. Hence if an integer encoding of arcs is used, it must be used consistently for the arc type and the set of arcs type.

3.3.2 Graph Variables

Graph variables are variables which are assigned a graph. Their domain is approximated using a graph interval.

As the domain of each graph variable is an interval, each graph variable of a CSP need to have a predetermined initial upper bound.

Definition 3.11. For a given CSP we define the *universe graph* of this CSP as the union of all initial upper bound graphs of its graph variables.

As depicted in table 3.1, graph variables can be viewed as two sets variables linked by an adjacency constraint: Their set of arcs must be included in the cross product of their node set with itself.

The choice of graph intervals as the domain approximation for graph variables seems adequate for the following reasons. First, it is similar to the interval abstraction for sets. This similarity eases the cooperation between the two domains and allows to transfer previous knowledge about sets intervals to graphs intervals. The graph interval model has also many similarities with previous integer and set models for graphs in CP (as detailed in section 3.2). This allows to adapt existing graph constraints to graph intervals. The lattice structure of graph intervals allows the constraint designer to reason about monotonicity properties of the constraint which provide elegant proofs of algorithms (see *e.g.* the downgrade lemma in section 5.6). Another advantage of this model is its simplicity: it is the simplest abstraction providing information about possible and mandatory elements in the graph variable, two notions which are common in practical problems and in graph theory (see *e.g.* dominators, bridges or cutnodes).

This simple model seems to be adequate for a general purpose graph domain abstraction. This does not mean that it is the best model for all applications: The choice of a domain abstraction determines the available communication channel between filtering algorithms. For some specific applications, this communication channel might cause a large gap between the filtering obtained by the conjunction of constraints and the filtering possible in theory when considering these constraints globally. Then another domain abstraction could be considered for those problems if it allows a better collaboration between filtering algorithms (see for instance the recent length-lex domain approximation for sets [56]).

3.3.3 Undirected Graphs

We model undirected graphs as directed graphs. Two different models allow to view directed graphs as undirected: either the orientation of arcs is forgotten and opposite arcs are merged (this is the undirected view of directed graphs presented in section 2.2) or each undirected arc is modeled as two opposite directed arcs (each undirected graph corresponds to a unique symmetric directed graph).

The latter model is used in CP(Graph) as it provides a simple and unique directed graph representation of undirected graphs. This model also has the advantage that reachability properties in the undirected graph are conserved in its directed representation: node i is reachable from j in the symmetric graph iff it is reachable in the undirected graph. Finally, modeling undirected graphs as directed graphs allows to apply directed graph constraints to these undirected graphs. On the other hand, to apply undirected constraints to the undirected version of a directed graph, we provide a constraint $Undirected(G, G_u)$ which states that G_u is a symmetric graph modeling the undirected version of G (see section 4.3.1).

3.3.4 Weight Function Variables

Many graph problems deal with weighted graphs. Such graphs have weights associated to their nodes and/or edges. We model such weighted graphs with a graph variable and a weight function variable which associates weights to the nodes and arcs of the graph.

For some problems, the weights are part of the problem formulation and are constant (many path finding problem). In other problems only an interval is given for weights in the problem formulation. These weights might be part of the solution such as in some inverse optimization problems or in sensitivity analysis. Or the weights are intervals simply because it allows to model uncertainty about these weights (such as in the robust spanning tree design problem).

We use map variables [30] which underlying implementation relies on a vector of variables but also provides constraints to represent the set of values in the source set and in the target set of the total surjective function. These constraints are very similar to the Range [12], global cardinality [100] and NValue constraints.

The weights need not be determined for every edge and node of the universe graph of the CSP but only for the edges and nodes present in the solution. In that case, the source set (domain) of the weight function is not determined beforehand. Map variables were described for integer values but can be extended to real values (float intervals) for the weights if necessary.

3.4 Kernel Graph Constraints

The kernel graph constraints of CP(Graph) define a set of three constraints which will enable us to express graph CSPs and constraints as conjunctions of these kernel constraints with set and integer constraints in section 3.5. These constraints link the graph variables with nodes, arcs, and sets of these.

- $Nodes(G, SN)$: This constraint holds if the value of the variable SN, a set of nodes, is equal to the set of nodes of the graph variable G.
- $Arcs(G, SA)$: This constraint holds if the value of the variable SA, a set of arcs, is equal to the set of arcs of the graph variable G.
- $ArcNode(A, N_1, N_2)$: This constraint holds if the value of the arc variable A is equal to the arc (N_1, N_2) where N_1 and N_2 are node variables.

The $ArcNode$ constraint is independent of the graph variables. All graphs in a CP(Graph) CSP are subgraphs of the universe graph of the CSP. Hence the mapping between values of node variables and arc variables encoded in the $ArcNode$ constraint is independent of the graph variables. It needs to be defined once for the universe graph of the CSP.

3.4.1 Consistency in CP(Graph)

We characterize the consistency level of the filtering algorithms using the same notions as in section 2.1.2.

When referring to optimal filtering for graph intervals, we use the same phrase as usually used for set intervals: *bounds consistency*. It should not be mixed up with bounds consistency for arithmetic constraints which was described in section 2.1.2.

Definition 3.12. Let $Sol(C, D)[X]$ denote the projection of $Sol(C, D)$ on variable X. For a graph or set variable X with domain $D(X) = [\underline{D}(X), \overline{D}(X)]$, we say c is *bound consistent* on X iff

$$\underline{D}(X) = \text{glb}\,(Sol(C, D)[X])\,,\ \overline{D}(X) = \text{lub}\,(Sol(C, D)[X])$$

Computing bounds consistency for a set or graph constraint amounts to compute the intersection (glb) and union (lub) of all solutions.

Note that the bounds of the interval must not belong to the set of solutions of the constraint. They do if the set of solutions to the constraint is closed under intersection and/or union. For instance, the bounds of a bounds consistent interval of a graph constrained to be symmetric (see section 4.3.1) are both symmetric. On the other hand, the bounds of a bounds consistent graph interval for an undirected graph constrained to be connected are not always connected themselves (see section 4.3.2).

Bounds consistency for set and graph intervals has an interesting relation with generalized arc consistency: If the graph interval is modeled with

Boolean variables, generalized arc consistency is equivalent to bounds consistency on the graph interval. The Boolean model is the following: for each element of the upper bound of the initial interval (each element of the set or each node and arc of the graph), one Boolean variable states the presence of this element in the final value of the set or graph variable.

3.4.2 Filtering Rules for the Kernel Constraints

We describe the filtering rules of the three kernel constraints and the adjacency constraint inherent to graphs. We first introduce some additional notation related to graph intervals and graph variables and constraints.

Notation

For the sets of nodes and arcs of the bounds of the domain of a graph, we use the following notation: if g is a fixed graph, $Nodes(g)$ denotes its set of nodes and $Arcs(g)$ its set of arcs. Hence the following equalities hold: $\underline{D}(G) = (Nodes(\underline{D}(G)), Arcs(\underline{D}(G)))$ and $\overline{D}(G) = (Nodes(\overline{D}(G)), Arcs(\overline{D}(G)))$. This notation refers both to the data structure used to store graph domains and to the value which is stored in it. When used on the left member of an assignment ($\leftarrow$), the expression denotes the data structure, and when used in the right member, the expression refers to the current value of the set. Hence, the expression

$$Nodes(\underline{D}(G)) \leftarrow Nodes(\underline{D}(G)) \cup \{n_1\}$$

Adds node n_1 to the lower bound graph of G. Remember that all domain updates must be contractant. Hence elements can be added to the lower bound or removed from the upper bound.

We also use a functional style for some constraints: We remove the last argument of a constraint and consider that the resulting syntactic expression denotes a variable (*e.g.* $Nodes(G)$ where G is a graph variable denotes SN in $Nodes(G, SN)$). We also write $(N_1, N_2) \in Arcs(G)$ as a simplified notation for $\exists A \in Arcs(G) \wedge ArcNode(A, N_1, N_2)$.

The Adjacency Constraint

Before we proceed with the kernel constraints, we give the propagation rules of the adjacency constraint of graphs which could be written $Arcs(G) \subseteq Nodes(G) \times Nodes(G)$. This constraint must be enforced by a filtering algorithm or set of simple constraints if the graph variable is encoded using set or integer variables. Its consistency properties are naturally derived from those of a subset constraint:

$$Arcs(\underline{D}(G)) \subseteq Nodes(\underline{D}(G)) \times Nodes(\underline{D}(G))$$

$$Arcs(\overline{D}(G)) \subseteq Nodes(\overline{D}(G)) \times Nodes(\overline{D}(G))$$

Those properties are translated into the following update operations which compute the new domains:

$$Arcs(\overline{D}(G)) \leftarrow Arcs(\overline{D}(G)) \cap Nodes(\overline{D}(G)) \times Nodes(\overline{D}(G))$$

$$Nodes(\underline{D}(G)) \leftarrow Nodes(\underline{D}(G)) \cup \{n | \exists v : \{(n,v),(v,n)\} \cap Arcs(\underline{D}(G)) \neq \emptyset\}$$

Once these two updates have been applied, the filtering is optimal. Remember these domain updates can lead to empty domains ($\underline{D}(G) \not\subseteq \overline{D}(G)$) in which case the CSP fails.

The Arcs Constraint

The $Arcs(G, SA)$ constraint is similar to the set equality constraint. The properties to maintain for the $Arcs(G, SA)$ constraint are the following:

$$\underline{D}(SA) = Arcs(\underline{D}(G))$$
$$\overline{D}(SA) = Arcs(\overline{D}(G))$$

We derive the filtering rules from these simple properties:

$$\begin{aligned}
\underline{D}(SA) &\leftarrow \underline{D}(SA) \cup Arcs(\underline{D}(G)) \\
Arcs(\underline{D}(G)) &\leftarrow \underline{D}(SA) \cup Arcs(\underline{D}(G)) \\
\overline{D}(SA) &\leftarrow \overline{D}(SA) \cap Arcs(\overline{D}(G)) \\
Arcs(\overline{D}(G)) &\leftarrow \overline{D}(SA) \cap Arcs(\overline{D}(G))
\end{aligned}$$

A single application of these rules leads to bound consistent domains: the properties are clearly respected after the application of these rules. It is also clear that the filtering rules do not discard solutions.

The Nodes Constraint

The $Nodes(G, SN)$ constraint is similar to the $Arcs(G, SA)$ constraint. The consistency properties are:

$$\underline{D}(SN) = Nodes(\underline{D}(G))$$
$$\overline{D}(SN) = Nodes(\overline{D}(G))$$

The filtering rules are:

$$\begin{aligned}
\underline{D}(SN) &\leftarrow \underline{D}(SN) \cup Nodes(\underline{D}(G)) \\
Nodes(\underline{D}(G)) &\leftarrow \underline{D}(SN) \cup Nodes(\underline{D}(G)) \\
\overline{D}(SN) &\leftarrow \overline{D}(SN) \cap Nodes(\overline{D}(G)) \\
Nodes(\overline{D}(G)) &\leftarrow \overline{D}(SN) \cap Nodes(\overline{D}(G))
\end{aligned}$$

We show that a single application of these rules achieves bound consistency. The rules clearly do not discard solutions and after an application of these rules, both bounds belong to the set of solutions (if the CSP is not failed). Hence, the union of all solutions is equal to the upper bound and its intersection is equal to the lower bound.

The ArcNode Constraint

The $ArcNode(A, N_1, N_2)$ constraint links an arc variable to two node variables. The properties for arc consistency are:

$$\begin{aligned}
&\forall (u, v) \in D(A) : u \in D(N_1) \wedge v \in D(N_2) \\
&\forall n \in D(N_1) : \exists (u, v) \in D(A) : u = n \\
&\forall n \in D(N_2) : \exists (u, v) \in D(A) : v = n
\end{aligned}$$

The update of the domains is straightforward: Apply these rules until a fixed point is reached

$$dom(A) \leftarrow dom(A) \cap (dom(N_1) \times dom(N_2))$$

$$\begin{aligned}
dom(N_1) &\leftarrow \{n_1 \in dom(N_1) | \exists (n_1, n_2) \in dom(A)\} \\
dom(N_2) &\leftarrow \{n_2 \in dom(N_2) | \exists (n_1, n_2) \in dom(A)\}
\end{aligned}$$

Proposition 3.13. *A single application of the above three rules in the given order leads to arc consistent domains.*

Proof. Figure 3.3 illustrates the different cases of pruning which can arise.

The rules explicitly compute the set of all values which belong to a solution to the constraint. Hence once a fixed point is attained, arc consistency is achieved.

Rule one propagates updates of the domains of N_1 and N_2 to the domain of A. On the other hand, rule 2 and 3 propagate the information the other way around, from A to the node variables. We prove the claim by showing that an additional application of the first rule never leads to additional

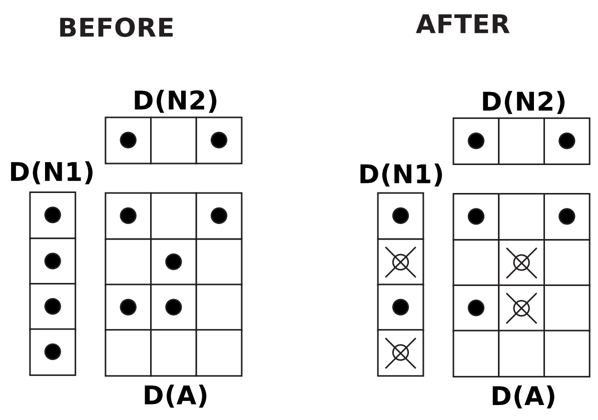


Figure 3.3: Example of arc consistent pruning for the ArcNode constraint

pruning. Such an additional application would further remove from $D(A)$ all arcs whose one end node has been removed from N_1 or N_2 by the application of rule 2 or 3. But these rules remove nodes only if they have no support arc in $D(A)$. So the arcs which would be removed are arcs which were not there in the first place. □

A simple algorithm can be derived from these rules. First store the values of the domain of the node variables in efficient hash data structures. Then create two similar empty structures for the endpoints of the arcs (called the projections data structures). Scan the arcs in the domain of the arc variable. For each arc, if its endpoints are part of the domains of the node variables, then add them to the projections, else, remove this arc from the arc domain. Finally, set the node domains to the computed projections.

Complexity of the filtering

The complexity of an algorithm implementing the listed filtering rules depends on several aspects. We consider here a computation of the new values from scratch as we assume an execution model similar to AC-3 as pointed out in section 2.1.

The complexity of a rule such as

$$\overline{D}(SA) \leftarrow \overline{D}(SA) \cap Arcs(\overline{D}(G))$$

depends on the cost of the set intersection ($\cap$) operation which in turns depends on the data structure used for the sets of arcs. This complexity can be assumed to be linear in the number of arcs of the upper bound graph :

$O(m)$ (when describing the complexity of a graph algorithm n denotes the number of nodes and m the number of arcs).

The complexity of optimal filtering for the constraints presented in this section are linear; m and n are respectively the size and order of the upper bound graph.

- $Adjacency(G)$: $O(m+n)$
- $Arcs(G, SA)$: $O(m)$
- $Nodes(G, SN)$: $O(n)$
- $ArcNodes(A, N_1, N_2)$: $O(\#D(A) + \#D(N_1) + \#D(N_2))$

3.5 Expressiveness of CP(Graph)

In this section we show how the kernel graph constraints described in section 3.4 can be used to model more complex constraints and problems. We present combinatorial graph problems as conjunction of high-level graph constraints on graph variables. These constraints are then translated to conjunctions of kernel graph constraints and basic set and integer constraints. We show that, in general, this decomposition leads to a large number of constraints for graph constraints dealing with transitive closure properties. This applies only to the constraint alone: we show that when put in conjunction with other constraints, such as in the Hamiltonian path problem, such properties can be expressed into smaller models.

3.5.1 Expressing Classical Graph CSPs and Optimization Problems

Numerous combinatorial graph problems can be expressed with CP(Graph). The graph constraints presented in other works [3, 110, 21] can be implemented in the CP(Graph) framework and used to solve these problems. To show the expressiveness and conciseness of CP(Graph), we present classical combinatorial graph problems as graph CSPs. In these expressions, $Subgraph(G, g)$ is used to declare a new graph domain variable G with initial upper bound g and empty lower bound. Similarly, $InducedSubgraph(G_1, G_2)$ holds if G_1 is an induced subgraph of G_2. Other graph properties are used as constraints : $Tree(G)$ holds if G is a tree, $Cycle(G)$ holds if G is a cycle, etc. . .

Constrained Path Problems

Constrained path problems are an important class of hard problems in graphs. We give a formulation for the Hamiltonian path, traveling salesman problem and resource constrained shortest path problems.

- Hamiltonian path in a graph g:

$$Subgraph(G, g) \wedge Cycle(G) \wedge Nodes(G) = Nodes(g)$$

- Traveling Salesman Problem (TSP) in graph g with weights w: minimize $Weight(G, w)$ subject to the same constraints.

- Resource constrained shortest path (with a single resource): Find the shortest weight constrained (maximum weight k) path of g with weights w, length function w_l, start node n_1, end node n_2: minimize $Weight(G, w_l)$ subject to

$$Subgraph(G, g) \wedge Path(G, n_1, n_2) \wedge Weight(G, w) \leq k$$

Steiner Tree Problems

Steiner tree problems are problems arising in network design. The problem is to find the smallest network which connects a pre-determined set of nodes. This problem arises for instance when a telecommunication company wants to build a network to inter-connect a given set of cities. With positive costs, the solution is a tree. A variation of this problem is the prize collecting Steiner tree problem where a prize is associated to each node (a cost is incurred for each non-connected node). The optimization criteria is the sum of the non-connected node prizes and the cost of connecting arcs.

- Steiner Tree Problem: g is the initial graph, the nodes to connect are sn and the arc weights are w_a: minimize $Weight(G, w_a)$ subject to

$$Subgraph(G, g) \wedge Connected(G) \wedge sn \subseteq Nodes(G)$$

- Prize Collecting Steiner Tree Problem: g is the initial graph, the arc weights and node prizes are w_a and w_n: minimize $Weight(G, w_a) + Weight(SN, w_n)$ subject to

$$Subgraph(G, g) \wedge Tree(G) \wedge SN = Nodes(g) \setminus Nodes(G)$$

Problems related to Steiner Trees can also be modeled with the MST and $WBST$ constraints defined in chapter 5. The MST constraint links two graph variables and holds if one is the minimum spanning tree of the other. The Steiner tree problem can be modeled in the following way using this constraint: Minimize $Weight(T, w_a)$ subject to

$$InducedSubgraph(G, g) \wedge MST(G, T, w_a) \wedge sn \subseteq Nodes(G)$$

Cut Problems

Cut problems arise for instance in circuit and VLSI design or in parallel computing [60]. These cuts are edge-cuts, sets of edges which once removed split a connected graph into several connected components (usually two). They are not the same cuts as in flows.

- Maximum cut: maximize $\#\left(Arcs(g) \setminus (Arcs(G_1) \cup Arcs(G_2))\right)$ subject to:

$$\begin{gathered} InducedSubgraph(G_1, g) \wedge InducedSubgraph(G_2, g) \wedge \\ Nodes(G_1) \cap Nodes(G_2) = \emptyset \wedge Nodes(G_1) \cup Nodes(G_2) = Nodes(g) \end{gathered}$$

- Finding an equicut of a graph g of even order: minimize $\#\left(Arcs(g) \setminus (Arcs(G_1) \cup Arcs(G_2))\right)$ subject to:

$$\begin{aligned} &Subgraph(G_1, g) \wedge Subgraph(G_2, g) \wedge \\ &\qquad Nodes(G_1) \cup Nodes(G_2) = Nodes(g) \wedge \\ &\qquad\qquad \#Nodes(G_1) = \#Nodes(G_2) = \frac{1}{2}\#Nodes(g) \end{aligned}$$

3.5.2 Expressing higher level constraints

We proceed by expressing graph property constraints which were used in the previous section to express graph CSPs. These constraints are decomposed into a combination of kernel graph constraints and basic set and finite domain constraints. We discuss bound consistency for these constraints and whether their decomposition allows the same level of consistency. We also discuss the cost of this decomposition in terms of the number of additional intermediate variables and constraints.

In this section we deal with simple constraints such as $Weight$, $Subgraph$ and $InNeighbors$. Constraints such as *Path* or *Connected* are more complex to decompose and are dealt with in the next section.

The Subgraph Constraint

The $Subgraph(G_1, G_2)$ constraint holds if $G_1 \subseteq G_2$, it can naturally be decomposed by using the definition of graph inclusion presented in section 3.1.

$$Subgraph(G_1, G_2) \equiv Nodes(G_1) \subseteq Nodes(G_2) \wedge Arcs(G_1) \subseteq Arcs(G_2)$$

This decomposition makes the $Subgraph$ constraint bound consistent since the propagator of subset ($\subseteq$) reduces domains to bound consistency too.

The Induced Subgraph Constraint

The $InducedSubgraph(G_1, G_2)$ constraint holds if G_1 is an induced subgraph of the graph G_2 i.e. the largest subgraph of G_2 containing the nodes of G_1 (largest with respect to graph inclusion).

$$\begin{aligned} InducedSubgraph(G_1, G_2) \equiv\ & Subgraph(G_1, G_2) \wedge \\ & SN = Nodes(G_2) \setminus Nodes(G_1) \wedge \\ \forall (n1, n2) \in Arcs(G_2) : (n_1 \in SN \vee n_2 \in SN) & \not\Leftrightarrow (n1, n2) \in Arcs(G_1) \end{aligned}$$

The Neighbors Constraints

The $InNeighbors(G, N, SN)$ constraint holds if SN is the set of all nodes of G from which an inward arc incident to N is present in G. If N is not in G then SN is empty. This constraint is helpful to express constraints based on local properties of the graph. For instance, it allows to express the $QuasiPath$ constraint (see below). This constraint is also a good complement to the set model of the Boolean adjacency matrix. In that model each set models the set of out-neighbors of each node. With this constraint, the in-neighbors can be modeled too. It can be expressed as the following network of constraints:

$$\begin{aligned} InNeighbors(G, N, SN) \equiv\ & SN \subseteq Nodes(G) \wedge \\ & (\#SN > 0 \Leftrightarrow N \in Nodes(G)) \wedge \\ & \forall n \in Nodes(G) : n \in SN \Leftrightarrow (n, N) \in Arcs(G) \end{aligned}$$

Note that this decomposition of $InNeighbors$ into basic and kernel constraints does not hinder consistency. It achieves an optimal filtering if the basic constraints achieve it too. The $(\#SN > 0 \Leftrightarrow N \in Nodes(G))$ constraint is redundant but needed to perform optimal filtering. The last implication constraint must be posted for all possible member of SN and for all possible in-neighbors of N. A simple solution is to post an implication for all n in $Nodes(\overline{D}(G))$ as SN is constrained to be part of that set. Similar expressions exist for inward arcs and the "out" versions of these constraints. $OutDegree$ and $InDegree$ are the cardinality of these sets.

The Weight Constraint

Many graph problems deal with weighted graphs. In weighted graphs, the weights can be put on the edges, on the nodes or even on both. The weight functions in CP(Graph) are used together with a graph variable to model a weighted graph. The $Weight(G, W, I)$ constraint allows to compute the total weight of a graph variable. That weight is then typically used in a branch and bound search procedure to solve these problems.

$Weight(G, W, I)$ holds if I is the total weight associated to the graph variable G according to the weight function W. It can be decomposed using the weight constraint for sets:

$$Weight(G, W, I) \equiv I = Weight(Nodes(G), W) + Weight(Arcs(G), W)$$

The $Weight(S, w, I)$ constraint (with a fixed w) is a reformulation of the weight constraint of Conjunto over weighted sets [54]. In that approach weighted set variables are set variables whose elements are pairs (value, weight). In the CP(Graph) design, weights are separated from the objects they operate on and they can as well be non-fixed. The $Weight$ constraint is presented in more detail in Chapter 4. In brief, optimal filtering (equivalent to generalized arc consistency [14] on the Boolean model) is NP-Hard by direct reduction from SUBSET-SUM. In section 4.5, we also propose a flow-based algorithm to perform a stronger propagation than the set decomposition in the general case of weights on both nodes and arcs (by taking the adjacency constraint into account).

3.5.3 Decomposing Transitive Closure Constraints

In this section, we describe constraints which deal with the transitive closure of the adjacency relation of graphs. Graph properties such as being connected or being a path, a cycle, a directed acyclic graph or a tree all deal with the transitive closure of the adjacency relation. We show that these transitive closure constraints require a large number of basic constraints in their decomposition; a bottleneck limiting the size of problems which can be handled by constraint solvers.

A constraint which illustrates the difficulty of decomposing transitive closure properties into basic constraints is the *Connected*(G) constraint which enforces that a graph variable G is a connected graph. We show that a decomposition of this constraint requires the computation of all terms of the matrix product presented below for the transitive closure (see also [20] for experiments with the connected subgraph problem). We also show that in certain circumstances this constraint can be decomposed in a much more compact way. This is illustrated by showing that a decomposition of the *Path* constraint into degree constraints and a connected constraint can be greatly simplified when the path is further constrained to be Hamiltonian.

Finally we conclude with the presentation of a transitive closure constraint which allows to concisely express these constraints. The Boolean model and the connected graph constraint are experimentally compared in chapter 7.

Preliminaries on transitive closure

Remember from section 2.2 that there is an arc (i, j) in the transitive closure g^+ of a graph g iff there is a path from i to j in g.

A graph $G = (N, E)$ of order n can be described as an Boolean adjacency matrix A of size $n \times n$ where $A_{i,j}$ is true iff $(i, j) \in E$. The Boolean matrix product is defined in a way similar to classical matrix product:

$$A^2_{i,j} = \bigvee_x A_{i,x} \wedge A_{x,i}$$

It means that there is an arc from i to j in A^2 iff there is at least one path of length 2 between i and j in A. We can then define the transitive closure of a graph, using Boolean matrix operations:

$$A^+ = \bigvee_{0 \leq x < n} A^x$$

Decomposing the Path Constraint

The *Path*(G, N_1, N_2) constraint holds if G is a simple path (a path with no repeated nodes) from N_1 to N_2. Optimal filtering for the path constraint is NP-Hard [110] (see also section 4.6).

This constraint can be decomposed using in two simpler constraints: the *QuasiPath*(G) constraints which consists in degree constraints enforcing some kind of flow conservation law (or Kirchoff's law) and the *Connected*(G) constraint which enforces that G is connected.

a. Enforcing degree constraints The *QuasiPath* predicate is presented by B. Courcelle in [26] (the G parameter is implicit in that work). This predicate turns into a constraint in CP(Graph). That constraint states the graph induced by SN in G *contains* a path from N_1 to N_2 by enforcing a local degree constraint for each node in SN.

$$QuasiPath(G, SN, N_1, N_2) \equiv N_1 \in SN \wedge N_2 \in SN \wedge$$
$$\forall n \in SN : \exists I_o = \#(OutNeighbors(G, n) \cap SN) \wedge I_o \leq 1 \wedge (n \neq N_2) \Rightarrow I_o = 1 \wedge$$
$$\forall n \in SN : \exists I_i = \#(InNeighbors(G, n) \cap SN) \wedge I_i \leq 1 \wedge (n \neq N_1) \Rightarrow I_i = 1$$

This constraint is not sufficient to express a $Path(N_1, N_2)$ constraint. A graph consisting of a path from N_1 to N_2 plus some additional disjoint cycles satisfies the *QuasiPath* constraint (see figure 3.4). To implement the path constraint one has to force the graph to be connected or acyclic. We focus on connectedness here but it can be shown that acyclicity is as hard to decompose as connectedness.

b. Enforcing connectedness In CP(Graph) enforcing connectedness can be done by posting a $Connected(G)$ constraint. This constraint can be decomposed using the undirected transitive closure in which the undirected matrix product TC_u is used: Let $n = \#Nodes(\overline{D}(G))$ denote the order of the upper bound of G,

$$\begin{aligned}
Connected(G) &\equiv \forall u, v \in Nodes(G) : (u, v) \in TC_u(G) \\
TC_u(G) &= \bigvee_{1 \leq l \leq n} U^l \\
U^l(i, j) &= \bigvee_{1 \leq k \leq n} U^{l-1}(i, k) \wedge (U(k, j) \vee U(j, k)) \\
U(i, j) = U^1(i, j) &= (i, j) \in Arcs(G) \vee (j, i) \in Arcs(G)
\end{aligned}$$

This general decomposition of the $Connected(G)$ constraint uses $O(n^3)$ additional constraints and $O(n^4)$ variables. This indicates that a dedicated filtering algorithm could perform better than this constraint decomposition; This is confirmed by an experiment in chapter 7.

In the next section we show that while this general decomposition of the *Path* constraint uses a large number of constraints, decompositions for more constrained problems such as the Hamiltonian path can be smaller.

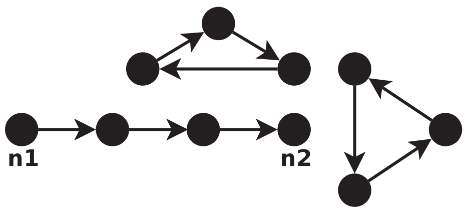


Figure 3.4: A Solution to the $QuasiPath(G, n_1, n_2)$ constraint. The graph consists in a path from n_1 to n_2 and two additional circuits.

The Knight's Tour Problem

This problem consists in finding a path visiting each square of a chess board exactly once by using the moves of the knight. It is a instance of the Hamiltonian path problem in a special kind of graph: the Knight's graph. A finite domain model using the `circuit` constraint is presented in [120]. A model in which the circuit constraint is decomposed into basic integer constraints is provided as an example within Mozart and Gecode; we describe it below.

A simple CP(Graph) model for the Knight's Tour follows: Let $KG_{8,8}$ be the Knight's graph for a square board of 8×8 squares. This graph consists of a node for each square of the chess board and arcs connecting each pair of squares which can be joined by one knight's move. Let $n_{0,0}$ be a corner square and $n_{1,2}$ be a square one knight's move away from this corner, the problem is:

$$Subgraph(G, KG_{8,8}) \wedge Nodes(KG_{8,8}) = Nodes(G) \wedge Path(G, n_{0,0}, n_{1,2})$$

This problem model uses the path constraint and we show that it is possible to come up with a much more compact model for this problem than the combination of the $QuasiPath(G)$ and $Connected(G)$ constraint presented above. We show that the $Connected(G)$ constraint can be enforced by a much more compact model by capitalizing on the properties of the problem.

In the classical model for the Knight's tour problem which we present here, the squares of the chess board are numbered from 0 to 63. For each square the model contains 3 integer variables: The first one ($jump$) gives the index of the square in the path sequence, the second one ($succ$) gives the number of the next square in the path and the last one gives the number of the preceding square (*pred*).

(1) $\forall n \in [0, 63] : pred_n \in Neigh(n, KG_{8,8}) \wedge succ_n \in Neigh(n, KG_{8,8}) \wedge$
$jump_n \in [0, 63] \wedge pred_n \neq succ_n$

(2) $\forall (u, v) \in Arcs(KG_{8,8}) : succ_u = v \Leftrightarrow pred_v = u \Leftrightarrow jump_v = jump_u + 1$

(3) $\forall n, n' \in [0, 63] : jump_n \neq jump_{n'}$

(4) $jump_0 = 0 \wedge jump_{17} = 1$

$Neigh(n, KG_{8,8})$ denotes the set of squares which can be reached by one knight's move from n. In 2D coordinates $n = (i, j)$ and the neighbors are $(i-2, j-1), (i-2, j+1), (i-1, j-2), (i-1, j+2), (i+1, j-2), (i+1, j+2), (i+2, j-1), (i+2, j+1)$. Using the raster order encoding of the integer square positions for the $8, 8$ board, the neighbors of n are $n-17, n-15, n-10, n-6, n+6, n+10, n+15, n+17$ as long as these values are in $[0, 63]$. The variables *succ* encode the successor of each node in the Hamiltonian path. *pred* is constrained to model the predecessor of each node on line (2). The *jump* variables encode the position of the node in the path sequence, the source is 0 and its successor is 1 (line (4)). The alldiff on line (3) is redundant but provides global pruning of the variables.

a. Enforcing degree constraints Modeling the previous and next square of each square of the board with an integer domain enforces the $QuasiPath(G, SN, N, n_2)$ constraint: Each square has exactly one in-neighbor and one out-neighbor.

b. Enforcing connectivity The connectivity property is enforced by modeling the index of each square in the path with an integer and forcing neighbor squares in the path to have successive indexes. This works fine as each square must be part of the path. Adding a global alldifferent constraint on the index variables provides the suitable pruning for solving the Knight's tour problem in a reasonable time. This way of enforcing connectivity is much more compact than by using the transitive closure decomposition presented above. However it is not generic as it relies on additional properties of the CSP (the fact that each node is visited only once and they are in sequence).

The transitive closure global constraint

A conclusion of this decomposition exercise is that computing transitive closure in a declarative way is expensive in terms of intermediate variables. We have indeed shown that in the general case, enforcing connectivity with the transitive closure decomposition uses $O(n^3)$ N-ary disjunction constraints and $O(n^4)$ Boolean variables. A exception is the Hamiltonian path problem (an instance called the Knight's tour was presented) for which we presented a model using only $O(n)$ integers and $O(n.d)$ Booleans where d is the average degree of nodes in the graph (d can be constant as in the Knight's tour problem or up to n in the general case).

This explosion of the number of variables and constraints tends to indicate that using global constraints for properties related to transitive closure could be beneficial as their propagation can typically takes only $O(n^2)$ time (linear in the number of edges) and $O(n)$ space (when based on a single application of DFS [114]). Such constraints will be described in chapter 4 and an experimental comparison of the integer approach and the graph plus global constraint approach for connectedness is presented in chapter 7.

We conclude this section by describing a global transitive closure constraint which allows to express many graph properties in a very compact way. This constraint is described in more depth in section 4.7.

$TC(G, G^+)$ holds if G^+ is the transitive closure of graph G.

Graph properties such as being connected or being a path, a cycle, a directed acyclic graph or a tree which were introduced in the beginning of this section can be easily expressed using TC:

- $Connected(G) \equiv \forall n_1 \neq n_2 \in Nodes(G) : (n_1, n_2) \in TC(G)$
- $Cycle(G) \equiv Connected(G) \wedge \forall n \in Nodes(G) : OutDegree(n) = InDegree(n) = 1$
- $DAG(G) \equiv \forall (n_1, n_2) \in Arcs(G) : (n_2, n_1) \notin TC(G)$
- $Tree(G) \equiv DAG(G) \wedge Connected(G)$

This constraint is a part of the *domReachability* constraint introduced in [92]. Bounds consistency for the TC constraint is shown to be NP-Hard in section 4.7.

3.6 Graph Intervals and Finite Domain Filtering Algorithms

In this section, we show the relation between graph intervals and various constraints over integers. We first describe graph constraints which have been defined over the single successor integer model and show that even alldiff can be considered a graph constraint. Then we show that the filtering algorithm based on graphs of many integer constraints can be interpreted as filtering algorithms for graph constraints over a graph interval. Finally, we present a graph-based generic filtering algorithm [5] and show that it can be viewed as a graph interval CSP.

3.6.1 Finite Domain Graph Constraints

Many graph constraints have been designed since the seminal ALICE system [76]. Two graph constraints can be found in ALICE: *Hamiltonian path* and *circuit*. A thesis [19] (in french) was dedicated to the `cycle` constraint[1]. The catalog of global constraints [3] is a list of complex constraints. Among the “graph partitioning constraints” listed in [3] on page 86, we describe three such constraints: `cycle`, `tree`, and `symmetric alldifferent`. They all use the single successor model for graphs, a vector of integer variables in which an integer variable at index i with value j models the inclusion of arc (i, j) in the graph (see section 3.2).

The `cycle` constraints holds if the sequence of integers models a set of circuits. In addition to the set of integers modeling the graph, the `cycle` constraint has a parameter counting the number of cycles in the graph. In [19], more than 10 additional parameters are presented for this constraint in order to model various aspects of a wide range of vehicle routing problems.

Apart from these additional parameters, we show that the core of the `cycle` constraint is the `alldiff` constraint which holds if its arguments are all pairwise distinct. Obviously, any assignment of values to variables is interpreted as a collection of trees and circuits covering the fixed set of nodes of the graph (see figure 3.2 for an example). Ensuring that any two arcs point to distinct nodes results in avoiding trees in this model which amounts to requiring that the assignments model circuit covers. This shows that the `alldiff` constraint can be interpreted as a graph constraint (`cycle`). The next section gives another graph interval interpretation of the filtering algorithm of the `alldiff` constraint.

The `tree` constraint covers the graph with trees directed to the root; they could not be directed the other way around as this type of graph is not representable with the single successor model. As the cycle constraint, it has also a parameter counting the number of trees. The core of the filtering is subtour elimination which avoids the creation of cycles (see section 4.4.1

[1]However the algorithms are kept confidential until 2010

for a detailed exposition).

The `symmetric alldifferent` constraint enforces that the directed graph represented by its arguments in the single successor model is symmetric (closed under arc reversal) and that the undirected graph obtained by merging reverse arcs is a matching (each node is connected to exactly one other node). It is a particular case of the `cycle` constraint when the number of cycles is exactly half of the number of nodes.

This shows that the frontier between graph constraints and finite domain constraints is blurry. First because graph constraints have first been defined in the finite domain. Second, because some constraints which have an interpretation outside graphs turn out to also have an interpretation as a graph property. This second argument is further developed when investigating the filtering algorithm behind the `alldiff` constraint and other global constraints below.

3.6.2 Graphs in Filtering Algorithms

Graphs are used in filtering algorithms for global constraints. Some consistency properties of these constraints are easily defined using graphs and graph algorithms enable efficient pruning. For the `alldiff(X)` constraint [99] which holds if all its integer variables `X` have different values, a solution corresponds to a maximum matching in a bipartite variable-value graph (if all nodes corresponding to variables are covered). A necessary and sufficient condition for an arc to belong to any maximum matching is based on alternating paths and cycles and arc consistent domains can be computed in $O(\sqrt{n}m)$ where m is the number of arcs and n the number of nodes of the graph. See [122] for a survey on this constraint.

If we interpret the sequence X of variables as modeling a graph G, the variable-value graph, the filtering algorithm of the `alldiff` constraint performs an optimal pruning for the graph constraint $MaximumMatching(G)$ provided all the nodes corresponding to variables are included in $\underline{D}(G)$. This algorithm is very easy to extend to the general case or any variant such as $Matching(G)$ or $PerfectMatching(G)$.

Many variants of this constraint use similar results: The symmetric all different constraint [101] deals with general graph matching (*i.e.* not only bipartite matching). Bound consistency for the *Same* constraint [9] uses parity matching. And the weighted partial all different constraint [117] uses maximum weight matching. All those algorithms can be re-interpreted as filtering algorithms for graph constraints.

The global cardinality constraint [100] (abbreviated `GCC`) is a generalization of `alldiff` enforcing bounds on the number of occurrences of values in a set of variables. Its filtering algorithm is based on a flow computation. It uses the same consistency technique as `alldiff`: modeling the assignment as an admissible flow and computing all arcs which can and cannot belong to such a flow. This constraint can be reinterpreted as a graph constraint which holds if the variable-value graph models an admissible flow ($AdmissibleFlow(G)$). The filtering algorithm of `GCC` performs optimal filtering for this constraint.

Other results have been based on flow theory, like global cardinality with costs [102] or arc consistency for the *Same* constraint [9] or optimization constraints based on `GCC` and `alldiff` and usable as soft constraints [121].

This shows that many global constraints either presented with a graph semantic or simply with a filtering algorithm based on a graph property can be converted to a graph constraint over a graph interval. In the next section, this affirmation is extended to almost all global constraints.

3.6.3 Networks of Constraints and Graph-Based Filtering for Global Constraints

As constraints are links between variables, a constraint satisfaction problem can be directly modeled as a graph or hypergraph. This graph model has been used in various applications. The graph structure of constraint satisfaction problems has been used since the beginning of studies on consistency [80] and consistency names like node, arc, path or hyper-arc consistencies come from this graph model. Properties of constraint graphs have been used to study the hardness of CSPs [112]. These graphs have also been used to solve systems of geometric constraints [45]. Different types of constraint graphs (variable graphs, propagator graphs, ...) were used in [82] to debug constraint programs.

In the global constraint catalog, this constraint network model is used to describe the semantics of many global constraints. The global constraint is decomposed in a graph which nodes are the integer variables of the global constraints and which arcs model simple binary constraints holding on these variables.

Recent work [5] has been directed towards the definition of a filtering scheme based on this constraint network representation of global constraints (on integers and sets). This algorithm uses the concepts of initial, intermediate and final graph. The initial graph is built according to the semantics of the constraint. The intermediate graph consists in labeling the arcs of the initial graph as true, undetermined and false arcs depending on the entailment of the simple binary constraints. The final graph contains all true arcs when all variables are fixed.

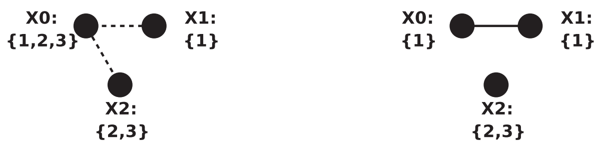
These graphs exactly correspond to a graph interval: The initial graph defines the upper bound of the graph interval and the final graph corresponds to a singleton interval, the true arcs constitute the lower bound and the upper bound is all arcs but the false ones.

Other parameters of the global constraints which are not part of the graph model are constrained by bounds of graph characteristics. These graph characteristics which can be interpreted as constraints of the form $C(G, I)$ where G is a graph interval and I is a property such as the number of nodes, edges, strongly or weakly connected components, sources or sinks. Algorithms which compute these bounds given an intermediate graph (a graph interval) were given in [4]. These algorithms do a global reasoning by taking into account the constraint of forbidden isolated nodes. They are also compared with formulas for the general case where isolated nodes are allowed.

As an example, we take the constraint `NValue(X,I)`, where `X` is an array of integer variables and `I` an integer variable. It holds if `I` is the number of different values in `X`. As illustrated in figure 3.5 and example 3.14, each variable x_i of `X` is mapped to a node n_i of the graph domain G. The set of nodes is fixed, *i.e.* all nodes are part of the lower bound.

Each arc is linked to the entailment of a basic constraint: $(n_i, n_j) \in Arcs(G) \Leftrightarrow x_i = x_j$. As soon as the domains of the variables x_i and x_j are disjoint, we know that (n_i, n_j) must not be part of G so it is removed from $\overline{D}(G)$. As soon as the variables x_i and x_j are assigned to the same value we know that this arc must be in the graph so it is included in $\underline{D}(G)$. The variable `I` is constrained to be the number of connected components in G. Hence a constraint $NCC(G, I)$ is posted on these two variables and filtering for this constraint includes or removes arcs from G and reduces the domain of I.

Example 3.14. We consider the constraint $NValue(X_0, X_1, X_2, C)$ with domains $X_0 \in \{1, 2, 3\}$, $X_1 \in \{1\}$, $X_2 \in \{2, 3\}$. The graph representation for this constraint is based on the constraint $X_i = X_j$. An undirected graph interval G is built by creating one node for each integer variable and considering the presence of an arc $\{i, j\}$ in G depending on the entailment of the constraint $X_i = X_j$. If the constraint is entailed, then the arc is present (*i.e.* in the lower bound) if it is disentailed (its negation is entailed), then the arc is not present (*i.e.* not in the upper bound). Figure 3.5 presents the graph for the domains which we considered above. The arc $\{1, 2\}$ is not in



(a) First graph interval

(b) After assigning X_0 to 1

Figure 3.5: Graph-based model for filtering the NValue constraint. See example 3.14 for a detailed comment.

the upper bound as the domains are disjoint and the variables cannot be made equal (the constraint $X_1 = X_2$ is disentailed). The two other arcs are in the upper bound but not in the lower bound as the variables are not equal yet.

The value of C is determined by the constraint $NCC(G, C)$ which holds if C is the number of connected components of G. Clearly, the domain of C is $[1, 3]$ as there are graphs with 1, 2 and 3 connected components in the interval. For example, if we set the value of X_0 to 1, the graph G gets fixed to the value depicted in figure 3.5(b). There are 2 connected components in this graph, hence $C = 2$. While the domains of the integer variables are not all fixed ($X_0 = 1$, $X_1 = 1$, $X_2 \in \{2, 3\}$), the graph G is fixed and the constraint is entailed: whatever the value assigned to X_2, there are 2 different values among the variables X_i.

In conclusion, integer constraints can be interpreted as graph constraints for graph intervals. Graph intervals can also be used to interpret important filtering algorithms such as those of `alldiff` or `GCC` and their many derivatives. Finally, they can be used in a generic filtering algorithm for almost all global constraints presented in the global constraint catalog [3].

3.7 Summary

In this chapter we first gave a theoretical background for using graph intervals as a graph domain abstraction. We presented three kernel graph constraints that allow us to express graph CSPs using graph, node, arc and set variables. These kernel constraints constitute the link with the set and integer domains and allow the decomposition of higher level graph constraints into CSPs containing kernel graph constraints and set and integer constraints.

We showed how these higher level constraints allow to formulate very concise models for classical graph problems arising for instance in Operations Research. We also showed that while this decomposition is feasible for constraints related to transitive closure, it however leads to very large models which makes these constraints amenable to an implementation as global graph constraints. In the next chapter we show that such constraints can often be filtered optimally in $O(m)$ time and $O(n)$ space using simple DFS algorithms [114].

Finally, we showed that the usage of graph intervals and constraints extend beyond the specification of CSPs using graph variables. They can be used as a formal framework to describe many filtering algorithms and could be used in a generic filtering algorithm for global constraints.

Chapter 4

Global Graph Constraints

4.1 Introduction

In the previous chapter of this thesis, we showed that kernel graph constraints allow to express CP(Graph) CSPs. We however pointed out that the decomposition of graph properties such as connectedness can be cumbersome and leads to huge CSPs. In this chapter we design a set of graph constraints which allow to model basic graph properties or relations in a CP(Graph) CSP.

Constraints for graph properties are not new: back in 1979, Laurière already used the *Hamiltonian circuit* and *spanning tree* graph properties in ALICE [76]. More recently, the *cycle* [6, 19], the *path* [110, 109, 49, 21, 36] or the *tree* [7] constraints have been studied. As presented in section 3.6.1, other constraints such as *alldiff* and its derivatives use a graph as the underlying model for their filtering algorithm which can be interpreted as the filtering for a graph property on a graph interval. For the *alldiff* constraint this graph property is *bipartite maximum matching*.

Previous work on filtering algorithms either for graph constraints or based on graph properties can be reused and extended in the context of CP(Graph) as most of them fit in the graph interval domain abstraction.

In this chapter we present graph constraints which cover many aspects of graph theory: reachability and connectedness, structural properties like bipartiteness, acyclicity and properties which combine those like path or tree. For each constraint, we provide a filtering algorithm which performs optimal filtering (bounds consistency) or prove that it is NP-complete. We examine filtering complexity and detection of fixed point and subsumption.

Section 4.2 presents constraints for graph relations such as subgraph, induced subgraph and complement graph. Section 4.3 describes constraints to support undirected graphs in CP(Graph) and constraints for connectedness properties of undirected and directed graphs. Section 4.5 presents the graph weight constraint. Section 4.6 covers the filtering algorithm for the different path constraints. Finally, section 4.7 describes the transitive closure constraint. Before the presentation of graph constraints, we look at a theorem which we use to prove NP-hardness of bound consistency (optimal filtering) for some of these constraints.

4.1.1 Complexity of Consistency

Recall from Section 3.4.1 that computing bound consistent domains for a graph relation implies that for each variable we look at the union and intersection of all values it can take in the relation; All elements (nodes and arcs) in the upper bound must belong to at least one solution and the elements which belong to all solutions must be placed in the lower bound. For each element in the upper bound but not in the lower bound, there is both one solution which contains this element and one solution which does not contain it. Bound consistency for a graph interval amounts a fixed point computation of optimal filtering functions for this interval.

The following theorem 4.1 from [72] is used to prove NP-hardness of bound consistency for graph variables. It implies that if it is NP-hard to determine whether there exists a solution to a constraint defined on graph variables (and possibly other variables) then it is NP-hard to filter the domains of the variables to bound consistency.

Theorem 4.1. *Let $C(G_1, \ldots, G_k)$ be a constraint defined on graph variables $G_1, \ldots, G_k$ If there is a polynomial-time algorithm that narrows the domains of all variables to bound consistency then there is a polynomial-time algorithm that finds a single solution to C.*

Proof. We show that a bound consistent filtering algorithm allows to find a solution to the constraint with no backtrack. When the domains are bound consistent, pick a value (arc or node) of the upper bound of a variable and include it in the lower bound. As there is at least a solution assuming this value, including it will not fail. Apply the filtering algorithm again and loop until all variables are instantiated. The number of iterations is upper bounded by the sum of the sizes of the upper bounds of variables. □

4.2 Graph Binary Relations

In this section we describe three constraints: The *Subgraph*(G_1, G_2) constraint holds if G_1 is a subgraph of G_2. The $Complement(G_1, G_2)$ constraint holds if G_1 and G_2 have the same node set and each arc of the complete graph built over these nodes belongs to exactly one of the two graphs. The $InducedSubgraph(G_1, G_2)$ constraint holds if G_1 is an induced subgraph of G_2: For each pair of nodes of G_1 if the arc is present in G_2 it must be present in G_1.

We give a complete description of the subgraph constraint as an example of consistency proof for this kind of simple relation constraints. The other two constraints are handled more informally.

4.2.1 The *Subgraph*(G_1, G_2) constraint

The *Subgraph*(G_1, G_2) constraint holds if G_1 is a subgraph of G_2.

This relation defines the partial order among graphs and gives a lattice structure to graph intervals. It is used in practice to model several subgraphs co-existing in a larger graph: The need to model multiple paths in a single graph lead to the choice of a list of Boolean variables instead of the single successor model in [21].

Theorem 4.2. *The Subgraph*(G_1, G_2) *constraint is bound consistent iff* $\overline{D}(G_1) \subseteq \overline{D}(G_2)$ *and* $\underline{D}(G_1) \subseteq \underline{D}(G_2)$.

Proof. We first show that these properties are necessary for bound consistency. Let $e \in \overline{D}(G_1) \setminus \overline{D}(G_2)$, e does not belong to any assignment of G_2 hence it cannot belong to one of its subgraphs and does not belong to a solution. Similarly, let $e \in \underline{D}(G_1)$, for any value of G_1, $e \in G_1$ hence in any solution e must belong to G_2 and it should be included in its lower bound. We now show that the properties are sufficient. We assume they hold and show we have bound consistency. We consider two solutions to the constraint. The first solution ($G_1 = \underline{D}(G_1)$ and $G_2 = \underline{D}(G_2)$) is a witness that every element not in the lower bound can be excluded from a solution. The second one proves that each element of each upper bound belongs to at least a solution. □

A filtering algorithm therefore needs to perform the following steps:

1. If $\underline{D}(G_1)$ is not a subgraph of $\overline{D}(G_2)$, the constraint has no solution.
2. If $\overline{D}(G_1)$ is a subgraph of $\underline{D}(G_2)$, the constraint is entailed.

3. For each node or edge in $\underline{D}(G_1) \setminus \underline{D}(G_2)$, include it in $\underline{D}(G_2)$.
4. For each node or edge in $\overline{D}(G_1) \setminus \overline{D}(G_2)$, remove it from $\overline{D}(G_1)$.

The running time of this algorithm is linear in the sizes of the bounds of G_1 and G_2 as each of these steps takes linear time.

4.2.2 The $Complement(G_1, G_2)$ constraint

The $Complement(G_1, G_2)$ constraint holds if G_1 and G_2 have the same node set and each arc of the complete graph built over these nodes belongs to exactly one of the two graphs. That is, if an arc is present in G_1 it cannot be present in G_2 and symmetrically. See figure 4.1 for a illustration of the following. The node sets must be equal, hence the node sets of the bounds must be equal too. The node sets of both upper bounds are set to their intersection and the node sets of both lower bounds are set to their union.

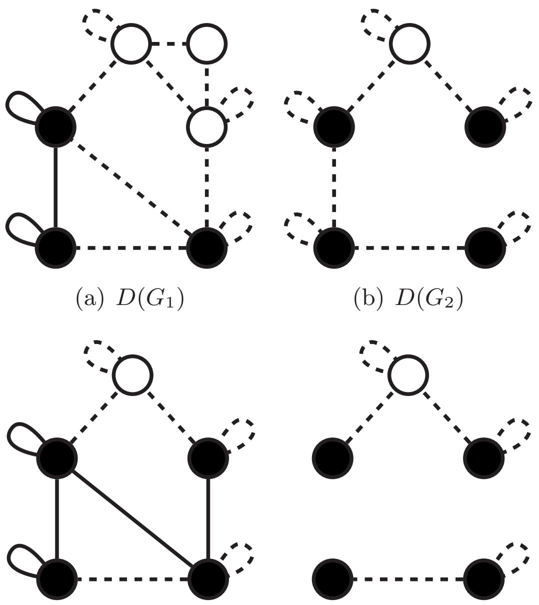
The constraint is symmetrical, we present the pruning rules for G_1 based on G_2 but those also apply symmetrically to filter G_2 according to G_1. As soon as an arc is present in one graph it cannot be present in the other graph. Hence we must remove from $\overline{D}(G_1)$ all arcs of $\underline{D}(G_2)$.

The lower bound of G_1 must be filtered too: all arcs which will never be part of G_2 and whose endnodes are already part of the node set $Nodes(\underline{D}(G_1))$ must belong to G_1: The complement of the subgraph of $\overline{D}(G_2)$ induced by $Nodes(\underline{D}(G_2))$ must be added to $\underline{D}(G_1)$.

To prove optimality there remains to show that each arc e in $\overline{D}(G_1) \setminus \underline{D}(G_1)$ can either belong or not belong to G_1 in a solution. It suffices to see that if the endnodes of e are in $\underline{D}(G_1)$ then the arc is also in $\overline{D}(G_2)$. We can choose a solution by including the arc in one of the graphs and excluding it from the other. The complexity of this filtering algorithm is linear as all steps are linear.

4.2.3 The $InducedSubgraph(G_1, G_2)$ constraint

The $InducedSubgraph(G_1, G_2)$ constraint holds if G_1 is an induced subgraph of G_2: For all pairs of nodes of G_1 if an arc is present in G_2 it must be present in G_1. In other words, if the endnodes of an arc of G_2 are present in G_1, then the arc is present in G_1 too. The consistency properties are the following (see also figure 4.2). G_1 must be a subgraph of G_2, hence the bounds are subject to the same consistency properties as in the *Subgraph* constraint. There remains to deal with the “induced” part of this constraint. It amounts to enforcing graph equality on the subgraphs induced by the mandatory



(a) $D(G_1)$ (b) $D(G_2)$

(c) $D(G_1)'$ (d) $D(G_2)'$

Figure 4.1: *Complement*(G_1, G_2): Original $D(G_1)$ and $D(G_2)$ and updated domains $D(G_1)'$ and $D(G_2)'$ after optimal filtering. Bold nodes and arcs denote elements of the lower bound. Dotted arcs and empty nodes denote other elements in the upper bound.

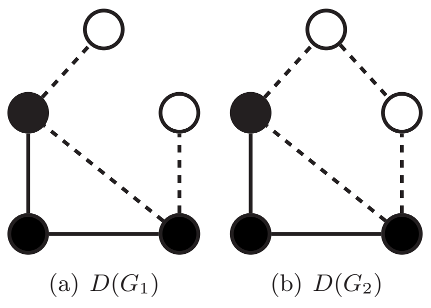


Figure 4.2: Bound consistent domains $D(G_1)$ and $D(G_2)$ for the $InducedSubgraph(G_1, G_2)$ constraint

nodes $Nodes(\underline{D}(G_1)) = Nodes(\underline{D}(G_2))$. This constraint can be written $\forall u, v : (u,v) \in Arcs(G_2) \wedge \{u,v\} \subseteq Nodes(G_1) \Leftrightarrow (u,v) \in Arcs(G_1)$. Note that the $\Leftarrow$ reverse implication is already present in the *Subgraph* relation. When faced with an implication, two propagation rules arise. If the left hand side of the implication is true, so must be the right side. If the right hand side of the implication is false, then so must be the left one.

Hence for each arc incident only to mandatory nodes, if the arc is present in one graph then it is present in the other too and if it is absent from one it must be absent from the other.

To prove bound consistency, we build solutions. The first consists in taking the lower bound of both graphs. It shows that each arc not in the lower bound can be excluded from at least one solution. For each arc e of $\overline{D}(G_1)$, we build a solution including e: $g_1 = \underline{D}(G_1) \cup \{e\}$ and $g_2 = \underline{D}(G_2) \cup \{e\}$. Finally, we build a solution with each arc e of $\overline{D}(G_2)$: add the arc to $\underline{D}(G_2)$ to build g_2. If both endnodes of e belong to $Nodes(\underline{D}(G_1))$, then take $g_1 = \underline{D}(G_1) \cup \{e\}$ otherwise simply take $g_1 = \underline{D}(G_1)$.

All steps of the algorithm are linear in the size of the bounds so the complete filtering algorithm is linear.

4.3 Connectedness Constraints

The *Connected*(G) constraint holds if G is an undirected connected graph. This constraint is important as it illustrates propagation techniques which are reused in the *Tree* and *Path* constraints in this chapter and in the spanning tree constraints presented in the next chapter.

We first show how we support undirected graphs in CP(Graph). Then an optimal filtering algorithm is presented for the connected constraint and two connectedness constraints for directed graphs are presented.

4.3.1 Constraints to Support Undirected Graphs

.

We provide two constraints to deal with undirected graphs: the *Symmetric*(G) constraint which holds if the adjacency relation of G is symmetric and the *Undirected*(G, G_u) constraint which holds if G_u is the symmetric version of the undirected graph obtained from G by forgetting the direction of its arcs.

The Symmetric Constraint

The *Symmetric*(G) constraints holds if the adjacency relation is symmetric:

$$\forall (u, v) \in Arcs(G) : (v, u) \in Arcs(G)$$

The pruning rules are the following : whenever $(u, v) \in \underline{D}(G)$, include the opposite arc : $(v, u) \in \underline{D}(G)$. When a single arc is present in $\overline{D}(G)$: $(u, v) \in \overline{D}(G) \wedge (v, u) \notin \overline{D}(G)$, if it is present in $\underline{D}(G)$ the constraint fails, otherwise remove (u, v). This constraint is entailed only when the graph is fixed, when its domain is a singleton. These pruning rules compute bound consistent domains; they obviously do not remove solutions to the constraint and once applied, both bounds are solutions.

The filtering algorithm for this constraint has linear $O(n+m)$ complexity: A traversal of the arcs of the upper bound collects the isolated arcs and the reverse arc of arcs in the lower bound. Then these arcs are respectively removed from the upper bound and included in the lower bound.

The Undirected Binary Constraint

The *Undirected*(G, G_u) constraint holds if G_u is the undirected (symmetric) graph obtained by ignoring the direction of the arcs of G:

$$\forall (u, v) \in Arcs(G) : (u, v) \in Arcs(G_u) \wedge (v, u) \in Arcs(G_u)$$

This constraint allows to apply an undirected constraint to an undirected version of a directed graph. This allows for instance the definition of a weakly connected directed graph as a directed graph whose undirected version is connected.

To prune G_u, include in $\underline{D}(G_u)$ every arc of $\underline{D}(G)$ and its reverse arc. Remove from $\overline{D}(G_u)$ every arc which is not in $\overline{D}(G)$ nor its reverse arc. To prune G, remove from $\overline{D}(G)$ all arcs which are not present in $\overline{D}(G_u)$. Include arcs in $\underline{D}(G)$ if there are corresponding arcs in $\underline{D}(G_u)$. For each arc (u, v) in $\underline{D}(G_u)$, if there is only one corresponding arc in $\overline{D}(G)$, it is included, if there are two such arcs, nothing is done and if there is no such arc, the constraint fails.

Obviously this pruning does bound consistency. It does not lose solutions and prunes the domains up to a point where both bounds can easily be extended to belong to a solution, like for the *Subgraph* and *Symmetric* constraints.

The complexity is again linear in the size of the largest bound.

4.3.2 The Undirected Connected Constraint

We presented a filtering algorithm for this constraint in [35]. The filtering algorithm is based on the following properties:

Lemma 4.3. *If two nodes of $\underline{D}(G)$ belong to different connected components of $\overline{D}(G)$, then there is no connected graph in $D(G)$.*

Proof. Connected components are maximal connected subgraphs. As each graph of $D(G)$ contains at least a node in each connected component, it cannot be connected. □

Proposition 4.4. *If $\underline{D}(G)$ is not empty, the union of all connected graphs in $D(G)$ (if there are) is a connected graph.*

Proof. Assume $\underline{D}(G)$ contains at least one node n. If $\overline{D}(G)$ is connected, it is the union of all connected graphs as it belongs to that set. If $\overline{D}(G)$ is not connected, either we fall in the case of lemma 4.3 and there is no solution, or all connected graphs are subgraphs of one connected component of $\overline{D}(G)$. In that case, that component is the union of all connected subgraphs. □

Proposition 4.5. *The intersection of all connected graphs in $D(G)$ is $\underline{D}(G)$ plus all bridges and cutnodes on a path between two nodes of $\underline{D}(G)$.*

Proof. We first show that the intersection contains these nodes. Then we show that it contains no other nodes.

1. By definition, we disconnect $\overline{D}(G)$ by removing any cutnode or bridge so there is no connected graph which does not contain them.

2. We now show that every other element of $\overline{D}(G)\setminus\underline{D}(G)$ is not present in at least one solution. By proposition 4.4 and without loss of generality, we assume that $\overline{D}(G)$ is connected. Consider the graph $\overline{D}(G) \setminus \{e\}$, it is a solution if and only if it is connected and in $D(G)$. If it is not, then either it is not connected which makes e a bridge or cutnode, or it is not in $D(G)$ which means e belongs to $\underline{D}(G)$.

□

The two propositions can directly be applied to prune the bounds of the domain: If $\underline{D}(G) \neq \emptyset$ and $\overline{D}(G)$ contains more than one connected component, remove all but the connected component containing $\underline{D}(G)$. Then all cutnodes and bridges on a path between two nodes of $\underline{D}(G)$ are included in $\underline{D}(G)$.

Computing connected components can be done with DFS in time $O(m+n)$. Bridges, cutnodes, their components and condensed trees can be computed in linear time as a byproduct of DFS [114, 17].

Directed Connectedness Connectedness is an important notion for undirected graphs as well as for directed graphs. Two notions of connectedness exist for directed graphs, weak and strong connectedness. A directed graph is strongly connected if each of the nodes can reach each of the other nodes. It is weakly connected if the undirected view of the graph is connected.

The strongly connected constraint is listed in the catalog of global constraints [3]. Its potential pruning has also been mentioned in [22]. This notion plays an important role in networks as it models the reciprocal accessibility of a set of nodes. For instance, when deciding on the direction of one-way streets the mayor of the city must ensure that all houses stay accessible from every other.

Bound consistent pruning for this constraint is not easy to achieve. The pruning of the upper bound is similar to the connected constraint: the upper bound must contain only one strongly connected component. Strongly connected components can be computed in linear $O(m)$ time by a modified DFS [114]. For the lower bound, a brute force approach consists in taking each edge e of $\overline{D}(G) \setminus \underline{D}(G)$ and test if its removal breaks the strongly connected component. The edges which should be included in the lower bound are the edges that are said to "decompose" the strongly connected component of the upper bound. The test for strong connection must be made m times, hence an $O(m^2)$ complexity for computing bound consistency with a brute force approach. By using the best known algorithm for decremental strongly connected components ($O(mn)$ time [96]), it is possible to identify each of these edges by removing the edge, querying the updated data structure and restoring it to its original state.

A graph is weakly connected iff its undirected counterpart is connected. By definition, this constraint can be modeled by posting a *Connected* constraint on an undirected version of the directed graph (obtained with the *Undirected* constraint). The filtering algorithm for the undirected connected constraint can also be adapted to work on a directed graph by forgetting the direction of all arcs in the filtering algorithm.

4.4 No-Cycle Constraints

No-cycle constraints are constraints forbidding cycles. The DAG, forest and bipartite constraints all belong to this class of constraints.

4.4.1 The Directed Acyclic Graph Constraint

The $DAG(G)$ constraint holds if G is a directed acyclic digraph. It forbids any circuit in G.

Subtour elimination was presented for the `circuit` constraint of OPL in [120], it is later rediscovered in [22] for solving small TSPs with time-windows. In these works, it is used to avoid forming small cycles as the aim is to build an Hamiltonian cycle. We show that this rule performs bound consistent pruning for the DAG constraint. The constraint is monotonic: all subgraphs of a DAG are DAGs and all supergraphs of a cyclic graph are cyclic. If there is a circuit in $\underline{D}(G)$ then the constraint fails; If $\overline{D}(G)$ is a DAG then the constraint is entailed. Otherwise, for any partial path in $\underline{D}(G)$, remove all edges of $\overline{D}(G)$ which could create a circuit if they were included.

Theorem 4.6. *The $DAG(G)$ constraint is bound consistent iff $\overline{D}(G)$ does not contain arcs which close a circuit with $\underline{D}(G)$.*

Proof. The proof is by construction of solutions containing and excluding arcs of $\overline{D}(G) \setminus \underline{D}(G)$: $\underline{D}(G)$ is a DAG or the constraint is inconsistent, $\underline{D}(G)$ is a solution which excludes all non-mandatory arcs. Each arc of $\overline{D}(G)$ belongs to at least a solution: either it is part of $\underline{D}(G)$ or adding it to $\underline{D}(G)$ yields an acyclic graph. □

A naive algorithm for this constraint computes the transitive closure of the lower bound graph in $O(mn)$ (with n and m order and size of the lower bound) and performs an $O(1)$ query for each arc of $\overline{D}(G) \setminus \underline{D}(G)$.

4.4.2 The Bipartite Constraint

An undirected graph (N, E) is bipartite iff its node-set can be partitioned into two sets N_1 and N_2 such that each edge has its endpoints in both sets: $E \subseteq (N_1 \times N_2) \cup (N_2 \times N_1)$. Bipartite graphs are commonly used to model many interactions or assignment problems. For instance assign a job to a person, a resource to a task, or a male to a female.

Theorem 4.7. ([57]Bipartite Graph Characterization Theorem) *A graph is bipartite if and only if the length of each of its cycles is even.*

This graph property is monotonic, all subgraphs of a bipartite graph are bipartite and all supergraphs of a non-bipartite graph are non-bipartite. As for the DAG constraint, we need to check that the lower bound is bipartite and remove from the upper bound all edges that would make the lower bound non-bipartite. Then the lower bound is a solution and any edge of the upper bound participates in at least one solution.

Checking that a graph is bipartite can be done in linear time: build a spanning forest of the graph by applying DFS until all nodes are spanned. During that process, color the nodes by alternating between two colors. Clearly, the two endpoints of each edge of the forest have different colors (see figure 4.3). We need to check that the two endpoints of each remaining edge of the graph have different colors, if they do then the graph is bipartite. If they are not then the graph contains a cycle of odd length: The non-tree edge which has two endpoints (u, v) of the same color forms a unique cycle with the tree of the forest it is connected to. As the endpoints of the tree path from u to v have the same color, the tree path length is even and the cycle length is odd.

This leads to the following optimal filtering algorithm for pruning the upper bound: Once we have checked that the lower bound is bipartite, we keep the spanning forest and check the colors of endnodes of edges in the subgraph of $\overline{D}(G)$ induced by the nodes of $\underline{D}(G)$. Each edge which joins two nodes of different color is removed as it would create a odd-length cycle. The complexity of this filtering is linear $(O(m + n))$.

4.4.3 The $Tree(T)$ Constraint

The tree constraint holds if T is an undirected tree, *i.e.* a connected acyclic undirected graph. ALICE [76] had a *spanning tree* constraint and other tree

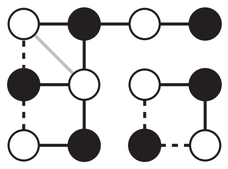


Figure 4.3: Checking bipartiteness. A 2-coloring of a spanning forest allows checking whether each non-tree edge belongs to an odd-length cycle. The solid black edges constitute the spanning forest. The dashed edges are non-tree edges which respect bipartiteness. The gray edge makes the graph non-bipartite.

constraints have been proposed: A integer model for building ultrametric supertrees (phylogenetic trees) was presented in [50]. The tree constraint of [7] partitions a graph in anti-arborescences, *i.e.* a forest of trees directed to their root. It was later extended into a unified tree and path partitioning constraint in [8].

These tree constraints are useful to model many network problems (e.g. the Steiner tree problem [67]). Trees are also central in phylogeny and the work of [7] was inspired by this application. In the next chapter we design filtering algorithms for two spanning tree related constraints: the weight bounded spanning tree and minimum spanning tree constraints.

Theorem 4.8. *The pruning obtained by applying the pruning rules of the DAG (avoiding cycles, undirected in this case) and Connected constraints is bound consistent.*

The proof is very similar to the proof for the connected constraint as the acyclic property only deals with the upper bound. Once undirected cycles have been forbidden, only nodes and edges which are necessary to connect $\underline{D}(G)$ need to be included in this bound.

If $\underline{D}(T)$ is empty, the constraint is bound consistent because any node or edge in $\overline{D}(T)$ belongs to a tree in $D(T)$; namely the tree consisting of a single node or a single edge. However, filtering may be necessary if $\underline{D}(T)$ is not empty.

If $\underline{D}(T)$ is not contained in a connected component of $\overline{D}(T)$, the constraint is inconsistent because T cannot be connected. Otherwise, all nodes and edges that are not in the same connected component with $\underline{D}(T)$ should be removed from $\overline{D}(T)$. Next, we need to find bridges and articulation nodes in $\overline{D}(T)$, *i.e.* an edge or a node whose removal disconnects the graph.

If such an element belongs to a path between two nodes of $\underline{D}(T)$, then it must belong to the tree and is inserted into $\underline{D}(T)$.

Finally, if $\underline{D}(T)$ contains a cycle the constraint is inconsistent, and an edge $e \in \overline{D}(T) \setminus \underline{D}(T)$ between two nodes that belong to the same connected component of $\underline{D}(T)$ must be removed from $\overline{D}(T)$, because including it in the tree would introduce a cycle.

All steps above can be computed in linear time with DFS [114].

4.5 The $Weight$ constraint

Many graph problems deal with weighted graphs. Depending on the problems, the costs/weights can be put on the edges (e.g. roads, communication lines), on the nodes (e.g. depots, routers) or even on both. In CP(Graph), weighted graphs are modeled by adjoining a weight function to a non-weighted graph variable.

The $Weight(G, W, I)$ constraint allows to model the total weight of a graph variable. We first present a constraint for the set weight then investigate how to adapt its pruning to graph weights. We consider fixed weights then cope with the adaptation of the algorithms to non-fixed weights. We show that even in the simple case of fixed weights and set variable, optimal filtering is NP-hard for the weight constraint. We present a bounded weight constraint both for sets and graphs and show that for the set weight the consistency which can be obtained by combining optimal filtering for two bounded weight constraints is equivalent to bound consistency for an arithmetic sum with a Boolean model of the set.

We show that graph weight is more complex than set weight because of the adjacency constraint which does not allow to treat graph elements independently of each other. We show that optimal filtering for the bounded graph weight is polynomial by presenting a naive $O(m^4)$ algorithm based on maximum flows.

4.5.1 The $Weight$ constraint for sets

$Weight(G, W, I)$ holds if I is the total weight associated to the graph variable G according to the weight function W. This constraint can be reformulated using the set weight constraint of CONJUNTO [54].

$$Weight(G, W, I) \equiv I = Weight(Nodes(G), W_n) + Weight(Arcs(G), W_a)$$

Where W_a is the restriction of the weight function to the arcs domain, and respectively, W_n for nodes. Hence if we only have node weights or arc weights we can use a set weight constraint.

In CP(Graph), a weighted set is modeled using a set interval together with a weight function which can be negative and non-fixed. However, in CONJUNTO, the weight constraint is restricted to fixed positive values and it applies to sets of pairs (value,weight). We show that the pruning rules of the set weight constraint are similar to those of a sum constraint over finite domains.

In CONJUNTO, the filtering algorithm for the set weight constraint computes the bounds of $D(I)$ as the weights of the bounds of $D(S)$. It also assigns the set variable to one of its bounds if its weight corresponds to the value of the opposite bound of the weight domain. It should be noted that this is however only valid for positive weights ($\mathbb{Z}^+$) since zero-weight subsets can be removed from or added to any solution.

The $Weight(S, w, I)$ constraint does not perform optimal pruning (bound consistency for the set variable or generalized arc consistency for its equivalent Boolean model). Arc consistency is known to be NP-hard for sum constraints by reduction from SUBSET-SUM: According to Theorem 4.1, if we had a polynomial bound consistent propagator for $Weight(S, w, I)$, we could use it to determine if a ground set s has a subset which sums to k : take $D(S) = [\emptyset, s]$, $D(I) = [k, k]$, the failure of the bound consistent propagator for $Weight(S, w, I)$ is the decision problem of SUBSET-SUM for s and k.

The problem of filtering both bounds can be modeled as an arithmetic sum with Boolean variables in the following way: $\sum_k w[k]B_k = I$ where $w[k]$ is the weight of element k and B_k is a Boolean variable stating the presence of k in S.

Bounds consistency for this sum constraint can be maintained for the $Weight(S, w, I)$ constraint by performing optimal filtering independently for these two constraints:

$$WeightLower(S, w, I) \wedge WeightHigher(S, w, I)$$

Where $WeightLower(S, w, I)$ holds if the weight of set S is not greater than I and $WeightHigher(S, w, I)$ holds if it is not lower than I. Optimal filtering for the $WeightLower(S, w, I)$ constraint amounts to finding all elements of S which must or cannot be present for the total weight to be lower than I.

We describe the filtering task for $Weight(S, w, I)$ constraint, but focus only on the $WeightLower$ component of the constraint, as the other is symmetrical. Let $minW(D(S), w)$ be the minimum weight among all sets in

$D(S)$. The minimum weight $minW$ can be computed by adding all negative weight elements of $\overline{D}(G)$ to $\underline{D}(G)$. Then,

$$\underline{D}(I) \leftarrow \max(\underline{D}(I), minW(S,w))$$

Let $Mandatory(S,w)$ denote the elements which must be added to the lower bound to make it bound consistent and $Forbidden(S,w)$ those which must be removed from $\overline{D}(S)$ for it to become bound consistent. These sets are:

$$\begin{aligned} Mandatory(S,w) &= \{x | minW\left([\underline{D}(S), \overline{D}(S) \setminus \{x\}], w\right) > \overline{D}(I)\} \\ Forbidden(S,w) &= \{x | minW\left([\underline{D}(S) \cup \{x\}, \overline{D}(S)], w\right) > \overline{D}(I)\} \end{aligned}$$

Example 4.9. Assume $D(S) = [\{0,1,-2\},\{0,1,-2,-3,100\}]$, $D(I) = [-10,10]$, and $w(x) = x$. The first rule recomputes the lower bound for $D(I)$:
$minW(D(S),w) = \sum\{0,1,-2,-3\} = -4$, hence $D(I)$ is set to $[-4,10]$. The second rule can detect that 100 cannot be part of S: $minW([\underline{D}(S) \cup \{100\}, \overline{D}(S)] = \sum\{0,1,-2,-3,100\} = 96 > \overline{D}(I)$

A simple $O(n)$ algorithm (with $n = \#\overline{D}(S)$) to perform this filtering proceeds as follows : First, in an initialization phase, the weights of undetermined elements (in $\overline{D}(S) \setminus \underline{D}(S)$) are sorted once for all further filtering tasks down the search tree. For each filtering task, the minimum weight subset is computed by adding the weights of all negative elements with the weight of the lower bound. Then the minimum weight element is checked to see if it must be included and the maximum weight element is checked to see if it must be excluded. If one of them do, then the second element is checked, and so on. The total complexity is $O(n)$ for computing the minimum weight subset plus amortized $O(Sort(n)/n)$ for the rest of the filtering task where $Sort(n)$ refers to the cost of sorting n values.

Dealing with non-fixed weights When filtering with non-fixed weight functions we still need to find minimum and maximum weight subsets. The major difference is that weights need to be filtered too. The weight function used in the $minW$ and $maxW$ computations can be transformed into ground weight functions. Let $\underline{D}(W)$ denote the minimal weight of each element in the domain of W and $\overline{D}(W)$ denote the maximal weight for each element, then: $minW(s,W) = minW(s, \underline{D}(W))$ and $maxW(s,W) = maxW(s, \overline{D}(W))$. To filter the upper bound of each weight, we consider the maximum weight $\overline{D}(I)$ acceptable for the set. For an element e, we look at the best case by assigning each other element of $\underline{D}(S)$ to its lowest possible weight. We then compute how much is left for e:

$$\overline{D}(W[e]) \leftarrow \\ \min\left(\overline{D}(W[e]), \overline{D}(I) - minW\left([\underline{D}(S) \cup \{e\}, \overline{D}(S)], \underline{D}(W)\right) + \underline{D}(W[e])\right)$$

The analysis to filter the lower bound of each weight is symmetric.

4.5.2 Graph weight

We present an algorithm to compute optimal filtering for the bounded weight subgraph constraint $WeightLower(G, w, I)$. As for its set counterpart, this constraint needs to compute for each arc e, the minimum weight subgraph in the intervals $[\underline{D}(G) \cup \{e\}, \overline{D}(G)]$ and $[\underline{D}(G), \overline{D}(G) \setminus \{e\}]$. When weights are all positive the solution is to take the lower bound of the interval. When nodes can be negative, the problem is similar to the minimum weight set problem. The solution is to select all negative weight nodes. The problem is more challenging when the arcs can be negative. Ideally the solution would simply select all negative arcs but as their endnodes can have positive weights, the problem consists in finding a set of arcs and their endnodes such that their sum is minimal. This problem looks combinatorial. We show that it is equivalent to the minimum induced subgraph problem. The latter problem was shown to be equivalent to a maximum flow problem [95]. We show that optimal filtering can be performed in polynomial time by presenting a naive $O(m^4)$ algorithm.

Computing the minimum weight subgraph

We first reduce the general minimum weight subgraph in a graph interval to the minimum weight induced subgraph problem. This problem has been applied to OR problems such as open-pit mining [66] or investment programs [95]. We show how this problem can be solved using a minimum cut maximum flow algorithm.

Definition 4.10. The original problem is the following: Given a graph interval $[g_1, g_2]$, and a weight function w (which specifies possibly negative weights $w[e]$ and $w[n]$ for each edge e and node n of g_2), find the graph g with $g_1 \subseteq g \subseteq g_2$ which minimizes the total weight of g.

We present a sequence of small results which allow to transform this problem into a min-cut-max-flow problem.

Lemma 4.11. *The minimum weight subgraph in a graph interval* $[g_1, g_2]$

- *does not contain positive weight edges of* $g_2 \setminus g_1$.
- *contains all negative weight nodes of* $g_2 \setminus g_1$.

Proof. Let g a minimum weight subgraph of $[g_1, g_2]$ and $e \in g_2 \setminus g_1$ one of its positive weight edges. $g \setminus e$ has a weight smaller than g, a contradiction. The case of negative weight nodes is similar. □

From now on we assume without loss of generality that all edges have negative weight and all nodes of negative weight belong to g_1.

Lemma 4.12. *The minimum weight subgraph in a graph interval* $[g_1, g_2]$ *with only negative weight edges is an induced subgraph of* g_2

Proof. Let g be a solution which is not an induced subgraph of g_2. By adding to g all negative weight edges which are needed for it to become an induced subgraph, we lower its total weight, a contradiction. □

Definition 4.13. A *negative marginal weight element* (NMWE) in a graph interval $[g_1, g_2]$ with a weight function w is a pair (sn, sa) of a set of nodes sn and a set of edges sa, both disjoint from the nodes and edges of g_1, whose union with g_1 is a graph and whose total weight is negative.

Lemma 4.14. *In a graph interval* $[g_1, g_2]$ *with only negative weight edges, the problem of finding the minimum weight subgraph can be reduced to the problem of finding a minimum weight induced subgraph in a graph by contracting all edges of* g_1 *and setting the weight of the resulting nodes to* 0.

Proof. By lemma 4.12, the solution g is determined by a set N of nodes which when added to the nodes of g_1 form the set of nodes of the solution. The solution can be partitioned into two sets of elements: the first set s_1 is composed of the subgraph of g_2 induced by the nodes of g_1, and the second set s_2 is the NMWE composed of the nodes N and all other edges in $g \setminus s_1$. The s_1 part is contained in every solution and directly determined by g_1 and g_2. Finding the solution amounts to finding the minimum weight s_2 part.

By contracting all edges of s_1 in g_1 and g_2, while adding the weights, the problem is reduced to a problem where g_1 is just a set of nodes (one per connected component of g_1). By turning the weight of these nodes to 0, the s_2 part becomes the minimum weight induced subgraph of the contracted g_2 graph. □

We now proceed by showing how this minimum induced subgraph problem in graph G with non-negative weight nodes and negative weight arcs can be mapped to a min-cut-max-flow problem in a new graph G_f. We call elements of G nodes and arcs while elements of G_f are called vertices and edges.

The flow graph G_f is a bipartite graph which vertices correspond to the nodes and arcs of G. On the top layer of Figure 4.4(c), each vertex models a node of G; We call them *node vertices*. On the lower layer, each vertex of G_f models a negative weight arc of G, we call them *arc vertices*. Uncapacited edges link nodes to their incident arcs (solid edges in the center of G_f). Two vertices, a source s and a sink t are added and capacited edges link s to the the node vertices and link arc vertices to t. The capacities are positive and correspond to the positive weight of a node or the opposite of the negative weight of an arc. As zero weight nodes (arising from the connected components of the induced subgraph built from g_1) have been turned into zero capacity edges they can be dropped for the flow computation (they are displayed in gray on Figure 4.4(c)).

We now prove that a minimum cut in this flow graph G_f corresponds to a minimum weight induced subgraph of G. Remember from section 2.2 that a cut is a connected set of nodes of the flow graph containing the source. The weight of the cut is the sum of the weights of the edges leaving the cut (displayed dashed in Figure 4.4(d)).

Theorem 4.15. *A minimum cut C in the flow graph G_f corresponds to a minimum weight induced subgraph of the original graph G: The nodes of G_f not in C constitute a minimum induced subgraph of G (displayed in gray in Figure 4.4(d)).*

Proof. This proof is illustrated in Figure 4.4; An example is also presented below. We first show that a minimum cut in the flow graph corresponds to a subgraph of the original graph, then we show it is an induced subgraph. Finally, we prove that this subgraph is a minimum induced subgraph.

The first property we prove corresponds to the adjacency constraint of graphs. As uncapacited edges linking node and arc vertices have infinite capacity, they cannot leave a minimum cut. If a node is removed from the solution, it is part of the cut. Then its arcs are also part of the cut since the uncapacited edges linking the node to the arcs cannot cross the cut. Therefore the arcs out of the cut have their endnodes out of the cut too, and the elements out of the cut form a graph.

The second property is the induced subgraph property, it can be expressed as: If an arc is removed one of its endnodes must be removed; This follows from connectedness of the cut.

We now prove minimality, *i.e.* that the minimum cut corresponds to the minimum weight subgraph. The weight of the cut is the weight of edges leaving the cut. As no uncapacited edges belong to these edges, they can be partitioned into two sets, those incident to the source and those incident to the sink. The total weight of the edges leaving the cut from the source is the total weight of the nodes we keep in the subgraph. The total weight of edges crossing the cut towards the sink is the opposite of the total weight of arcs which are discarded from the subgraph. Hence by minimizing the weight of the cut we minimize the weight of the kept induced subgraph. $\square$

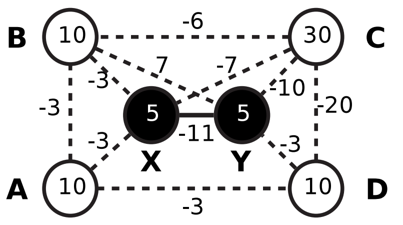
Example 4.16. In figure 4.4, an original graph, its corresponding flow graph and the minimum cut are illustrated. Note that the central node which is anyway part of the solution is not present in the flow graph. If it was included it would have an input edge with zero capacity and would not change the solution (we showed in lemma 4.14 that negative weight node can be set to 0). A max flow min cut computation computes the cut, all elements not in this cut need to be added to the mandatory elements (negative weight nodes and lower bound) to build a minimum weight subgraph in the graph interval.

Filtering algorithm

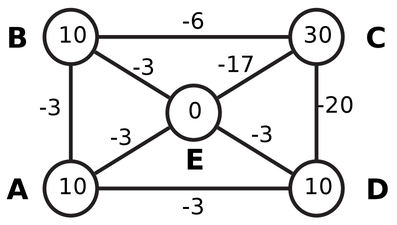
The filtering algorithm for the $WeightLower(G, w, I)$ constraint can be adapted from the algorithm for this constraint over sets and consists in the following steps.

Compute the minimum weight of a graph in the interval $[\underline{D}(G), \overline{D}(G)]$ and check that this weight is lower than $\overline{D}(I)$, if it is not, then the constraint fails. Then for each element x of $\overline{D}(G) \setminus \underline{D}(G)$ compute a minimum weight subgraph in the interval $[\underline{D}(G) \cup \{x\}, \overline{D}(G)]$. If its weight is greater than $\overline{D}(I)$, the element cannot be part of the solution and is excluded from $\overline{D}(G)$. Otherwise, compute a minimum weight subgraph of $[\underline{D}(G), \overline{D}(G) \setminus \{x\}]$ and check that its weight is not greater than $\overline{D}(I)$, if it is greater then include x in the lower bound.

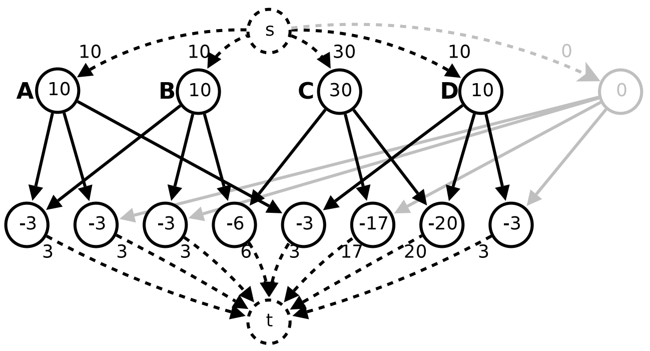
This brute force algorithm requires $O(m)$ maximum flow computations. At a cost of $O(m^3)$ per maximum flow, this algorithm is unpractical. (A max flow computation costs $O(nm^2)$ or $O(mn^3)$ with n and m the order and size of the flow graph, both linear in the number of edges of the original graph). We are investigating an algorithm which would not require $O(m)$



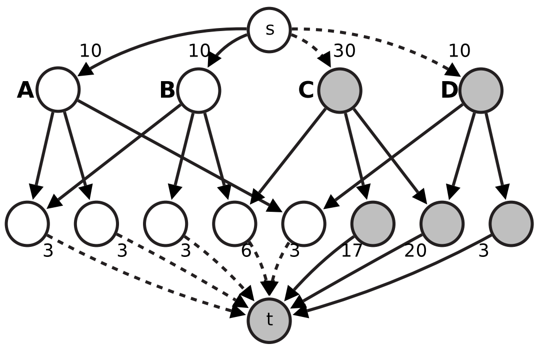
(a) Minimum weight subgraph problem in a graph interval



(b) Minimum weight induced subgraph problem



(c) Flow graph



(d) Cut (white) and crossing edges (dashed)

Figure 4.4: Successive transformations of the problem. Figure 4.4(a) presents the minimum weight subgraph problem in a graph interval $[g_1, g_2]$, g_1 is black and the rest of g_2 is drawn with dashed lines and empty nodes. Figure 4.4(b) presents the equivalent induced subgraph problem. The arc $\{B, X\}$ is removed and the arc $\{X, Y\}$ is contracted, the equivalent weight is -1 and is set to 0 in 4.4(b). Figure 4.4(c) presents the flow graph, the upper vertices correspond to the nodes of 4.4(b) and the bottom vertices to its arcs. The gray part corresponds to the zero weight node (condensed lower bound) and can be dropped as it cannot carry flow. Figure 4.4(d) presents a minimum cut (white). The other nodes, which constitute the solution to the previous problems is grayed. The crossing edges are dashed.

max flow computations but would try to adjust the flow computed for one element in order to test the next element.

4.6 The Path Constraints

The simple path constraint $Path(G, N_1, N_2)$ holds if G is a simple path (*i.e.* path without edge repetitions) from N_1 to N_2. The shorter path constraint $Path(G, N_1, N_2, W, I)$ holds if G is a simple path from N_1 to N_2 which total weight is no greater than I.

Constrained path problems are common in constraint programming. Vehicle routing problems or the TSP are example problems from OR directly modeled as constrained path finding problems. Constrained shortest path problems are found as sub-problems of many other problems in OR such as scheduling [58] or planning [59].

As a global constraint, the path constraint is closely related to the cycle constraint [6, 19] as a cycle is a closed path (*i.e.* $N_1 = N_2$). ALICE [76] contains an *Hamiltonian path* constraint. Optimal pruning for the path constraint is NP-hard by reduction from the Hamiltonian path problem and theorem 4.1: An Hamiltonian path can be modeled by including all nodes of a graph interval in its lower bound. Then detecting failure of the constraint solves the Hamiltonian path decision problem. A constructive reduction is given in [110] for mandatory edges (for a graph model in which nodes cannot be placed in the lower bound).

Graph theory tells us a simple path is a *DAG* and is *Connected*, it also tells us the target is reachable from the source (*QuasiPath*, see chapter 3). Hence filtering rules for these constraints can be applied to the filtering of a simple path constraint (see sections 4.4.1 and 4.3.2). The shorter path constraint which holds for a path of bounded length was introduced in [110] and extended in [109, 49]. A global simple path constraint was presented in our papers [35, 37] for undirected graphs and simultaneously presented in [21] for directed graphs. In [92], additional filtering rules are proposed for cases in which additional information is available about reachability, non-reachability and order of the nodes in the solution.

In this section, we give a structured presentation of the pruning rules for the simple path and shorter path constraints by showing which rules are common to the two constraints and which are specific to the shorter path constraint. We give an alternative algorithm saving one order of magnitude compared to that of [21] for filtering the lower bound of the graph for the directed simple path constraint. For the same problem, we also present a simpler (but weaker) algorithm by extending the algorithm presented in [49] to also deal with mandatory nodes (in [49], it deals only with mandatory edges). We finally discuss briefly the exact weight path constraint and extend the two global path constraints to non-fixed source and sink nodes and to non-fixed (interval) weights.

4.6.1 Cost-Based Filtering for the Shorter Path Constraint

The constraint $Path(G, n_1, n_2, W, I)$ holds if G is a simple path from n_1 to n_2 which total weight is no greater than I. It is an adaptation of the shorter path constraint of [110, 109, 49] which was presented for the Boolean arc model. We describe this constraint in the context of a graph computation domain. Adapting it to graph intervals allows to enhance the pruning rules: In addition to the detection of mandatory arcs, we show the algorithm presented in [49] can be extended to also detect mandatory nodes at the same asymptotic cost. We defer the handling of non-fixed source and target nodes and non-fixed weights to section 4.6.4.

Optimal filtering for this constraint is NP-hard, as the $Path(G, n_1, n_2)$ constraint is a particular case of the shorter path constraint with $I = \infty$. Hence, a relaxed-consistency is achieved instead.

As in [110], to test if an edge (u, v) is forbidden, the algorithm only looks at $\overline{D}(G)$ and computes a single source shortest paths from n_1 to all nodes in $\overline{D}(G)$ yielding a minimum distance $d(n_1, x)$ for all nodes x of the graph. A similar computation in the reverse graph of $\overline{D}(G)$ using n_2 as the source yields distances $d(x, n_2)$. Then, (u, v) is forbidden if $d(n_1, u) + w(u, v) + d(v, n_2) > \overline{D}(I)$. Hence the relaxed consistency property is: all arcs of $\overline{D}(G)$ belong to at least one path in $\overline{D}(G)$ from n_1 to n_2 with cost no greater than $\overline{D}(I)$.

The lower bound pruning does not use $D(I)$. Therefore, once $\overline{D}(G)$ has been pruned, the pruning done for $\underline{D}(G)$ for the shorter path constraint is the same as for the simple path constraint. We present this pruning for the directed and undirected cases in the next sections about the simple path constraint.

For fixed-point reasoning, as the filtering of $\overline{D}(G)$ only considers $\overline{D}(G)$ itself, the filtering of $\underline{D}(G)$ does not impact on $\overline{D}(G)$ and the computation has reached a fixed point whatever the pruning done for $\underline{D}(G)$.

The complexity of the cost-based upper bound pruning is $O(m + n \log n)$ by using Dijkstra. If negative weight edges are present one application of Bellman-Ford $O(mn)$ computes reduced costs as in Johnson's all pair shortest path to lift the negative weights and make them positive. Practical dynamic single source shortest path algorithms [28] such as Ramalingam-Reps [94] could be used to speed up this computation.

4.6.2 Undirected $Path(G, n_1, n_2)$

This constraint holds if G is an undirected simple path from n_1 to n_2.

In addition to the pruning of degree constraints provided by the *QuasiPath* constraint, we reason on the bridge-reduced and cutnode-reduced trees (see definition in section 2.2). We identify a set of mandatory and forbidden nodes and arcs *i.e.* nodes and arcs which must be part of the solution if there is one and, respectively, which are part of no solutions.

The filtering rules of the *QuasiPath* constraint are simple. Each node of the graph, except n_1 and n_2 must have a degree of 2. The nodes n_1 and n_2 must have a degree of 1. As soon as this value of degree is reached in the upper bound, the arcs are included in the lower bound. As soon as it is reached in the lower bound, the other arcs of the upper bound incident to that node are removed. By modeling the *QuasiPath* problem as a cycle cover problem as in section 3.6.1 (this requires adding an arc (n_2, n_1) in the lower bound), a global filtering is possible by using the filtering algorithm of the alldiff constraint. This global filtering is not necessary in the context of the path constraint as the other filtering rules of the path constraint take care of this pruning.

The following pruning rule results from the undirected connected constraint of section 4.3.

Theorem 4.17. *Bridges and cutnodes of* $\overline{D}(G)$ *on a path between two nodes of* $\underline{D}(G)$ *are mandatory (part of all solutions).*

Proof. If a bridge (or cutnode) is on a path between two mandatory nodes then it is mandatory: if we remove it, those nodes are no more in the same connected component and cannot be joined by a simple path. □

As the path we are looking for must be simple and as the reduced graphs are trees, we can identify some forbidden nodes and arcs (this is on of the contributions of our papers [35, 37]).

Theorem 4.18. *Bridges of* $\overline{D}(G)$ *which are not on the path from* n_1 *to* n_2 *in the bridge-reduced tree of* $\overline{D}(G)$ *are forbidden.*
Cutnodes of $\overline{D}(G)$ *which are not on the path from* n_1 *to* n_2 *in the cutnode-reduced tree of* $\overline{D}(G)$ *are forbidden.*

Proof. Consider the bridge-reduced tree of $\overline{D}(G)$ (or its cutnode-reduced tree). In this tree there is only one path between n_1 and n_2. If a simple path goes through another edge e of the tree, it is not able to reach n_2 unless it traverses that edge e in the other direction, a contradiction as the path must be simple. □

In [114], a linear algorithm based on DFS that computes biconnected components is presented. The principle of this algorithm can be applied to compute 2-edge connected components, cutnodes, bridges and reduced trees. The filtering based on cutnodes and bridges can be performed in linear time.

Using bridges for pruning the lower bound for this constraint is presented in detail in the undirected shorter path constraint of [110]. Bridge and cutnode pruning from the upper bound was presented in our papers [35, 37] and later independently discovered in [49].

4.6.3 Directed $Path(G, n_1, n_2)$

As in the undirected case, optimal filtering for this constraint is NP-hard.

Forbidden Arcs

The upper bound pruning obtained by applying the filtering of the undirected approach is subsumed by the approach presented in the global path constraint of [21] which consists in contracting strongly connected components to produce an SCC-reduced DAG. In a DAG, bound consistency for a path constraint can be done in linear time: Intuitively, in the following theorem, the forbidden arcs are those which either are not reachable from the source node or cannot reach the target node or jump over a mandatory node. This is presented on figure 4.5.

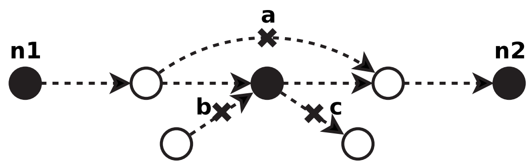


Figure 4.5: Forbidden nodes identified by theorem 4.19. Arc a jumps over a mandatory node, arc b does is not reachable from n_1 and arc c does not reach n_2.

Theorem 4.19. **([21][Property 1-2])** *Let $TC(\overline{D}(G))$ denote the set of arcs in the transitive closure of $\overline{D}(G)$, An arc $(u, v) \in \overline{D}(G)$ is forbidden if*

$$\begin{aligned}(n_1, u) \notin TC(\overline{D}(G)) \vee (v, n_2) \notin TC(\overline{D}(G)) \vee \\ \exists n \in Nodes(\underline{D}(G)) : (n, u) \notin TC(\overline{D}(G)) \wedge (v, n) \notin TC(\overline{D}(G))\end{aligned}$$

In a DAG, this requires two DFS computations (one from n_1, and one from n_2 in the reverse graph) for the reachability and a topological sort for the arcs jumping over nodes of $\underline{D}(G)$, for a total time of $O(m+n)$ where m and n are respectively the size and order of $\overline{D}(G)$.

Mandatory Nodes

In [21], Cambazard and Bourreau also propose to test for mandatory nodes by removing each node from the graph in turn and checking that all nodes of $\underline{D}(G)$ are still reachable from n_1 and still reach n_2. This algorithm has a total cost of $O(nm)$. We propose a linear $O(m)$ algorithm to perform this pruning.

In [92], a dominators algorithm is used to detect some of these mandatory nodes. By applying a dominator computation [51] to the graph and its reverse, we can identify all mandatory nodes of [21]: all nodes which dominate a node of $\underline{D}(G)$ are mandatory. This lowers the cost of this computation to $O(m)$.

Mandatory Edges and Forbidden Nodes

Mandatory edges are detected by the the degree constraints: if an edge (u, v) is a bridge, its endnodes are mandatory and the out-degree of u and in-degree of v are 1.

Isolated nodes are forbidden in the graph. If a node is forbidden then all incident edges must be forbidden too and the node will get isolated after the forbidden edges are removed.

A Simpler but Weaker Algorithm for Mandatory Elements

In [109], an algorithm to detect some mandatory edges in $O(m + n \log n)$ is presented. In [49] it is simplified and runs in linear time $O(m + n)$. These algorithms compute the mandatory edges which belong to the shortest path

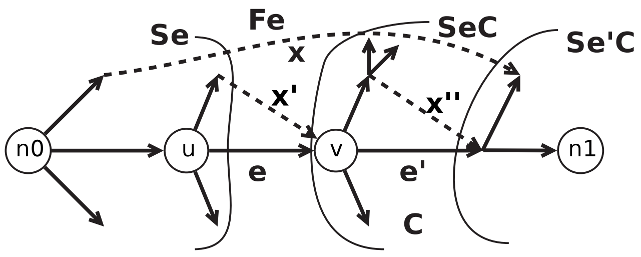


Figure 4.6: Detection of mandatory edges along the shortest path from n_0 to n_1. The edges of $F_e = \{x, x'\}$ are drawn with dashed arrows, the shortest path tree is drawn with solid arrows and other edges are not drawn. If x' was not present, then node v would dominate n_1. When moving to edge e', the edge x', a replacement for e would be removed as it is incident to v a node of C, leaving F_e empty. But as x' and x'' are replacement for respectively e and e' those edges are not mandatory.

from n_1 to n_2 and dominate n_2 (*i.e.* a subset of those detected with the dominators but at a lower cost).

We describe this algorithm and show how to adapt it to detect dominating nodes (see Figure 4.6). The algorithm uses the shortest path tree t from the source n_1 in $\overline{D}(G)$ which is computed to filter the upper bound (see section 4.6.1). The algorithm inspects each edge $e = (u, v)$ of the shortest path in turn and computes the set F_e of edges which can replace e in a path from n_1 to n_2. If this set is empty, the edge e is mandatory.

The algorithm maintains two sets of nodes S_e and S_e^C: the nodes of the two subtrees of $t \setminus \{e\}$. It also maintains the set of edges F_e from S_e to S_e^C; These edges can replace e to build a new admissible path. We extend this definition to F_v, the set of edges which could replace node v in the path.

When going from the edge e to the next edge $e' = (v, w)$, the algorithm updates its data structures in the following way. The set $C = S_{e'} \setminus S_e$ of all nodes connected to v in $t \setminus \{e, e'\}$ must be removed from S_e^C and added to S_e in order to build $S_{e'}^C$ and $S_{e'}$. The set $F_{e'}$ is F_e minus the edges coming into C plus the edges going out of C (to $S_{e'}^C$, all other edges have been removed in a previous step).

This is the point where mandatory nodes can be detected: When updating F_e to build $F_{e'}$, after deleting all incoming edges to C and before adding the outgoing ones, we have a state where the set data structure F is actually F_v the set of all edges which could replace node v if it was removed. If this set is empty, then node v is mandatory.

4.6.4 Non-fixed Source, Target and Weights

When dealing with simple paths it might be useful to be able to model some uncertainty about the origin or destination of the path. By allowing the source and target nodes of a path constraint to be non-fixed variables, it is possible to model that uncertainty. The semantic is not changed: The constraint $Path(G, N_1, N_2)$ still states that G is a path from node N_1 to node N_2.

As the nodes are not fixed, the previous filtering algorithm must be adapted. It must detect impossible and mandatory nodes and edges for a path that goes from any node in $D(N_1)$ to any nodes in $D(N_2)$. The filtering problem also includes a step to filter the domain of the source and target node variables.

A simple solution to this problem is presented in figure 4.7: Add a virtual source s and target t to the upper and lower bound graphs. In $\overline{D}(G)$, connect s with all nodes in $D(N_1)$ and connect each node of $D(N_2)$ with t. For the shorter path constraint, the added edges are assigned zero weight. Then the set of all (possibly empty) simple paths from a node of $D(N_1)$ to a node of $D(N_2)$ is the set of all simple paths from s to t. All pruning of G can be made as in the fixed case. To filter N_1 and N_2, we add a simple rule to the filtering algorithm: when it removes an edge connecting one of the virtual nodes, the adjacent node is removed from the domain of the corresponding node variable (for instance, if an edge (u, t) is removed, node u is removed from $D(N_2)$).

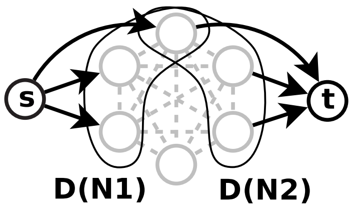


Figure 4.7: Dealing with non-fixed source and sink in the path constraints. A virtual source (s) and target (t) are added to the graph (in gray). Arcs are added to the nodes in the domains of the real source (N_1) and target (N_2). This allows to reuse the previous filtering algorithm.

To cope with non-fixed weights in the $Path(G, n_1, n_2, W, I)$ constraint, we allow each value $W[e]$ of the weight function to be an interval which end-points are denoted $\underline{D}(W[e])$ and $\overline{D}(W[e])$. To filter G, we only need to adapt the cost-based filtering: the filtering of $\overline{D}(G)$ presented in Section 4.6.1. The shortest path computations are done with the lower bound of the weights in order to actually measure the shortest possible path. Then, when each edge (u, v) is checked to see if it can belong to an admissible-weight path from n_1 to n_2, we again use the lower bound of the edge weight. The difference with the fixed weight case is the filtering of the weights: a new upper bound is computed for the weight of each edge (u, v): $\overline{D}(I) - d(n_1, u) - d(v, n_2))$.

The lower bound of edge weights is not filtered. If a shorter path constraint holds for fixed weights w then it holds for any w' such that $\forall e : w'[e] \leq w[e]$. This pruning would however be necessary for a $ShortestPath(G, P, W)$ constraint where non-path edges (not in $\overline{D}(P)$) would not be allowed to be too light or they would replace an edge in the shortest path P. This kind of reasoning is held for the minimum spanning tree constraint which is presented along with the weight-bounded spanning tree in next chapter.

4.6.5 Negative Cycles and the Exact Weight Path Constraint

In this short section we describe the problem raised by negative cycles for shortest paths and show that cost-based filtering is probably not practical to deal with. This leads to the same consideration for the exact weight path constraint.

If the *Path* constraint is to be used with graphs which contain negative cycles, the cost-based filtering algorithm described in the previous sections does not apply as it computes shortest paths which are infinite in such a graph.

The problem of finding the shortest *simple* path in a graph with negative cycles is equivalent to finding the longest simple path in a graph (take opposite weights) which is known to be NP-complete [46]. The problem of the shortest simple paths which allows negative cycles has been tackled in [38] under the name Elementary Weight Constrained Shortest Path Problem (EWCSPP) where the cost of a simple path is to be minimized under some additional non-negative weight bound constraints (negative cost cycles are allowed).

To build a filtering algorithm for the upper bound of the domain under the presence of negative cycles, an approximation for the point to point shortest simple path computation could be used to compute the cost of paths containing or excluding elements of the upper bound. However, unless P=NP there is no constant factor approximation for this problem and probably no $O(n^\delta)$ approximation neither [70]. Only logarithmic approximations have been proposed. Hence filtering based on such an approximation is improbable.

Some models need to use the exact cost of a path. We suggest that the weight constraint is more appropriate than an exact weight global path constraint. For such a constraint to use the lower bound of weight for filtering, the propagator would have somehow to compute the total weight of the longest possible path which could use or avoid a given element. Because of the above non-approximability results, it is improbable that such an algorithm would prune values from the graph.

4.7 The Transitive Closure Constraint

The $TC(G, G^+)$ constraint holds if G^+ is the transitive closure of G. The transitive closure constraint $TC(G, G^+)$ can be viewed as a generalization of the path constraint. The propagation uses concepts similar to the concepts used in the connected and path constraints. The transitive closure constraint is a part of the reachability constraint of Quesada et al. [92]. We show that optimal filtering is NP-hard for this constraint and describe optimal filtering rules for cases where at least one variable is fixed.

Optimal filtering for this constraint is shown to be NP-hard [91] by Theorem 4.1 and a reduction from the disjoint path problem [46]. This decision problem concerns the existence in a graph g of two node-disjoint point to point simple paths from i to j and from k to l. Assume $D(G) = [\emptyset, g]$ and $D(G^+)$ such that $\overline{D}(G^+)$ contains all edges of g except (i, l) and (k, j) and let $\underline{D}(G^+)$ contain only (i, j) and (k, l). To see that these are equivalent, observe the two following properties: In an hypothetical graph g', if $(i, j) \in TC(g')$, then j is reachable from i and that graph contains a path from i to j. Moreover if any two paths from i to j and from k to l share at least a node, they contain an X shape, and l is reachable from i and j from k.

The case where G is fixed and denoted g is the transitive closure computation. The optimal filtering consists in computing the transitive closure of g and setting G^+ to $TC(G)$. The case where only G^+ is fixed (and denoted g^+) and where we are looking for a satisfying G is known as transitive reduction or equivalent digraph [123]; Its computation costs $O(n^3)$.

Pruning rules for this constraint are presented for a single source reachability constraint in [92]. They use the computation of the transitive closure of the graphs $\underline{D}(G)$ and $\overline{D}(G)$ and the computation of dominators from the source s in $\overline{D}(G)$. For the transitive closure constraint this computation can be done from each node and a dominator of a node which should be reachable from another must be included in the lower bound.

This reachability constraint is the topic of the thesis of L. Quesada and the transitive closure constraint will be further described in that thesis.

4.8 Summary

In this chapter we studied filtering algorithms for global constraints modeling graph properties. While some pruning rules for some of these constraints can be found in previous works, this study constitutes a corpus of pruning rules which can be enforced in CSPs using graph properties, regardless of their implementation. This study provides optimal filtering rules for most of the constraints or proves NP-hardness. Filtering algorithms are presented and their complexity and practicality is discussed. For some of the constraints only naive algorithms are provided, leaving the design of more efficient algorithms as an open problem. We believe that this chapter illustrates that graph intervals are a practical abstraction for describing and proving filtering algorithms for graph constraints regardless of the underlying model used for the graph interval (graph, set, integer or Boolean variables).

A summary of our results is presented in table 4.1. In this table, the first column identifies the constraint. The second column lists some notable properties of the constraint. We indicate if the lub or glb is a solution once the graph interval domain is bound consistent. We also list redundant properties for the *Path* constraints. The third column indicates the complexity of computing bound consistency or relaxed consistency if bound consistency is NP-hard.

Constraint	Properties	Bound Consistency
$Subgraph(G_1, G_2)$	lub and glb	$O(m+n)$
$Symmetric(G)$	lub and glb	$O(m+n)$
$Undirected(G, G_u)$	lub	$O(m+n)$
$Connected(G)$	lub	$O(m+n)$
$WeaklyConnected(G)$	lub	$O(m+n)$
$StronglyConnected(G)$	lub	$O(mn)$
$Forest(G)$	glb	$O(m+n)$
$DAG(G)$	glb	$O(nm)$ in $\underline{D}(G)$
$Bipartite(G)$	glb	$O(m+n)$
$Weight(S, W, I)$		NP-hard
$Weight(G, W, I)$		NP-hard
$WeightLower(S, W, I)$	$W \geq 0$: glb	$O(n + n \log n)$
$WeightLower(G, W, I)$	$W \geq 0$: glb	$O(m^4)$
$Tree(G)$	$Connected$ $Forest$	$O(m+n)$
$Path(G, N_1, N_2)$	$QuasiPath$ $Connected$ DAG	NP-hard or $O(m+n)$
$Path(G, N_1, N_2, W, I)$	$Weight(G, W) \leq I$ $QuasiPath$ $Connected$ DAG	NP-hard or $O(m + n \log n)$
$TC(G, G^+)$		NP-hard or $O(n^3 + nm)$

Table 4.1: Summary of our main results in this chapter

Chapter 5

Spanning Tree Constraints

5.1 Introduction

The MST problem arose in the context of a network design problem for a telephone company. The problem is to find the cheapest way of connecting all the telephone centers in a country.

Finding the MST of a graph takes almost-linear time, but several interesting variants of the MST problem, such as minimum k-spanning tree [47] (finding a minimum-weight tree that spans any set of k nodes) and Steiner tree [67] (finding a minimum-weight tree that spans a given set of nodes) are hard problems which have applications as other network design problems.

In many applications, there is uncertainty in the input. The weight of edges might not be known precisely. For instance, when installing cables underground, the cost is mostly related to digging the trenches and this cost might depend on what is found in the soil (pollution or archaeological sites might increase the cost).

For instance, the MST sensitivity analysis problem [34, 86] and the robust spanning tree problem [1] both deal with unknown edge weights. In the MST sensitivity analysis problem, we are given a fixed graph g with fixed weights and an MST t of g. We need to determine, for each edge, the amount by which its weight can be changed without violating $t = MST(g)$.

The robust spanning tree problem addresses uncertainty from a different point of view. Its input is a graph with an interval of possible edge weights for every edge. A *scenario* is a selection of a weight for each of the edges. For a spanning tree t and a scenario s, the *regret* of t for s is the difference between the weight of t and the weight of the MST of the graph under scenario s. The output is a spanning tree that minimizes that worst-case regret.

There might also be uncertainty about the graph in which one has to compute a minimum spanning tree: you might want to start installing the network while all contracts with clients have not been signed. In that case it would be nice if the first installed cables belong to the network regardless of the success of these contract negotiations.

Finally, there are cases where the tree is known and the question is what graphs have this tree as a minimum spanning tree. For instance, a network operator might want to influence the cost evaluation or the set of sites a potential client wants to connect in order for his own network to be selected by the client. This is the focus of inverse parametric optimization problems [41]. In that problem, we are interested in finding parameters of an optimization problem given its solution. For instance, given a tree, determine which graphs have this tree as a minimum spanning tree.

In this chapter we present the $WBST(G, T, W, I)$ and $MST(G, T, W)$ constraints. These constraints are specified on two graph variables G and T, a weight variable W and an integer variable I. The $WBST$ constraint holds if T is a spanning tree of G (i.e., a connected acyclic subgraph containing all nodes of G) such that the total weight of T is at most I, with the positive edge weights specified by W. The MST constraint is satisfied if T is a minimum spanning tree (MST) of G (i.e., the minimum-weight connected subgraph that contains all nodes of G), where the positive weights of the edges in G and T are specified by W.

These constraints address the problem of finding small spanning trees in a novel way: The filtering algorithms need to propagate information from each parameter to the others in order to reach a fixed point. And these parameters are graph or weight intervals which allow to model uncertainty about each of them. A filtering algorithm for the minimum spanning tree constraint embraces the problems of minimum spanning tree verification, minimum spanning tree sensitivity and inverse parameter optimization in one single algorithm.

A large body of research exists on weighted spanning trees of undirected graphs; in particular on the MST problem: From algorithms that find the MST [18, 68, 71, 75, 87, 88] to related problems such as MST verification [74], finding several smallest trees [40] and sensitivity analysis. We reuse some of these results for the design of our filtering algorithms. In particular, we use components of Eppstein's algorithm for computing the k smallest trees [40] and of King's algorithm for MST verification [74].

In a CP approach to the robust spanning tree [1], an algorithm that partially solves a special case of the filtering algorithm for MST that we

	Fixed Edge-Weights		**Non-Fixed Edge-Weights**	
	Fixed Graph	Non-Fixed Graph	Fixed Graph	Non-Fixed Graph
Fixed Tree	MST verification $O(m+n)$ [74]	$O(m+n)$ [Section 5.4]	$O(m\alpha(m,n))$ [Section 5.7.1]	$O(m\alpha(m,n))$ [Section 5.7.2]
Non-Fixed Tree	$O(Sort(m)+ m\alpha(m,n))$ [Section 5.5]	$O(Sort(m)+ m\alpha(m,n))$ [Section 5.6]	$O(Sort(m)+ m\alpha(m,n))$ [Section 5.7.3]	$O(mn(m+\log n))$ [Section 5.7.4]

Table 5.1: Our results for the *MST* constraint

address in this chapter is described. More precisely, for the case in which the graph is fixed, they show how to detect edges which must be removed from the upper bound of the tree domain.

Our results. For the *WBST* constraint, we present an NP-hardness proof for bound consistent pruning in the general case. We give a similar result for *EWST*, the exact weight minimum spanning tree constraint where I is not an upper bound but the exact weight of the tree. We restrict the constraint to operate on a graph with fixed nodes and give an almost-linear $O(m\alpha(m,n))$ bound consistency filtering algorithm for this special case.
Our results for the *MST* constraint are summarized in Table 5.1. We look at various restrictions of the *MST* constraint, and develop a bound consistency algorithm for each of them. Let n and m be, respectively, the number of nodes and edges in the upper bound of the domain of G (and hence also of T). Let $Sort(m)$ be the time it takes to sort m edge-weights and α the slow-growing inverse-Ackerman function. As the table indicates, our algorithm computes bound consistency for the most general case in cubic time but whenever the domain of one of the variable is fixed, bound consistency can be computed in almost-linear time. The table indicates in which section of the chapter each case is handled.

Roadmap. The rest of the chapter is structured as follows. Section 5.2 contains some preliminaries on minimum spanning trees. In section 5.3 we describe a filtering algorithm for the *SpanningTree* constraint which is used as a preprocessing step and whose result is maintained as an invariant during the execution of the algorithms described in subsequent sections. The bulk of the chapter starts with sections 5.4 to 5.7, we describe the bound consistency algorithms for special cases of the *MST* constraint. Starting with the case of fixed tree and fixed weights and gradually building up to the most general case. In section 5.8 we derive the NP-hardness results for the filtering of the *EWST* and *WBST* constraints. Then as for the *MST* constraint, we address the filtering problem for the *WBST* constraint in sections 5.9 to 5.11.

5.2 Preliminaries on Minimum Spanning Trees

In this section we present a short background of graph theory and algorithms on spanning trees.

5.2.1 Graph Theory

Two important properties hold for minimum spanning trees: the cut and the cycle properties.

Definition 5.1. A cut (also called edge cut) in a connected graph $G = (V, E)$ is a set S of edges such that $G' = (V, E \setminus S)$ is not connected.

Property 5.2 (Cut property)**.** *Let T be a minimum spanning tree of G. In any edge cut S of G, an edge of minimum weight is part of T.*

Proof. We show the property holds for a minimum spanning tree by building a lighter spanning tree if it does not. Assume T is an MST of G and there is a cut s splitting T into T_1 and T_2 such that a lightest edge $e = \{u, v\}$ does not belong to T. There must be an edge e' of T which connects T_1 and T_2, and this edge must be in s. By replacing e with e' in T we build a lighter spanning tree. □

Property 5.3 (Cycle property)**.** *Let T be a minimum spanning tree of G. In any cycle C of G, one of the edges of maximum weight is not part of T.*

Proof. We build again a lighter spanning tree if the property does not hold. Assume T is an MST of G and there is a cycle C in G with the heaviest edge $e = \{u, v\}$ belonging to T (without loss of generality we assume there is only one such edge). The edge e splits T into T_1 and T_2. The cut s splitting G into the node sets of T_1 and T_2 contains e and at least one additional edge e' of c. By replacing $e \in T$ with e', which is lighter, we build a lighter spanning tree. □

The following definitions and lemmas cope with multiple minimum spanning trees. They are illustrated in figure 5.1.

Definition 5.4. Let T be a minimum spanning tree of G and let $e \in T$ be an edge whose removal from T disconnects T into two trees T_1 and T_2. Define the replacement edge $r_G(e)$ to be a minimum weight edge in G other than e which connects a node from T_1 and a node from T_2. The replacement weight of e is the weight of $r_G(e)$ or ∞ if there is no such edge.

Definition 5.5. Let T be a minimum spanning tree of G and $e = \{u, v\}$ a non tree edge ($e \notin T$), we define a replacement edge $r_G(e)$ of e as an edge of maximum weight along the tree path joining u and v. The replacement weight of e is the weight of a replacement edge.

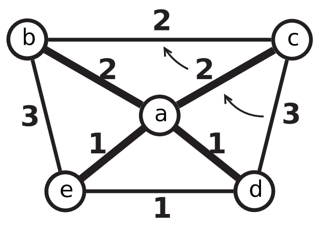


Figure 5.1: Replacement edges. The replacement edge of the tree edge $\{a, c\}$ is $\{b, c\}$. The replacement edge of the non-tree edge $\{c, d\}$ is $\{a, c\}$. The MST is highlighted with bold edges.

Two important lemmas characterize the replacement edges.

Lemma 5.6 ([40], Lemma 3)**.** *For any edge e in an MST t of g such that $g \setminus \{e\}$ is connected, $(t \setminus \{e\}) \cup \{r_g(e)\}$ is an MST of $g \setminus \{e\}$.*

Proof. Let us consider all cycles in g; the set of cycles in $g \setminus \{e\}$ is a subset thereof. We show by contradiction that the weight of $r_g(e)$ cannot be greater than the maximum weight in a cycle not containing e. Therefore we show that $(t \setminus \{e\}) \cup \{r_g(e)\}$ satisfies the cycle property. We use the fact that an edge cut must contain at least two edges of a cycle. Let $c \subseteq g \setminus \{e\}$ be a cycle and let us assume that $\forall e' \in c \setminus \{r_g(e)\} : w(r_g(e) > w(e')$. Let t_1 and t_2 be the two subtrees of $t \setminus \{e\}$ and let s be the edge cut splitting g in (t_1, t_2). The edge cut s contains $r_g(e)$ and at least one other edge e' of c. As $w(r_g(e)) > w(e')$, we have a contradiction with the definition of $r_g(e)$, a minimum weight edge of $s \setminus \{e\}$. $\square$

Lemma 5.7. *For any edge $e \notin t$, $(t \cup \{e\}) \setminus \{r_g(e)\}$ is a minimum spanning tree of g that contains the edge e.*

Proof. Contract e and find an MST of the remaining graph. By the cycle property, it will exclude one of the maximum weight edges on the cycle that was created by the contraction. □

5.2.2 Algorithms

The three most famous algorithms for computing minimum spanning trees are Boruvka's [18], Prim's [88] and Kruskal's [75]. They are based on the cut property. As we adapt Kruskal's algorithm in following sections, we present the original version here in Algorithm 3. The algorithm builds a minimum spanning tree by inserting edges in a spanning forest by increasing cost. It uses the disjoint set [115] data structure to maintain a partition of the nodes of the graph in the connected components of the spanning forest. This data structure supports the $Union(x, y)$ operation which merges the sets containing elements x and y and the operation $Find(x)$ which returns an identifier of the set containing x.

```
Data: $G = (N, E)$ an edge-weighted graph: $W[e]$ is the weight of
        edge $e$
Result: $T$ the set of minimum spanning tree edges
begin
    $P \longleftarrow Sort(E)$ (increasing order)
    $UF \longleftarrow$ partition with singletons for each $n \in N$
    $T \longleftarrow \emptyset$
    while $P \neq \emptyset \wedge |T| < |N| - 1$ do
        $e = \{u, v\} \longleftarrow$pop minimum value from $P$
        if $Find(UF, u) \neq Find(UF, v)$ then
            $Union(UF, u, v)$
            $T \longleftarrow T \cup \{e\}$ ;
        end
    end
end
```

Algorithm 3: Kruskal's algorithm for minimum spanning trees

Given a graph g and an MST t of g, the replacements $r_g(e)$ for all edges in t can be computed in time $O(m\alpha(m, n))$. Finding the replacement edge for every non-tree edge can be done in linear-time on a RAM using a component of King's algorithm for MST verification [74]: King described how, in linear

time, we can determine the weight of the heaviest edge on the path between every two tree nodes.
Finding the replacement edge for every tree edge can be done in $O(m\alpha(m,n))$ using the path compression technique of Tarjan [116]. This algorithm was used by Eppstein in his k minimum spanning tree algorithm [40].

5.3 The Spanning Tree Constraint

Throughout the chapter, we will assume that the following bound consistent filtering for the $SpanningTree(G,T)$ constraint was performed in a preprocessing step. Bound consistent filtering for this constraint is equivalent to the conjunction of the bound consistent filtering for $\mathit{Nodes}(G) = \mathit{Nodes}(T)$, $\mathit{Subgraph}(T,G)$, $Tree(T)$ and $\mathit{Connected}(G)$. We describe the pruning steps:

First, the algorithm applies the bound consistency algorithms described in chapter 4 for the constraints $\mathit{Nodes}(G) = \mathit{Nodes}(T)$, $\mathit{Subgraph}(T,G)$ (which specifies that T is a subgraph of G) and $\mathit{Tree}(T)$. Filtering for the $\mathit{Tree}(T)$ constraint removes from $\overline{D}(T)$ each arc whose endnodes belong to the same connected component of $\underline{D}(T)$. The inclusion of any of those arcs in the lower bound would create a cycle as its endnodes are already connected.

Then it enforces that T is connected: If there are two nodes in $\underline{D}(T)$ that do not belong to the same connected component of $\overline{D}(T)$ then the constraint has no solution. Otherwise, any bridge or cut-node in $\overline{D}(T)$ whose removal disconnects two nodes from $\underline{D}(T)$ is placed in $\underline{D}(T)$.

The algorithm applies the filtering for the $\mathit{Connected}(G)$ constraint. As the conjunction of $\mathit{Tree}(T)$ and $\mathit{Subgraph}(T,G)$ includes in $\underline{D}(G)$ all the bridges and cut-nodes in between nodes of $Nodes(\underline{D}(T)) = Nodes(\underline{D}(G))$, this filtering step only consists in setting $\overline{D}(G)$ to its only connected component containing $\underline{D}(G)$.

Finally, since $\underline{D}(T)$ is contained in the MST in any solution, we reduce the problem to the case in which $\underline{D}(T)$ is initially empty, as follows. We contract all edges of $\underline{D}(T)$ in g and obtain the graph g'. An edge is contracted by merging its endnodes into a single node. For any MST t' of g', the edge-set $t' \cup \underline{D}(T)$ is a minimum-weight spanning tree of g that contains $\underline{D}(T)$.

5.4 *MST* with Non-Fixed Graph

To simplify the exposition of the algorithm, we begin with a bound consistent algorithm for the special case $\mathit{MST}(G,t,w)$ in which the variables T and W

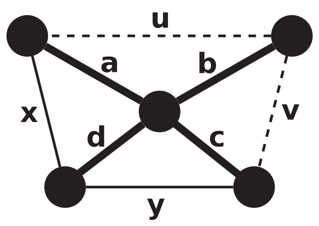


Figure 5.2: MST with fixed tree (bold edges: $t = \{a, b, c, d\}$) and non-fixed graph (solid edges $\{y, x\} = \underline{D}(G)\backslash t$ and dashed edges $\{u, v\} = \overline{D}(G)\backslash\underline{D}(G)$).

of the constraint are fixed (and are therefore denoted by lowercase letters). Since we assume that $D(T)$ contains exactly one graph and that $D(W[e])$ contains exactly one value for all e, we only need to filter the domain of the variable G. The following sections examine the cases where only T then both T and G need to be filtered. Non-fixed weights are handled in following sections.

After applying the filtering described in Section 5.3 for $\mathit{Subgraph}(T, G)$ and node-set equality, it remains to enforce that t is an MST of the value assigned to G. This means that we need to remove from $\overline{D}(G)$ any edge $e = \{u, v\}$ which is not in t and which is lighter than all edges on the path p in t between u and v; by the cycle property, the heaviest edge on the cycle $p \cup \{e\}$ (which is in t), cannot belong to any MST, so the cycle must not be in G. The only way to exclude the cycle is to remove $\{u, v\}$ from $\overline{D}(G)$. This can be done in linear time using King's algorithm [74], which receives a weighted tree and, in linear time, constructs a data structure that supports constant-time queries of the form "which is the heaviest edge $e*$ on the tree path between u and v?" If $w(e) < w(e*)$, e may not belong to G, remove it from $\overline{D}(G)$.

Example 5.8. In Figure 5.2, if $w(u) < w(a)$ or $w(u) < w(b)$ then u cannot be part of the graph or it would not have a, b, c, d as an MST. Similarly if $w(y) < w(c)$ or $w(y) < w(d)$ the constraint fails as this edge is already included in $\underline{D}(G)$.

5.5 *MST* with Non-Fixed Tree

We turn to the case $\mathit{MST}(g, T, w)$ where the variables G and W of the constraint are fixed. The tree T is constrained to be a minimum spanning tree of the given graph g and the bound-consistency problem consists in finding the union and the intersection of all MSTs of g.

5.5.1 Analysis of g to filter $D(T)$

We now describe Algorithm 4, a variant of Kruskal's algorithm [75] that constructs an MST of g while partitioning its edge-set into the sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$, defined as follows.

Definition 5.9. Let g be a connected graph. The sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$ contain, respectively, the edges that belong to all, some or none of the MSTs of g.

For an unconnected graph g whose maximal connected components are $g_1, \ldots, g_k$, we extend this definition to be the union of the respective set for each maximal connected component of g. Formally,

$$\begin{aligned}
Mandatory(g) &= \cup_{i=1}^{k} Mandatory(g_i),\\
Possible(g) &= \cup_{i=1}^{k} Possible(g_i),\\
Forbidden(g) &= \cup_{i=1}^{k} Forbidden(g_i),
\end{aligned}$$

As in Kruskal's original version, Algorithm 4 begins with a set of n singleton nodes and grows a forest by repeatedly inserting a minimum weight edge that does not create a cycle. The difference is that instead of considering one edge at a time, in each iteration we extract from the queue all edges of minimal weight, determine which of them are mandatory, possible or forbidden, and only then attempt to insert them into the forest. Let t_k be the forest constructed by using edges of weight less than k and let E_k be the set of edges of weight k. Let $\{u, v\} \in E_k$ and let $C(u)$ and $C(v)$ be the connected components in t_k of u and v, respectively. If $C(u) = C(v)$, then by the cycle property $\{u, v\}$ does not belong to any MST of g (i.e., $\{u, v\} \in Forbidden(g)$). If $C(u) \neq C(v)$ and $\{u, v\}$ is a bridge in $t_k \cup E_k$, then by the cut property $\{u, v\}$ belongs to all MSTs of g (i.e., $\{u, v\} \in Mandatory(g)$).

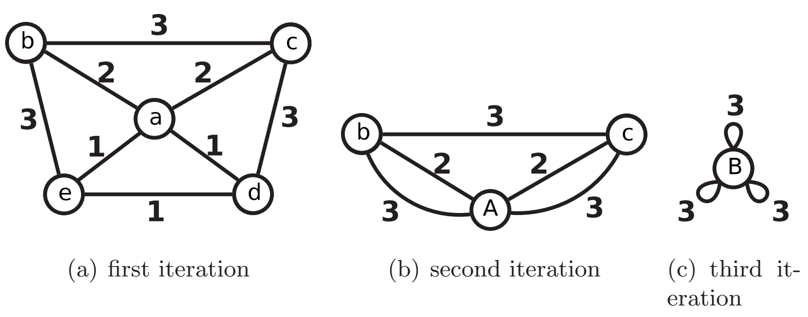
The running time of this algorithm is $O(Sort(m) + m\alpha(m, n))$ where $Sort(m)$ is the time required to sort the edges by weight and α is the inverse-Ackerman upper bound of the union-find disjoint-sets data structure [115] that represents the sets of nodes of the trees of the forest. When a batch of edges is extracted from the queue we need to perform a bridge computation in the graph composed of these edges to distinguish between possible and

```
Data: $G = (N, E)$ and $\underline{D}(T)$ two ground graphs
Result: $Forbidden$ and $Mandatory$: edges which belong to none and
        all of the MSTs of $G$
begin
    $P \longleftarrow Sort(E)$
    $UF \longleftarrow$partition with $N$ as singletons
    foreach $e = \{u, v\} \in Arcs(\underline{D}(T))$ do $Union(UF, u, v)$
    $Forbidden \longleftarrow \emptyset$
    $Mandatory \longleftarrow \emptyset$
    while $P \neq \emptyset$ do
        $E' \longleftarrow$pop minimum values from $P$
        $B \longleftarrow$ compute bridges in graph $(Cc(UF), E')$
        foreach $e = \{u, v\} \in E'$ do
            if $Find(UF, u) == Find(UF, v)$ then
                $Forbidden \longleftarrow Forbidden \cup \{e\}$
            else
                $Union(UF, u, v)$
                if $e \in B$ then
                    $Mandatory \longleftarrow Mandatory \cup \{e\}$
                end
            end
        end
    end
end
```

Algorithm 4: Partition Edges, fixed graph case

mandatory edges. Bridge detection takes time which is linear in the number of edges [114] and each edge of g participates in one bridge computation.

Example 5.10. Figure 5.3(a) presents a fixed graph on which we apply Algorithm 4. In the first step edges of weight 1 are popped. In the subgraph induced by nodes $\{a, d, e\}$, no edge is a bridge hence all edges are tagged *Possible*. The nodes $\{a, d, e\}$ are merged into node A giving the graph in Figure 5.3(b). Edges of weight 2 are popped, both are bridges and are tagged mandatory. Their endnodes are merged leaving only one node (Figure5.3(c)). All remaining weight 3 edges are loops and are forbidden.



(a) first iteration (b) second iteration (c) third iteration

Figure 5.3: Application of Algorithm 4 to classify the edges of the leftmost graph into mandatory (weight 2), possible (weight 1) and forbidden (weight 3) edges.

5.5.2 Filtering the Domain of T

We are now ready to use the results of the analysis of g to filter the domain of T. This entails the following steps: (1) For each mandatory edge $e \in Mandatory(g)$, place e in $\underline{D}(T)$. (2) For each forbidden edge $e \in Forbidden(g)$, remove e from $\overline{D}(T)$. Of course, if $\underline{D}(T) \not\subseteq \overline{D}(T)$ the constraint fails.

Since $Mandatory(g)$ and $Forbidden(g)$ are disjoint, the two steps have no effect on each other, so they may be applied in any order, and it suffices to apply each of them only once. But could we achieve more filtering by repeating the whole algorithm again, from the preprocessing step through the analysis of g to the filtering steps? We will now show that we cannot.

Let e be an edge that was placed in $\underline{D}(T)$ in the first filtering step. Then $e \in Mandatory(g)$, which means that it belongs to all MSTs of g. Let t_1

and t_2 be the two trees that e merges together when it is inserted into the forest by our variant of Kruskal's algorithm on g. Then the edges that were extracted from the queue before e do not contain an edge between t_1 and t_2, because in that case e would have been placed in either $Possible(g)$ or $Forbidden(g)$. This means that if e is in $\underline{D}(T)$ from the start, all edges lighter than e would be classified as before. Clearly, the edges heavier than e see the same partition of the graph into trees whether e is in $\underline{D}(T)$ or not. Since e is mandatory, the edges that have the same weight as e do not belong to a path between e's endpoints that uses only edges with weight at most equal to that of e. Hence, placing e in $\underline{D}(T)$ cannot change their classification.

Now, let e be an edge that was removed from $\overline{D}(T)$ in the second filtering step. Then $e \in Forbidden(g)$, which means that it does not belong to any MST of g. Then its removal from $\overline{D}(T)$ does not have any effect on the classification of other edges as $Forbidden$, $Possible$ or $Mandatory$.

We do not need to apply the filtering steps of section 5.3. The nodes of T are fixed so we cannot detect new cut nodes. Clearly a forbidden edge does not disconnect $\overline{D}(T)$ (otherwise it would be mandatory by definition). We show that the pruning of $\overline{D}(T)$ cannot creates new bridges as these bridges are already in $Mandatory(g)$. We consider a forbidden edge e is removed and creates a bridge in $\overline{D}(T)$. As it is forbidden, its endpoints are connected by a path composed of edges lighter than e. Then when the bridge e' of weight k was processed, the edge e was not in the graph g_k, and e' was a bridge in g_k. Hence it was classified as mandatory.

In conclusion, if we apply the algorithm again, the analysis of g would classify all edges in the same way as before. In other words, applying the algorithm again will not result in more filtering, the algorithm computes a fixed point and domains are bound consistent.

5.6 MST with Non-Fixed Graph and Tree

We now turn to the case $MST(G, T, w)$, in which both the graph and the tree are not fixed (but the edge weights are). Recall that we begin by applying the preprocessing step described in Section 5.3.

5.6.1 Analyzing $D(G)$ to filter $D(T)$

The main complication compared to the fixed-graph case is in the analysis of the set of graphs described by $D(G)$ in order to filter $D(T)$. We extend the definition of the sets $Mandatory$, $Possible$ and $Forbidden$ for a set of graphs as follows:

Definition 5.11. For a set S of graphs, the set $Mandatory(S)$ contains the edges that belong to every MST of any connected graph in S, the set $Forbidden(S)$ contains the edges that do not belong to any MST of a connected graph in S and the set $Possible(S)$ contains all other edges in the union of the graphs in S.

We need to identify the sets $Mandatory(D(G))$ and $Forbidden(D(G))$. We will show that it suffices to analyze the two bounds of the graph interval, namely the graphs $\underline{D}(G)$ and $\overline{D}(G)$.

Lemma 5.12 (Downgrade lemma)**.** *The addition of an edge to a graph can only downgrade the state of the other edges of this graph. Here, downgrading means staying the same or going from "mandatory" to "possible" to "forbidden". Formally: Let $g^* = g \cup \{e = \{u, v\}\}$ where $e \notin g$ and let $k = w(e)$. Then:*

$$\begin{aligned} \forall a \in g: \quad & (a \in Mandatory(g^*) \Rightarrow a \in Mandatory(g)) \wedge \\ & (a \in Possible(g^*) \Rightarrow a \in Mandatory(g) \cup Possible(g)) \end{aligned}$$

Proof. We compare the classification of the edges obtained by running algorithm 4 of section 5.5.1 twice in parallel, one copy called A running on the graph g and one copy called A^* running on g^*. Clearly, as long as the edge e is not popped from the queue, the edges are classified in the same way in both graphs.

If e is a forbidden edge of g^*, then the edges popped after it will still be classified in the same way in both graphs. Otherwise, it can affect the fate of the edges that are popped at the same time or later:
Case 1: If u and v belong to trees that are also merged by another edge e' with weight equal to that of e, then both e and e' are classified in g^* as possible, while in g the edge e' was classified as either possible or mandatory (depending on whether it was the only path connecting these trees in the batch). After this, the partition of the nodes of the graphs into trees is the same for g and g^*, and the algorithms classify the remaining edges in the same way.
Case 2: Otherwise, A leaves u and v in different trees while A^* merges the two trees. At some point, both algorithms see a batch whose insertion connects u and v in g. These edges will be classified by A as either possible

(if there is more than one) or mandatory (if there is only one). On the other hand, A^* will classify them as forbidden because their endpoints already belong to the same tree. In all other steps, both algorithms classify the edges in the same way. □

Example 5.13. We illustrate the previous proof by taking the graph of Figure 5.3(a) as g^*. If e is an edge of weight 1, we have Case 1: in g the edges are mandatory while in g^* they are possible. On the other hand if e is an edge of weight 2, we have Case 2, and its endpoints are not merged. An edge of weight 3 merges its endpoints and is mandatory in g while it is forbidden in g^*.

Note that the addition of a node of degree 1 and its incident edge to a graph does not change the status of the other edges in the graph. This edge just becomes mandatory.

We now show how to identify the sets $Forbidden(D(G))$ and $Mandatory(D(G))$. The set $Possible(D(G))$ consists of the remaining edges in $\overline{D}(G)$. Recall that the set $Forbidden(\underline{D}(G))$ is the union of the forbidden edges of each maximal connected component of $\underline{D}(G)$.

Theorem 5.14. *The set $Forbidden(D(G))$ of edges that do not belong to any MST of a connected graph in $D(G)$ is*

$$Forbidden(D(G)) = Forbidden(\underline{D}(G)).$$

Proof. A direct consequence of the downgrade lemma. □

We now turn to computing the $Mandatory$ set. The following lemma states that the mandatory edges belong to the mandatory set of $\underline{D}(G)$. Note that this does not follow from the definition of $Mandatory(G)$, because if $\underline{D}(G)$ is not connected, it does not belong to $D(G)$.

Lemma 5.15. *The set of edges that belong to all MSTs of all connected graphs in $D(G)$ is contained in the set of mandatory edges for $\underline{D}(G)$, i.e.,*

$$Mandatory(D(G)) \subseteq Mandatory(\underline{D}(G)).$$

Proof. If $\underline{D}(G)$ is empty, $Mandatory(D(G)) = \emptyset$. Otherwise, there are two cases: If $\underline{D}(G)$ is a connected graph, then it belongs to $D(G)$ and the lemma holds by definition. Otherwise, we may assume that $\overline{D}(G)$ is connected. This follows from the pruning rules of the connected constraint in the preprocessing step: Otherwise either the constraint has no solution or $\underline{D}(G)$ is empty or we can remove all but one connected component of $\overline{D}(G)$. Let $\overline{D}(G)'$ be the graph obtained from $\overline{D}(G)$ by contracting every connected component of $\underline{D}(G)$.

Let g be a minimal connected graph in $D(G)$, i.e., a graph in $D(G)$ such that the removal of any node or edge from g results in a graph which is either not connected or not in $D(G)$. Then g consists of the union of $\underline{D}(G)$ with a set t' of additional nodes and edges in $\overline{D}(G) \setminus \underline{D}(G)$. The set of mandatory edges of g, then, is the union of $Mandatory(\underline{D}(G))$ and the mandatory edges in t'.

Since we assume that the preprocessing step was performed, we know that every edge in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one connected graph in $D(G)$: Otherwise, it is a bridge in $\overline{D}(G)$ that connects two mandatory nodes and was not included into $\underline{D}(G)$ in the preprocessing step, a contradiction. We get that the intersection of the mandatory sets over all minimal connected graphs in $D(G)$ is equal to $Mandatory(\underline{D}(G))$.

Since every connected graph in $D(G)$ is a supergraph of at least one minimal connected graph of $D(G)$, we get by the downgrade lemma that the mandatory set for any graph is contained in the mandatory set for some minimal graph. This concludes the proof. □

Theorem 5.16. *The set $Mandatory(D(G))$ of edges that belong to all MSTs of graphs in $D(G)$ is:*

$$Mandatory(D(G)) = Mandatory(\overline{D}(G)) \cap Mandatory(\underline{D}(G))$$

Proof. If an edge is present in all MSTs of all connected graphs in $D(G)$ then it is present in all MSTs of $\overline{D}(G)$ and all MSTs of the minimal connected graphs of $D(G)$. Hence, by lemma 5.15, $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G)) \supseteq Mandatory(D(G))$. Assume that there exists an edge e in $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G))$ which is not in $Mandatory(D(G))$. Then there is a graph $g \in D(G)$ such that e is not in $Mandatory(g)$. Since $g \subset \overline{D}(G)$, we can obtain $\overline{D}(G)$ by a series of edge insertions. One of these insertions turned e from a non-mandatory edge into a mandatory one, in contradiction to the downgrade lemma. □

Example 5.17. Assume that the domain of G is the graph shown in Figure 5.4, where the solid edges are in $\underline{D}(G)$ and the dashed edge is in $\overline{D}(G) \setminus \underline{D}(G)$. Then $Mandatory(\overline{D}(G))$ contains the edges weighted 1 and 2, while the edge of weight 3 is forbidden. On the other hand, $Mandatory(\underline{D}(G))$ contains the edges weighted 2 and 3, because the edge of weight 1 is not in $\underline{D}(G)$. Hence, only the edge of weight 2 is mandatory in all graphs of $D(G)$.

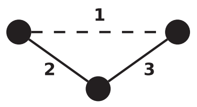


Figure 5.4: The domain of G in Example 5.17.

5.6.2 Filtering the Domains of G and T

Using the results of the previous sections, we derive a simple algorithm to filter the domains of T and G to bound consistency.

As before, we begin by applying the preprocessing step of Section 5.3. We then proceed as follows.

Step 1: We filter $D(T)$ according to $Mandatory(D(G))$ and $Forbidden(D(G))$ as in Section 5.5.2. By Theorems 5.14 and 5.16, we have these two sets if we know $Forbidden(\underline{D}(G))$, $Mandatory(\underline{D}(G))$ and $Mandatory(\overline{D}(G))$. They can be computed by applying the algorithm described in section 5.5.1 to both bounds of $D(G)$. Once these sets have been computed, we use them as in Section 5.5.2 to filter $D(T)$.

Step 2: To filter $D(G)$, we need to identify edges that cannot be in G because otherwise T would not be the minimum spanning tree of G. An edge $\{u, v\}$ has this property iff on every path P in $\overline{D}(T)$ between u and v there is an edge which is heavier than $\{u, v\}$.

Example 5.18. To illustrate this condition, assume that the domain of T is as described in Figure 5.5 and that $\overline{D}(G)$ contains the edge $\{u, v\}$ with weight 2. Although u and v are not connected in $\underline{D}(T)$, any path from u to v in a tree in $D(T)$ contains an edge which is heavier than $\{u, v\}$, so $\{u, v\}$ must not be in G. On the other hand, if the weight of the edge $\{x, y\}$ is 1, then $\{u, v\}$ can be in G if $\{x, y\} \in T$.

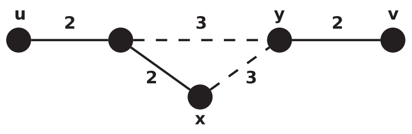


Figure 5.5: The domain of T in the example. Edges of $\underline{D}(T)$ and solid and those of $\overline{D}(T) \setminus \underline{D}(T)$ are dashed.

To find these edges, we apply Algorithm 5, a modified version of Algorithm 4 of Section 5.5.1 to $\overline{D}(G)$: We reverse the contraction of the edges of $\underline{D}(T)$ and create a sorted list of all edges of $\overline{D}(G)$, including those of $\underline{D}(T)$.

As before, we begin with a graph H that contains the nodes of $\overline{D}(G)$ as singletons and at each step we extract from the queue the batch B of all minimum-weight edges. We first insert the edges of $B \cap \overline{D}(T)$ into H and contract each of them by merging the connected components that its end-points belong to. Then, we remove from $\overline{D}(G)$ every edge in $B \setminus \overline{D}(T)$ that connects two different connected components of H; If any of these edges are in G, then at least one of them must belong to the MST of G. But they cannot be in T, so we must make sure that they are not in G either.

```
Data: $\overline{D}(G) = (N, E)$ a ground graph, $\overline{D}(T)$ ground graphs
Result: $prune$: edges which cannot be in $G$ if its MST is a
        subgraph of $\overline{D}(T)$
begin
    $P \longleftarrow Sort(E)$
    $UF \longleftarrow$partition with $N$ as singletons
    $prune \longleftarrow \emptyset$
    while $P \neq \emptyset$ do
        $E' \longleftarrow$pop minimum values from $P$
        $B \longleftarrow$ compute bridges in graph $H = (Cc(UF), E')$
        $E'_{in} \longleftarrow E' \cap Arcs(\overline{D}(T))$
        $E'_{out} \longleftarrow E' \setminus Arcs(\overline{D}(T))$
        foreach $e = \{u, v\} \in E'_{in}$ do
            if $Find(UF, u) != Find(UF, v)$ then
                $Union(UF, u, v)$
            end
        end
        foreach $e = \{u, v\} \in E'_{out}$ do
            if $Find(UF, u) \neq Find(UF, v)$ then
                $prune \longleftarrow prune \cup \{e\}$
            end
        end
    end
end
```

Algorithm 5: Filter $\overline{D}(G)$, non-fixed graph case, second step

After applying Step 2, we need to apply Step 1 again. However, we show that after doing so we have reached a fixed point. This is illustrated in figure 5.6. We first show that the second application of Step 1 does not change $\overline{D}(T)$. Since Step 2 depends only on $\overline{D}(T)$, we are done if we use the $\overline{D}(G)$ computed by the second step to update $\underline{D}(T)$.

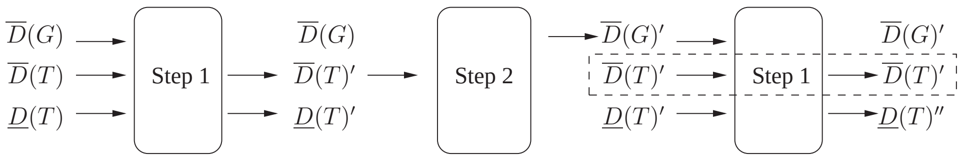


Figure 5.6: Dependency graph for the fixed point computation for the non-fixed tree and graph case. Fixed point is reached after an application of steps $1, 2, 1$ in that order. The figure is read from left to right. Primes and seconds indicate which bound can be modified by the pruning step, arrows indicate the input and output of each pruning step. The dashed box indicates what needs to be proved: the second application of Step 1 does not change $\overline{D}(T)'$. Therefore a subsequent application of Step 2 would again produce $\overline{D}(G)'$, the fixed point is reached.

Consider the impact of the removal of a non-tree edge e (i.e., an edge which is not in $\overline{D}(T)$) from $\overline{D}(G)$ during Step 1. The set $Forbidden$ is not affected because it depends only on $\underline{D}(G)$. Assume that the removal of e causes the insertion of an edge e' into $\underline{D}(G)$. Then e' was a bridge in $\overline{D}(G) \setminus \{e\}$. But since $e \notin \overline{D}(T)$, e' is a bridge in $\overline{D}(T)$ so it already was in $\underline{D}(T)$, and hence also in $\underline{D}(G)$, a contradiction. This proves that we have reached a fixed-point.

5.7 Handling Non-Fixed Weights

We now turn to the case where the edge weights are not fixed, i.e., the domain of each entry in W is an interval of numbers, specified by its endpoints. Once again, we begin with simple cases where some of the variables are fixed and gradually build up to the most general case.

5.7.1 Fixed Graph and Tree

When the graph and tree are fixed ($MST(g, t, W)$), the filtering task is to compute the minimum and maximum weight that each edge can have such that t is an MST of g[1]. A non-tree edge $\{u, v\}$ must not be lighter than any edge on the path in t between u and v. We can apply King's algorithm to

[1]This problem is tightly related to the *MST sensitivity analysis problem*, where we are given a graph with fixed edge weights and its MST and need to determine, for each edge, the amount by which its weight can be perturbed without changing the property that the tree is an MST of the graph.

t, while assuming that each tree edge e has the minimum possible weight $\underline{D}(W[e])$. Then, if the data structure returns the edge e' for the query $e = \{u, v\}$, we filter the domain of the weight of e, denoted $W[e]$, by setting $D(W[e]) \leftarrow D(W[e]) \cap [\underline{D}(W[e']), \infty]$.

For a tree edge e, let $t_1(e)$ and $t_2(e)$ be the two trees obtained by removing e from t. Then e must not be heavier than any other edge in g that connects a node from $t_1(e)$ and a node from $t_2(e)$. Let $r(e)$ be the minimum weight edge between $t_1(e)$ and $t_2(e)$ in the graph $g \setminus e$. Assuming that every non-tree edge has the maximal possible weight, we can find the $r(e)$'s for all tree edges within a total of $O(m\alpha(m, n))$ time [40, 116]. Then, for every $e \in t$ we set $D(W[e]) \leftarrow D(W[e]) \cap [-\infty, \overline{D}(W[r(e)])]$.

Now, if there is an entry $W[e]$ in the weights vector with $D(W[e]) = \emptyset$, e should be removed from g and the constraint has no solution.

5.7.2 Fixed Tree and Non-Fixed Graph

When the graph is not fixed but the tree is, the node-set of the graph is determined by the tree. After filtering $D(G)$ to equate the node-sets and to contain all edges of t, we have that the endpoints of all non-tree edges, i.e., edges of $\overline{D}(G) \setminus t$, belong to t.

We apply the filtering step of the previous section to the weights of the non-tree edges. If this results in $D(W[e]) = \emptyset$ for some edge e, we remove e from $\overline{D}(G)$ (if $e \in \underline{D}(G)$ then the constraint has no solution).

Next, we filter the weights of tree edges by applying the algorithm of the previous section on $\underline{D}(G)$. That is, for each tree edge e we find the weight of the lightest edge $r(e)$ in $\underline{D}(G)$ that connects $t_1(e)$ and $t_2(e)$, and shrink $D(W[e])$ as before. To see why it suffices to consider $\underline{D}(G)$, note that an edge e' in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one graph in $D(G)$, and in this graph, of course, e may be heavier than e'.

5.7.3 Fixed Graph and Non-Fixed Tree

In section 5.5, we handled the same problem with fixed weights by a variant of Kruskal's algorithm that required sorting the edges by weight. Since one weight interval can now overlap another, there is no longer a total order of the edge weights. In Algorithm 6, we will show how to adapt the Kruskal-based algorithm to this case.

Phase 1: First, Algorithm 6 considers edges in g. Instead of a list of edge-weights, we create a list of the endpoints of these edges' domains. We sort them in non-decreasing order, breaking ties in favor of lower bounds. Now, $\underline{D}(W[e])$ or $\overline{D}(W[e])$ is between $\underline{D}(W[e'])$ and $\overline{D}(W[e'])$ in the list if and only if $D(W[e]) \cap D(W[e']) \neq \emptyset$.

We then sweep over this list and examine each domain endpoint in turn. We say that an edge e is *unreached* before $\underline{D}(W[e])$ was processed, *open* if $\underline{D}(W[e])$ was already processed but $\overline{D}(W[e])$ was not, and *closed* after $\overline{D}(W[e])$ was processed. We maintain a graph H which is initially a set of n singleton nodes. In addition, we maintain a union-find disjoint-sets data structure UF that represents the connected components of the subgraph H' of H that contains only closed edges. Initially, the union-find data structure also has n singletons. During the sweep, when processing $\underline{D}(W[e])$ where $e = \{u, v\}$, we mark e as *open.* If $Find(UF, u) = Find(UF, v)$, we place e in the *forbidden set.* Otherwise, we insert it into H. When processing $\overline{D}(W[e])$ where $e = \{u, v\}$, we mark e as *closed.* If e is a bridge in H, we place it in the *mandatory set* Finally, we perform $Union(UF, u, v)$.

After the sweep, all edges of the mandatory set are included in $\underline{D}(T)$ and all of the forbidden edges are removed from $\overline{D}(T)$. Naturally, if this violates $\underline{D}(T) \subseteq \overline{D}(T)$, the constraint is inconsistent.

Phase 2: Next, the algorithm filters the weights of all edges. For a non-tree edge e, i.e., an edge $e \in g \setminus \overline{D}(T)$, the weight must be high enough so that e does not belong to any MST of g. In other words, the weight of e must be higher than the maximum weight of an edge on the tree path connecting its endpoints. In section 5.4 we mentioned that King's algorithm can find the desired threshold when the tree is fixed. But how do we find the MST in $D(T)$ that minimizes the weight of the heaviest edge on the path between u and v? Clearly, the desired weight is the lower bound of the domain of this edge's weight. The following lemma implies that it suffices to find any MST in $D(T)$ (while assuming that each edge has the minimum possible weight), and apply King's algorithm to this MST.

Lemma 5.19. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let u and v be two nodes in g, let p_1 be the path between u and v in t_1 and let p_2 be the path between u and v in t_2. Then*

$$\max_{e \in p_1} w(e) = \max_{e \in p_2} w(e)$$

Proof. Consider the symmetric difference $p_1 \oplus p_2$ of the two paths. It consists of a collection of simple cycles, each of which consists of a subpath of p_1 and

```
**Data**: $G = (N, E)$ and $\underline{D}(T)$ two ground graphs, $W$ defines $\overline{D}(W[e])$
        and $\underline{D}(W[e])$ for each edge $e$ in $E$
**Result**: $Forbidden$ and $Mandatory$: edges which belong to none and
          all of the MSTs of $\overline{D}(G)$
**begin**
    $P \longleftarrow \emptyset$  **foreach** $e \in E$ **do**
        Insert $(\underline{D}(W[e]),' <', e)$ in $P$
        Insert $(\overline{D}(W[e]),' >', e)$ in $P$
    **end**
    Sort $P$ in non-decreasing lexicographic order
    $UF \longleftarrow$partition with $N$ as singletons
    **foreach** $e = \{u, v\} \in Arcs(\underline{D}(T))$ **do** $Union(UF, u, v)$
    $H \longleftarrow$empty graph for incremental bridges
    $Forbidden \longleftarrow \emptyset$
    $Mandatory \longleftarrow \emptyset$
    **while** $P \neq \emptyset$ **do**
        $(w, b, e) \longleftarrow$pop minimum value from $P$
        $\{u, v\} \longleftarrow e$
        **if** $b =' <'$ **then**
            **if** $Find(UF, u) = Find(UF, v)$ **then**
                $Forbidden \longleftarrow Forbidden \cup \{e\}$
            **else**
                Insert $e$ into $H$
            **end**
        **else**
            /* $b =' >'$                                              */
            **if** *e is a bridge in H* **then**
                $Mandatory \longleftarrow Mandatory \cup \{e\}$
            **end**
            $Union(UF, u, v)$
        **end**
    **end**
**end**
```

Algorithm 6: Partition Edges, fixed graph, non-fixed weight case

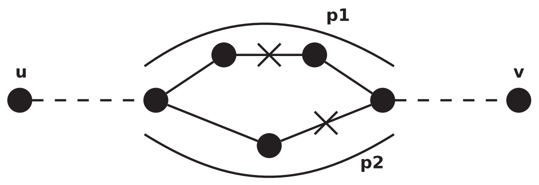


Figure 5.7: Illustration of the proof of lemma 5.19. A cycle and its p_1 and p_2 subpaths, the maximum weight edges excluded from each MST.

a subpath of p_2. Let c be one of these cycles. By the cycle property, any MST of g excludes a maximum weight edge from c (see Figure 5.7). Hence, there are at least two maximum weight edges in c, one in $c \cap p_1$ and one in $c \cap p_2$. Since this is true for every cycle, we get that the maximum weights on each of the paths must be equal. □

For an edge $e \in \underline{D}(T)$, i.e., an edge which belongs to all MSTs, the weight must not be so high that there is a cycle in g on which this is the heaviest edge. In other words, we need to find an MST t of g that contains $\underline{D}(T)$ and which maximizes the minimum weight of a non-tree edge r_e that together with t forms a cycle that contains e. In section 5.7.1 we mentioned that the desired threshold for all tree edges can be found in $O(m\alpha(m,n))$ time when the tree is fixed. Once again, we show that it suffices to apply the same algorithm to one of the MSTs in $D(T)$. Clearly, the desired weight is the upper bound of the domain of an edge weight. Furthermore, reducing the weight of any edge in g can only decrease the value of $w(r_e)$. The following lemma implies that it suffices to find any MST of g that contains $\underline{D}(T)$ (while assuming that each edge has the maximum possible weight), and use this MST to compute the thresholds for all tree edges.

Lemma 5.20. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let e be an edge in $t_1 \cap t_2$, let r_1 be the minimum weight edge that together with t_1 closes a cycle that contains e and let r_2 be the minimum weight edge that together with t_2 closes a cycle that contains e. Then*

$$w(r_1) = w(r_2)$$

Proof. It is known (see, e.g., Lemma 3 in [40]) that each of $t_1 \cup \{r_1\} \setminus \{e\}$ and $t_2 \cup \{r_2\} \setminus \{e\}$ is an MST of $g \setminus \{e\}$. Since all MSTs of a graph have equal weight, $w(r_1) = w(r_2)$. □

The time complexity of the algorithm described in this section is $O(Sort(m))$ to sort the endpoints of the domains of the edge weights and $O(m\alpha(m,n))$

for the modified Kruskal algorithm, the incremental connectivity [115] and bridge computations [125], two MST computations and the filtering of the edge weights.

5.7.4 General Case: Non-Fixed Graph and Non-Fixed Tree

We now turn to the most general case, in which all variables are not fixed. In the case of the *WBST* constraint, we were able to find efficient bound consistency algorithms for special cases, but the most general case is NP-hard. In contrast, we will show that the *MST* constraint is not NP-hard in its most general form. The naive bound consistency algorithm that we sketch in this section has a running time of $O(mn(m+\log n))$, which cannot be considered practical. We leave it as an open problem to find a more efficient method to approach the general case.

As before, we apply the filtering steps that follow from equality of the node-sets, inclusion of T in G and the connectedness of G and T. The main complication compared to the cases considered in the previous sections is in filtering the domains of the edge weights, because now the node-sets of the graph and the tree are not fixed. Again, we need to filter the lower bound weight of non-tree edges and the upper bound weight of tree edges.

Filtering the weights of non-tree edges. An edge $e = \{u, v\}$ in $\overline{D}(G) \setminus \overline{D}(T)$ cannot belong to the tree and therefore it must not be lighter than the maximum weight edge on the tree path from u to v. Let t be a tree and let $t(u, v)$ be the maximum weight edge on the path in t between u and v. We need to determine the minimum possible value of $t(u, v)$ over all trees in $D(T)$. Clearly, this weight would be a lower bound of the domain of some edge weight.

We set edge weights of all edge in $\overline{D}(T)$ to their lower bounds and contract the edges of $\underline{D}(T)$. In the remaining graph, we need to find a simple path from u to v that minimizes the maximum weight of an edge along it. This is done in $O(m + n \log n)$ time in Algorithm 7 with a dynamic programming approach that uses a variation of Djikstra's shortest-paths algorithm [32], where the sum of edge weights is replaced by a maximum computation. Since this computation needs to be repeated m times, once for every non-tree edge, the total time is $O(m(m + n \log n))$.

Filtering the weights of tree edges. A tree edge e must not be heavier than any non-tree edge that connects two nodes u and v such that e is on the tree path between u and v. To find the maximum possible weight of

```
Data: $G = (N, E)$ a ground graph, $e$ a (non-tree) edge, $w$ a weight
      for each $e$ in $E$
Result: the replacement weight of $e$
remove $e$ from $E$
$Q \longleftarrow$ Prio queue with all nodes from $N$ and $-\inf$ cost
$Threshold[u] = 0$
while $|Q| > 0$ do
    $S \longleftarrow ExtractMin(Q)$
    foreach neighbor $T$ of $S$ do
        $Threshold[T] \longleftarrow$
        $Min(Threshold[T], Max(Threshold[S], w(S,T)))$ if $T \in Q$
        then
            $DecreasePriority(Q, Threshold[T])$
        end
    end
    put $e$ back into $E$
    return $Threshold[v]$
end
```

Algorithm 7: Non-tree edge weight filtering in the general case

an edge $\{u, v\} \in \underline{D}(T)$, we will show how to check, for each edge $\{x, y\} \in \overline{D}(G) \setminus \underline{D}(T)$, whether there is a tree $t \in D(T)$ that contains $\{u, v\}$, such that $\{x, y\}$ is the minimum weight edge connecting the two components of $t \setminus \{\{u, v\}\}$.

Let $E' = (\underline{D}(T) \setminus \{\{u, v\}\}) \cup \{e | e \in \underline{D}(G) \cap \overline{D}(T) \wedge w(e) < w(x, y)\}$. If t exists, then each edge of E' is either in t or connects nodes that belong to the same connected component of $t \setminus \{\{u, v\}\}$. So we compute connected components of $G' = (Nodes(\overline{D}(G)), E')$. If x and y are in the same component, there is no such t. Otherwise, contract each connected component and merge the component of u with the component of v. Let G'' be the resulting graph. For a node $w \in G'$, we will refer to the node of G'' that represents the connected component of w by $CC(w)$. We will say that a node in G'' is mandatory if it represents a component in G' that contains at least one node from $\underline{D}(T)$.

Insert into G'' all the edges of $\overline{D}(T)$ which are not heavier than $\{x, y\}$. It remains to determine whether we can make a tree that spans $CC(x)$, $CC(y)$, $CC(u) = CC(v)$ and all the mandatory nodes of G'', such that $CC(u)$ is on the path from $CC(x)$ to $CC(y)$. To do this, root G'' at the $CC(u)$ and find its dominators (in linear time) [52]. If $CC(x)$ and $CC(y)$ have a common dominator, such a tree does not exist. Otherwise, find two disjoint paths, one from $CC(u)$ to $CC(x)$ and one from $CC(u)$ to $CC(y)$. Then add edges to the tree to make it span all mandatory nodes.

This test takes linear time, and needs to be repeated at most m times for every edge in $\underline{D}(T)$, i.e., $O(mn)$ times. The total running time is therefore $O(m^2n)$.

To sum up our results, we presented bound-consistent filtering algorithms for all cases of the MST constraint. For the special case where at least one of G, T or W is fixed, the algorithm is almost linear. In the general case, the algorithm is $O(m(m + n \log n))$.

5.8 Limitations of the $WBST$ Constraint

We establish two complexity results to justify our definition of the $WBST$ constraint as a bounded weight and not an exact weight and its limitation to graph intervals with a fixed node set. Let $EWST$ be the exact version of the $WBST$ constraint, i.e., the variant in which I is the exact weight of the spanning tree T.

Lemma 5.21. *It is NP-hard to check whether an* $EWST$ *constraint has a solution.*

Proof. By reduction from SUBSET-SUM, which is the following NP-hard problem [25]: Given a set S of integers and a target k, determine whether there is a subset of S that sums to k. We will assume that $k \neq 0$ (SUBSET-SUM remains NP-hard under this assumption).

Given an instance $(S = \{x_1, \ldots, x_n\}, k)$ of SUBSET-SUM, we construct a graph G_S that has a spanning tree of weight k iff S has a subset that sums to k (as shown in Figure 5.8). The graph has a node v_i for every element $x_i \in S$ plus two additional nodes, s and t. For each $1 \leq i \leq n$, the graph contains the edge $\{t, v_i\}$ with weight x_i. In addition, s is connected to every other node by an edge of weight 0. Clearly, for any spanning tree of the graph, the non-zero weight edges correspond to a subset of S that sums to the weight of the spanning tree. □

This justifies the definition of the parameter I of $WBST$ as an *upper bound* on the weight of the spanning tree. This constraint belongs to the category of optimization constraints [43]. Unfortunately, the following lemma states that $WBST$ is still an NP-hard constraint.

Lemma 5.22. *It is NP-hard to check whether a* $WBST$ *constraint has a solution.*

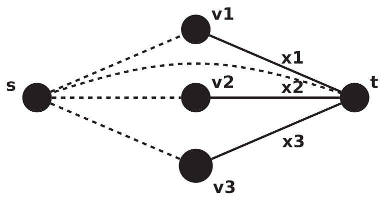


Figure 5.8: Graph for the proof of Lemma 5.21. The graph has a spanning tree of weight k iff the set $\{x_1, \ldots, x_k\}$ has a subset of weight k

Proof. By reduction from STEINER TREE, which is the following NP-hard problem [46]: Given an undirected graph $H = (V, E)$, a subset $U \subseteq V$ of the nodes and a parameter k determine whether the graph has a subtree of weight at most k that contains all nodes of U.

Given an instance of this problem, let $D(G) = D(T) = [U, H]$ and $D(I) = \{k\}$ (i.e., T must contain all nodes of U and may or may not contain the other nodes and edges of G). Then there is a solution to the STEINER TREE problem iff there is a solution to the constraint $WBST(G, T, I)$. $\square$

Together with Theorem 4.1, this implies that unless P=NP, we cannot find an efficient bound consistency algorithm for the $WBST$ constraint in its general form. In the following sections we develop an almost-linear bound consistency algorithm for the special case of $WBST$ in which the node-sets of G and T are fixed (but their edge-sets are not).

5.9 *WBST* with Fixed Graph and Edge Weights

To simplify the exposition of the algorithm, we begin with the special case $WBST(g, T, I, w)$ in which the variables G and W of the constraint are fixed (and are therefore denoted by lowercase letters). Since we assume that $D(G)$ contains exactly one graph and that $D(W[e])$ contains exactly one value for all e, we only need to filter the domains of T and I.

5.9.1 Analysis of g

Recall that we assume that the preprocessing step was performed and that $\underline{D}(T)$ is initially empty. Let t be an MST of g and let $w(t)$ be its weight. We will use t to partition the edges of g into three sets $Mandatory(g, \overline{D}(I))$, $Possible(g, \overline{D}(I))$ and $Forbidden(g, \overline{D}(I))$, which are defined as follows.

Definition 5.23. Let g be a connected graph. The sets $Mandatory(g, i)$, $Possible(g, i)$ and $Forbidden(g, i)$ contain, respectively, the edges that belong to all, some or none of the spanning trees of g with weight at most i.

Clearly, $Mandatory(g, \overline{D}(I)) \subseteq t$ and $Forbidden(g, \overline{D}(I)) \cap t = \emptyset$. Thus, we determine which of the tree edges are mandatory and which of the non-tree edges are forbidden. For the first task, we can use techniques that resemble those used in Eppstein's algorithm for finding the k smallest spanning trees of a graph [40]. This implies the following $O(m\alpha(m, n))$-time algorithm: Find an MST t of g and compute the replacements of every edge of g with respect to t. An edge $e \in t$ is in $Mandatory(g, \overline{D}(I))$ iff $w(t) - w(e) + w(r_g(e)) > \overline{D}(I)$, i.e., $g \setminus \{e\}$ has only spanning trees which are too heavy to be in the solution. An edge $e \notin t$ is in $Forbidden(g, \overline{D}(I))$ iff $w(t) + w(e) - w(r_g(e)) > \overline{D}(I)$. All other edges are in $Possible(g, \overline{D}(I))$.

5.9.2 Filtering the Domains of T and I

We are now ready to use the results of the analysis of g to filter the domain of T.

1. For each edge $e \in Mandatory(g, \overline{D}(I))$, place $e \in \underline{D}(T)$.
2. For each edge $e \in Forbidden(g, \overline{D}(I))$, remove e from $\overline{D}(T)$.
3. Since $\underline{D}(T)$ and $\overline{D}(T)$ may have changed, we need to re-apply the bound consistency algorithm for $SpanningTree(G, T)$ described in section 4.4.3.

As for $D(I)$, it is filtered by setting $D(I) \leftarrow D(I) \cap [\min(T), \infty]$, where $\min(T)$ is the minimum weight of a spanning tree of $\overline{D}(T)$ which contains $\underline{D}(T)$. In other words, the upper bound on the weight of the spanning tree is not changed by this filtering step. Note that repeating the algorithm again will not result in more filtering: Since $\overline{D}(I)$ did not change, the sets $Mandatory(g, \overline{D}(I))$ and $Forbidden(g, \overline{D}(I))$ are also unchanged.

5.10 *WBST* With Non-Fixed Tree and Graph

We now consider the case in which both G and T are not fixed (but the edge weights still are).

5.10.1 Filtering the Domain of G

In general, filtering the domain of G consists in adding nodes and edges to its lower bound and removing nodes and edges from its upper bound. In this case, we show that the filtering relating to nodes is done by our preprocessing step and that the filtering of edges is done trough the addition of edges in $\underline{D}(T)$.

As in the other cases, the filtering of nodes is done through the $Nodes(G) = Nodes(T)$ constraint. Obviously, adding edges (not nodes) to a graph will not hinder its capacity to contain a spanning tree of weight smaller than $\overline{D}(I)$. Finally, if an edge is mandatory in G for it to have such a spanning tree, it is obviously included in all such spanning trees.

Note this is true for all cases of the *WBST* and *MST* constraints.

5.10.2 Analysis of $D(G)$

The main complication compared to the fixed-graph case is in the analysis of the set of graphs described by $D(G)$ in order to filter $D(T)$. We generalize the definition of the sets $Mandatory$, $Possible$ and $Forbidden$:

Definition 5.24. For a set S of graphs, the set $Mandatory(S, i)$ contains the edges that belong to every spanning tree of weight at most i of any *connected* graph in S, the set $Forbidden(S, i)$ contains the edges that do not belong to any spanning tree of weight at most i of a *connected* graph in S and the set $Possible(S, i)$ contains all other edges appearing in at least one spanning tree of a *connected* graph in S.

We will show that it suffices to analyze the upper bound of the graph interval, namely the graph $\overline{D}(G)$. The following lemmas will be useful. Intuitively, they state that if an edge is removed from the graph, this can only decrease the number of weight-bounded spanning trees in the graph.

Lemma 5.25 (Monotony of the $Mandatory$ set)**.** *The removal of an edge from a graph cannot turn a mandatory edge into a possible or forbidden edge. Formally: Let g be a graph and $g' = g \setminus \{e\}$ the graph obtained by removing an edge $e = \{u, v\}$ from g. Then:*

$$\forall a \in g' : \qquad \big(a \in Mandatory(g, \overline{D}(I)) \Rightarrow a \in Mandatory(g', \overline{D}(I))\big)$$

Proof. Let t be an MST of g and let t' be an MST of g' such that if $e \notin t$ then $t' = t$ and otherwise, $t' = (t \setminus \{e\}) \cup \{r_g(e)\}$. Note that $w(t') \geq w(t)$.

Let a be an edge in $g' \cap Mandatory(g, \overline{D}(I))$. By the results of Section 5.9.1, this implies that $a \in t$ and $w(t) - w(a) + w(r_g(a)) > \overline{D}(I)$. The weight $w(r_g(a))$ of the replacement edge of a in g is not higher than the weight $w(r_{g'}(a))$ of its replacement edge in g'. Hence, we can conclude that $w(t') - w(a) + w(r_{g'}(a)) \geq w(t) - w(a) + w(r_g(a)) > \overline{D}(I)$, which means that $a \in Mandatory(g', \overline{D}(I))$. □

Lemma 5.26 (Monotony of the *Forbidden* set)**.** *The removal of an edge from a graph cannot turn a forbidden edge into a possible or mandatory edge. Formally: Let g be a graph and $g' = g \setminus \{e\}$ the graph obtained by removing an edge $e = \{u, v\}$ from g. Then:*

$$\forall a \in g' : \qquad \big(a \in Forbidden(g, \overline{D}(I)) \Rightarrow a \in Forbidden(g', \overline{D}(I))\big)$$

Proof. Let $w_{MST}(g)$ be the weight of an MST of G. By definition of $Forbidden$, $a \in Forbidden(g, \overline{D}(I))$ iff $w(a) + w_{MST}(g \setminus \{a\}) > \overline{D}(I)$. Similarly, $a \in Forbidden(g', \overline{D}(I))$ iff $w(a) + w_{MST}(g \setminus \{a, e\}) > \overline{D}(I)$. We show that removing an edge from a graph cannot decrease its MST weight: $w_{MST}(g) \leq w_{MST}(g \setminus \{e\})$. If the edge e is not part of the MST of g then both have the same weight. Otherwise $w(r_g(e)) \geq w(e)$ which means that the second MST will not be lighter. □

Corollary 5.27.
1. *The set $Forbidden(D(G), \overline{D}(I))$ of edges that do not belong to any spanning tree of weight at most $\overline{D}(I)$ of any connected graph in $D(G)$ is $Forbidden(D(G), \overline{D}(I)) = Forbidden(\overline{D}(G), \overline{D}(I))$.*

2. *The set $Mandatory(D(G), \overline{D}(I))$ of edges that belong to any spanning tree of weight at most $\overline{D}(I)$ of any connected graph in $D(G)$ is $Mandatory(D(G), \overline{D}(I)) = Mandatory(\overline{D}(G), \overline{D}(I))$.*

3. *The set $Possible(D(G), \overline{D}(I))$ consists of the remaining edges in $\overline{D}(G)$.*

Proof. A direct consequence of Lemmas 5.25 and 5.26. □

Filtering the Domains of T and I

The bound consistent algorithm begins by computing the sets $Mandatory(D(G), \overline{D}(I))$, $Possible(D(G), \overline{D}(I))$ and $Forbidden(D(G), \overline{D}(I))$. It then uses them to filter the domains of T and I in the same way as in Section 5.9.2, with a simple addition: Whenever an edge is placed in $\underline{D}(T)$, it is also placed in $\underline{D}(G)$.

5.11 *WBST* When All Variables Are Not Fixed

We now turn to the most general case, in which all four variables are not fixed. The weight of an edge e is now represented by the entry $W[e]$ of W, whose domain is an interval $[\underline{D}(W[e]), \overline{D}(W[e])]$. All steps that depend only on the topology of graphs (and do not involve edge weights) are unchanged. The other steps are modified as follows. Searching for a minimum spanning tree in a graph with interval edges amounts to finding an minimum spanning tree using the lower bound of the weights. In the preprocessing step, when contracting an edge $e \in \underline{D}(T)$, we subtract $\underline{D}(W[e])$ from the bounds of $D(I)$. When filtering G, we need to compute a minimum spanning tree and then search for the replacement edge for each MST edge. Here we need to minimize the weight of the spanning tree for all possible values of the weights so we assume that the weight of each edge e is $\underline{D}(W[e])$.

In the analysis of $\overline{D}(G)$, we again find an MST t using the minimum possible weight for each edge. When finding a replacement edge for a tree edge e, we search for the non-tree edge e' with minimal $\underline{D}(W[e'])$ and proceed as before (e is mandatory iff $w(t) - \underline{D}(W[e]) + \underline{D}(W[e']) > \overline{D}(I)$). When searching for a replacement edge for a non-tree edge e, we select the tree edge e' on the path between the endpoints of e with maximal $\underline{D}(W[e'])$. Then e is forbidden if $w(t) - \underline{D}(W[e]) + \underline{D}(W[e']) > \overline{D}(I)$).

Finally, we need to consider upper bounds on the weights of the tree edges (edges in $\underline{D}(T)$); the weight selected for them must not be so high that the total weight of the spanning tree is above $\overline{D}(I)$. We reverse the contraction of edges that were in $\underline{D}(T)$ in the input, and find a minimum spanning tree t of $\overline{D}(G)$ that contains $\underline{D}(T)$. For every edge in $e \in \underline{D}(T)$, we set $D(W[e]) \leftarrow D(W[e]) \cap [-\infty, \overline{D}(I) - w(t) + \underline{D}(W[e])]$.

In conclusion, we have shown the following

Theorem 5.28. *Let G and T be graph variables, let W be a weight function variable. If the node-sets of G and T are not fixed, it is NP-hard to filter $WBST(G, T, W)$ to bounds consistency. However, if the node-sets are fixed, there exists an algorithm that filters the constraint to bounds consistency in $O(m\alpha(m, n))$ time.*

5.12 Summary

We have shown that it is possible to compute bounds consistency for the *MST* constraint in polynomial time. For the special cases in which at least one of the variables is fixed, we found linear or almost-linear time algorithms. For the most general case, our upper bound is cubic and relies on a brute-force algorithm. It remains open whether the techniques we used for the simpler cases can be generalized to an efficient solution for *MST* in its most general form.

For the *WBST* constraint, we have shown that bound consistent filtering is NP-hard if the node-sets of G and T are not fixed. If they are fixed, we have shown that bound consistency can be computed in almost-linear time.

Chapter 6

A Gecode Implementation of CP(Graph)

This chapter presents an implementation of CP(Graph) in Gecode. Graph domains modeled as graph intervals were presented in chapter 3. Now, we describe their implementation in a practical framework. An emphasis is put on the integration of graph domains with the set and integer domains of Gecode and on the integration of CP(Graph) with the Boost graph library [17].

We describe the main concepts and features of Gecode in section 6.1. Then we give a short design rationale in section 6.2. In section 6.3, we describe the different operations which constitute the API of graph domains in CP(Graph). Section 6.4 describes the implementation of kernel constraints as propagators or views in Gecode. Sections 6.5 and 6.6 describe several implementations of graph domains either as their re-expression as set or integer domains or as a dedicated data structure. Section 6.7 describes features available in CP(Graph) to ease the implementation of filtering algorithms and search heuristics.

6.1 The Gecode Environment

Gecode stands for the Generic Constraint Development Environment. It is a C++ library for concurrent constraint programming which goal is to be easy to extend and efficient. The library offers features to ease the development of new search engines, search heuristics (labeling heuristics), new propagators and even new variable types. The architecture of Gecode is inspired from the architecture used for constraint programming in Mozart. Its architecture has been simplified and extended; it is described in [105, 107, 106]. Before we present the design and implementation of the CP(Graph) extension to Gecode, this section covers some important aspects of the Gecode library. This section can also be used as an introduction to the source code documentation.

6.1.1 Overview

Concurrent constraint programming is a model for solving CSPs where each constraint is associated to a *propagator*. Propagators are concurrent software entities which communicate through variable domains. A domain update performed by one propagator triggers other propagators which might imply more filtering. Propagators are run until they reach a fixed point where no additional execution of any propagator can result in more filtering. A solution to a CSP is found through the exploration of a search tree by alternating between phases of propagation and choice. A node of this search tree is called a *space*.

In Gecode, spaces, variables, and propagators are objects. The constraints are posted by instantiating an object for the propagator. This propagator knows on which variables it operates and has two main methods, "`propagate`" and "`cost`". A specific CSP is modeled by inheriting from "`Space`" and keeping references to the variables which are considered to constitute the solution.

The search tree is dynamically defined by *branching* strategies and it is explored by a *search engine*. When the propagators have reached a fixed point in a node of the search tree (the parent node), it is developed into its child nodes by adding a choice constraint into each child. For the search tree to be complete, the disjunction of the choice constraints should not further constrain the parent space. The choice constraint will hopefully cause some filtering and trigger other propagators. In Gecode, the search process is based on copy: On fixed point, the search engine asks the branching how many children it is going to produce for that space (`Branching::branch(Space *)`), then it can `clone` the parent space to keep it to produce the next children. The search engine asks the branching to post the appropriate constraint to turn the space into a specified child (`Branching::commit(Space*,int,BranchingDesc*)`). The search engine has a complete control on when, in which order and how many times it asks the branching to produce these children.

A search engine and one or more branching strategies are instantiated. The search engine triggers propagation in spaces then makes them branch.

Branching strategies are registered to the space and one is selected and run at each branch point in the search tree. The branching strategy queries the variables of the space to decide how many branches to create and which constraints to post in them.

Several techniques are used to speed up the search engines. To spare memory usage, recomputation is used. It consists in storing only some of the spaces along the path from the root. When a space is needed at a branching point, it is either already present in memory or can be recomputed by applying the appropriate constraints to an ancestor space stored higher in the tree. The search engine can query the branching strategy for an object called *branching description* which records the branching decision and is used for *batch recomputation*. Batch recomputation consists in posting all choice constraints on the search path from the stored space to the needed space at once and then compute only one fixed point. In Mozart, this batch recomputation is not available and a fixed point is recomputed at each branching point. If the space is recomputed from a distant parent, batch recomputation can lead to large speedups compared to simple recomputation [105].

6.1.2 Variables and Views

Views and Variables are object encapsulating a pointer to a variable implementation. They have the advantage that the user does not explicitly deal with variable pointers. They provide a different interface to the variable implementation: Variables are used when modeling a CSP and posting constraints. They allow to declare the initial domain, post constraints and query the state of the domain. On the other hand, views are used internally by propagators and branching strategies and provide an extended interface to perform basic tells on variable implementations.

An advantage of inserting a view between propagators and variable implementations is that multiple types of views can be created for the same variable type. For instance, with integer variables, offset and scaled views can be used. An offset view $X + k$ over variable X provides the same interface as a normal view over a variable Y which would have been created along with a propagator for $Y = X + k$. This feature allows to save propagators and additional variables for trivial constraints such as $X + k = Y$ or $a.X = Y$.

Example 6.1. Offset views allow to implement the queens problem with three alldifferent constraints, the classical one for keeping queens on different rows, and an additional alldifferent for each diagonal. These constraints are posted on offset views $X_i + i$ and $X_i - i$: $X_i + i \neq X_j + j \wedge X_i - i \neq X_j - j$ $(\forall i \neq j)$. The alldifferent propagator does not need to know whether it is dealing with offset or normal int views.

A potential pitfall is that a propagator must take care that fixed point reasoning copes with a potential sharing of variables in views. When several views used in a propagator are accessing the same underlying variable instance, changing the domain of one view affects the other view. This is described in the example below. If a propagator reports fixed point while it has not really achieved it, it might not be rescheduled and never detect an inconsistent partial assignment.

Example 6.2. Assume the constraint $X+1 \geq 2X$ is implemented by posting the constraint $Y \geq Z$ on an offset view $X+1$ and a scaled view $2X$. A propagator not dealing with sharing reaches a fixed point by pruning the lower bound of Y: $\underline{D}(Y) \leftarrow \max(\underline{D}(Y), \underline{D}(Z))$ and the upper bound of Z: $\overline{D}(Z) \leftarrow \min(\overline{D}(Z), \overline{D}(Y))$. With two different variables, a single application of these two rules reaches a fixed point. On the other hand, with our shared variable X, this is no longer the case. Let $D(X) = [1, 2, 3]$, $D(X+1) = [2, 3, 4]$ and $D(2X) = [2, 4, 6]$. The first rule does no pruning, the second one removes the value 6 from $D(2X)$. If those were different variables, we would be done but as the variable is shared the removal of 6 from $2X$ removes 4 from $X+1$. A fixed point computation instantiates the value of X to 1.

6.1.3 Iterators

Iterators are used by propagators to inspect a domain and to perform batch tells. The batch removal of values from an integer domain is one order of magnitude faster than n removals: In the worst case, a single removal needs to scan the whole domain. As the domain is a sorted list, removing a sorted list of values from it can be done in a single pass over the two lists.

Two types of iterators co-exist in Gecode: range and value iterators. Both of them share this part of their interface:

- `bool operator()(void)` is used as a "hasvalue" method.
- `void operator++(void)` is used as a "move to next" method.

Their difference lies in the type of values they iterate on:

- Range iterators provide `int min(void)` and `int max(void)` to access the current bounds of a range [?].
- Value iterators provide `int val(void)` to access the current value pointed by the iterator.

Range iterators are used for all tells and are the basic iterator type used to inspect domains. They are built with for instance: `LubRanges<SetView>::LubRanges(SetView& v)`. Such a range iterator can be converted to a value iterator by using a `Iter::Ranges::ToValues<It>` iterator built with: `ToValues<Iter>::ToValues(Iter & it)` where `it` is an instance of `LubRanges<SetView>` in this example.

Iterators can be combined using iterator composition such as union, difference or intersection. This provides a nice and efficient interface for computing the intersection and union of two bounds such as in the subset propagator. For instance: `Inter<It1,It2>::Inter(It1 &i1, It2 &i2)`

6.1.4 Scheduling of propagators

One asset of Gecode is the constraint propagator scheduling. The scheduler has one scheduling queue per variable type. In this sorted queue, the propagators are inserted according to their estimated cost. Lowest cost propagators are popped first, then higher cost propagators are processed only when no lower cost propagator is waiting to be processed.

This scheduling policy allows to use multiple propagators with different computational complexities at the same time for a single constraint. Cheap propagators are scheduled first and perform a rough pruning of the domains. Once these propagators have reached a fixed point, heavier but more precise propagators come into play.

Finally, Gecode has staged propagators [?]. These propagators combine the logic of several filtering algorithms of different complexities for the same constraint. The propagator chooses to run one of the filtering algorithms depending on the type of modification event that triggered it. This has the benefit that the propagator can detect entailment and disappear from the scheduling queues all at once while with several propagators, the others would need to be run for each of them to detect entailment.

6.1.5 Flow of information in the Gecode architecture

For the sake of documentation, we precise the flow of data and the responsibility of each entity in three important phases of a search.

Subscription When the user instantiates a propagator, this propagator calls the `subscribe` method of the views it is posted on. The variable stores a pointer to the propagator in a list and arranges for it to be scheduled (by setting its `pme` attribute). The `Propagator::Propagator(Space*, bool)` parent class constructor has a Boolean argument telling if the destructor of the propagator needs to be called when it is entailed (for instance if it has to free a data structure allocated on the heap).

Propagation When the search engine calls `Space::status()`, `Space::clone()` or `Space::commit()` (if no branching description is passed to it), the propagation is performed until a fixed point is reached. The scheduler calls the `propagate` method of a scheduled propagator. The propagator makes tells on variables. The variables notify the Space of any modification (`ModEvent`) that has occurred during the tell.The `VarTypeProcessor` is used to combine the modification events. The variable tell returns a ModEvent which the propagator takes into account. If this modification event is FAIL, the propagator must call the fail method of the space and exit immediately as the behavior of a space in this failed state is not guaranteed to be consistent.

Cloning When the search engine calls the `clone` method, the `copy` method of the Space is called (implemented in the user-defined derived class) this method calls the copy constructor of the derived class which calls the copy constructor of the base `Space`. All variables and propagators are copied by the `Space::Space(Space*,bool)` copy constructor. The original variables create temporary forwarding information: Each variable stores a temporary pointer to the corresponding new variable in the new space. The propagators are copied too and they call the `View::update` method on their views. That method uses the temporary forwarding information to update its encapsulated variable pointer. As the user-defined space derived class stores solution variables in data members, its copy constructor must also call the `View::update` method to point to the right underlying implementation. Finally, the `clone` method finishes by deleting the temporary information and restoring the original variables to their correct state.

6.2 Design of the CP(Graph) extension to Gecode

The main functional goals in the design of the CP(Graph) library consist of features to allow

- the implementation of graph domains,

- the implementation of graph constraints,
- the modeling and solving of subgraph extraction CSPs.

Non functional goals comprise extendability, maintainability and efficiency.

Most of the design of CP(Graph) is inherited from Gecode as it is an extension to that system. While it would have been possible to blend with Gecode in a minimalistic fashion, only obeying the mandatory rules, we tried to provide the same kind of tools that are provided in Gecode and to integrate with the whole framework as much as possible. For instance, we provide views and iterators in a way similar and interoperable with the rest of Gecode. The focus was put more on functionality than efficiency. The goal was to have first a system working with reasonable efficiency and only then optimize the bottlenecks.

To increase extendability and ease the prototyping of new graph propagators, we chose to also integrate CP(Graph) with a library of graph algorithms and data structures. We chose the Boost Graph library as it is free, offers a broad range of graph algorithms and is based on modern C++ design techniques called generic programming [111]. The Boost Graph library is used for the implementation of propagators and branching strategies. We also use the Standard Template Library (STL).

6.3 Graph View API

Graph domains were described in the previous chapter from a theoretical viewpoint. In its practical implementation, CP(Graph) provides several types of operations over graph domains which are described below. Unlike in the rest of Gecode, we did not define two different variable implementation wrappers (variables and views). We use views both for modeling and propagators. It is the user's responsibility to check modification events if basic tells are used when modeling. Note that variable wrappers could be introduced later ([105] introduces the idea of using views for modeling). Our implementation of graph views is described in sections 6.5 and 6.6.

6.3.1 Initialization

The graph domains are declared with an initial upper bound and an empty initial lower bound. One constructor of graph views takes the set of nodes and set of arcs as input:

- `GraphView::GraphView(Space* home,`
  `pair<vector<int>,vector<pair<int,int> > >& graph)`
  This method specifies the initial upper bound as a vector of node ids (`graph.first`) and a vector of pairs of node ids (`graph.second`).

- `GraphView::GraphView(Space* home, int numNodes)`
  This method specifies the initial upper bound as a complete graph over the node ids `0`,...,`numNodes-1`.

6.3.2 Domain inspection: Iterators

Graph domains provide iterators over the arcs and the nodes of the lower and the upper bounds of the graph domain. Two additional iterators on the unknown arcs and unknown nodes are also available. They traverse the set of elements which are present in the upper bound but not in the lower bound. The following methods are available for every graph view:

- `GraphView::LubNodeIterator GraphView::iter_nodes_UB(void)`
- `GraphView::LubArcIterator GraphView::iter_arcs_UB(void)`
- `GraphView::GlbNodeIterator GraphView::iter_nodes_LB(void)`
- `GraphView::GlbArcIterator GraphView::iter_arcs_LB(void)`
- `GraphView::UnkNodeIterator GraphView::iter_nodes_Unk(void)`
- `GraphView::UnkArcIterator GraphView::iter_arcs_Unk(void)`

These are Gecode value iterators, they obey the syntactic API of Gecode value iterators: `operator()`, `operator++` and `val()`. The latter method returns a node id (`int`) for node iterators and a pair of node ids (`pair<int,int>`) for arc iterators.

As the Gecode iterator combiners expect range iterators, we provide a `NodeRanges` version of the node value iterators (for instance `iter_node_ranges_UB`). This is a convenient syntactic sugar which allows to use range iterators without explicitly instantiating and converting a value iterator into a range iterator.

While in most cases, propagators dealing with arcs only focus on the node ids of the endpoints of each arc, some propagators might need the arc id instead. We provide an `ArcId<Iter,ArcNode>` iterator adapter which converts an iterator over node pairs to an iterator over arc ids (using the `ArcNode` template parameter to do the conversion, see also section 6.5).

These iterators have an API similar to the API of other iterators in Gecode. As we make an extensive use of STL and Boost containers and iterators in our design, we provide the `StlToGecodeValIterator` adapter which allows to handle STL iterators using the Gecode iterator API. In the STL, iterators are instantiated by pairs (most often using the `begin` and `end` methods of containers) and compared to test if the iterator is exhausted:
`for(it=c.begin(); it!=c.end(); ++it)`. Our iterator adapter is instantiated in the following way:
`StlToGecodeValIterator<Iter>::StlToGecodeValIterator(Iter begin, Iter end)`
A `StlToGecodeRangeIterator` is available too.

As with the integers and sets provided in Gecode, the use of iterators allows to efficiently perform multiple similar basic tells on the same bound. These tells are described in the next section.

As with sets, a graph domains carries information about the cardinality of its bounds. The last domain inspection methods deal with the order (number of nodes) and size (number of arcs) of the two bounds. It is possible to query the order and size of the current graph bounds in constant time:

- `int GraphView::GlbOrder(void)` returns the order of the lower bound graph of the graph domain.
- `Lub` and `Size` versions exist too.

Each graph domain also stores additional integer bounds for its order and size:

- `int GraphView::OrderMin(void)` returns the lower bound of the order of the graph domain.
- `Max` and `Size` versions exist too.

Note the following invariant for any graph view `g` : `g.OrderMin()`$\leq$`g.GlbOrder()`.

For propagators requiring more detailed inspection of graph domains (such as traversing the neighbors of a node), we provide another mechanism than iterators: *bounds graphs*. It is a customizable Boost graph data structure which reflects the current bounds of a graph domain. It allows to apply classical graph algorithms to the bounds or perform traversals. This data structure is documented in section 6.7.1.

6.3.3 Domain update: tells

In Gecode, modification events (`ModEvent`) are a mean to describe a modification of a domain. Three mandatory modification events are "*none*", "*assigned*" and "*failed*". In CP(Graph), the set of modification events has been kept as simple as possible. In addition to these three mandatory modification events (`ME_GRAPH_NONE`, `ME_GRAPH_VAL`, and `ME_GRAPH_FAILED`), we have events for the modification of the upper bound (`ME_GRAPH_LUB`), of the lower bound(`ME_GRAPH_GLB`) and of both bounds (`ME_GRAPH_BB`).

The propagation conditions (`PropCond`) have been defined accordingly. A propagator can be awakened when a graph variable is instantiated (`PC_GRAPH_VAL`), when its lower bound changes (`PC_GRAPH_GLB`), when its upper bound changes (`PC_GRAPH_LUB`) or when any modification happens (`PC_GRAPH_ANY`). These modification events are used when subscribing a propagator to a variable or canceling it.

The first basic tells allow to include one arc or node in the lower bound and exclude one from the upper bound:

- `ModEvent GraphView::arcIn(Space*,int u, int v)`
- `ModEvent GraphView::arcOut(Space*,int u, int v)`
- `ModEvent GraphView::nodeIn(Space*, int n)`
- `ModEvent GraphView::nodeOut(Space*, int n)`

They are there for convenience as iterator tells have the same complexity whenever used with a single value. When doing several similar tells on the same bound, the iterator tells should be used instead.

Other basic tells allow to update the bounds of cardinality:

- `ModEvent GraphView::OrderGq(Space*, int inf)`
- `ModEvent GraphView::OrderLq(Space*, int sup)`
- `ModEvent GraphView::SizeGq(Space*, int inf)`
- `ModEvent GraphView::SizeLq(Space*, int sup)`

Finally, as for other Gecode variables, we provide iterator tells which allow to perform several similar tells at once:

- `ModEvent GraphView::arcIn(Space*, Iter& i)`
- `ModEvent GraphView::arcOut(Space*, Iter& i)`
- `ModEvent GraphView::nodeIn(Space*, Iter& i)`
- `ModEvent GraphView::nodeOut(Space*, Iter& i)`

In these expressions, `Iter` is a template argument which should be instantiated to a Gecode value iterator. For tells on the nodes, the iterator value must be an integer, while for arc tells it must be a pair of integers.

6.4 Kernel Constraints

The three kernel constraints *Nodes*, *Arcs* and *ArcNode* were described in the previous chapter as the constraints which allow to link the graph domains with the sets and integer domains. We showed that using these constraints, it was possible to express our subgraph extraction problems with set and integer constraints. We also describe the adjacency constraint of graphs.

6.4.1 Constraint Propagators

The $ArcNode(A, N_1, N_2)$ constraint is implemented as explained in the previous chapter. The variables A, N_1 and N_2 are integer variables. The constraint is parametrized by an `ArcNode` mapping to convert pairs of node ids to arc ids and back: `arcnode(a,n1,n2,an)`.

The rules of the adjacency constraint `ArcImpliesNodes` were described in section 3.4. The graph view API provides the necessary methods to enforce this constraint. It is necessary to post it for some graph view implementations which do not guarantee that it is an invariant. The static `post` method is used by the views to post this constraint.

The $Arcs(G, SA)$ constraint is posted with `arcs(gv,sv,an);` where `an` is an `ArcNode` instance, `gv` is a graph view and `sv` is a set variable. The $Nodes(G, SN)$ constraint is posted with `nodes(gv,sv);` where `gv` is a graph view and `sv` is a set variable. These constraints are also available as views.

6.4.2 Constraints as Views

Thanks to the similarity of graphs and sets the $Nodes(G, SN)$ and $Arcs(G, SA)$ constraints are simple enough to also be expressed as set views over graph domains.

CP(Graph) provides a `NodeSetView` which is a set view adapter for graph views. It allows to plug a graph view in any constraint over sets. The constraint then applies to the set of nodes of the graph view. This is a cheap implementation of the *Node* kernel constraint. It does not require an additional set variable and propagator to maintain the coherence between these domains. A `NodeSetView` is simply instantiated by passing it a graph view: `NodeSetView<GraphView>::NodeSetView(GraphView & g)`.

Given the similarity between sets and graphs, the tells of the `SetView` API are easily translated to tells on the graph view. `exclude` and `include` correspond to `nodeOut` and `nodeIn`. `cardMin` and `cardMax` correspond to `OrderGq` and `OrderLq`. The last tell, `intersect` which lists values which can stay in the upper bound is converted to: `nodeOut` with a difference iterator between the nodes of the upper bound and the iterator passed to intersect.

A similar `ArcSetView` provides a view over the set of arcs of a graph variable and implements $Arcs(G, SA)$. A computation domain for pairs or tuples of integers domain is not available yet in Gecode. We chose to represent the arc type as integers, so sets of arcs are sets of integers. Arcs are mapped to non-negative integers using a user-defined mapping called the `ArcNode` mapping. For efficiency issues, there is however an invariant about arc ids that is assumed in the whole CP(Graph) system: the order of arc ids is the same as the lexical order of arcs (arbitrary gaps and offsets are allowed but shuffling is not). This constraint on the order of arc ids allows to have sequences of pairs of node ids and sequences of corresponding arc ids to be sorted accordingly. These sequences can then be used in iterator tells. CP(Graph) provides two predefined `ArcNode` mappings, the first one (`DefaultArcNode`) uses the index of the arc in the lexicographically ordered list of the arcs of a complete graph of given order. Example: in a graph with 12 nodes, arc $(1, 8)$ is numbered $12 \times 1 + 8 = 20$. The second one allows the user to specify the value of arc ids and stores the mapping in map data structures. By default, the first one is used.

6.5 Graph domains as views over set domains

We have just seen how a graph domain can be viewed as a set of nodes or arcs using `NodeSetView` and `ArcSetView`. These view adapter classes allow set propagators to transparently manipulate a graph variable as if it was the set of its nodes or the set of its arcs. In this section, we deal with the dual problem: can we model graph domains with sets. The answer is obviously yes. Three such views are provided in the version 1.0 of Gecode implementation of CP(Graph). They are described in the two following sections. The two first views are views for general graph domains. The last view corresponds to the single successor graph view.

6.5.1 The Arcs and Nodes Graph View

The `NodeArcSetsGraphView` is a translation of the definition of graphs: $G = (SN, SA)$. It relies on two set variables: one for the set of nodes and one for the set of arcs. The set of arcs is a set of non-negative integral arc ids. This view is implemented using the following template class parametrized by an optional `ArcNode` mapping:

```
template <class ArcNode=DefaultArcNode>
struct NodeArcSetsGraphViewT ;
```

Its constructors take an optional `ArcNode` instance used to translate end-node ids to/from arc ids. The default mapping is `DefaultArcNode` which numbers arcs by there lexical order in a complete graph.

```
NodeArcSetsGraphViewT(Space *home,
   const std::pair<std::vector<int>,
                   std::vector<std::pair<int,int> > >& graph,
   ArcNode *a=NULL);
NodeArcSetsGraphViewT(Space *home, int numNodes, ArcNode *a=NULL);
```

A constructor specific to this view allows to specify the sets to be used as node set and arc set, for this constructor the `ArcNode` mapping must be specified explicitly:

```
NodeArcSetsGraphViewT(Space *home, SetVar nodes, SetVar arcs, ArcNode *a);
```

The adjacency constraint is enforced by posting the `ArcImpliesEndNodes` propagator on the graph view. The propagation rules of this propagator were described in section 3.4. As this graph view is the only templated graph view, we define `NodeArcSetsGraphView` as the view using the default `ArcNode` mapping:

```
typedef NodeArcSetsGraphViewT<> NodeArcSetsGraphView;
```

As in the `NodeSetView` and `ArcSetView`, the conversion of tells and iterators from graph views to set views is straightforward. The size and order counts and bounds are present in the set domains so queries and tells about those are also directly translated. The complexity of iterating over the bounds is proportional to the size of the bound. Including or excluding one arc takes time linear in the size of the bound affected. When including a set of arcs or nodes, the overhead for seeking the element in the structure is proportional to the value of the element, seeking element 50 is on average twice as fast as seeking element 100. As nodes and arcs are required to be strictly ordered, this overhead can be spared when doing several modifications at once. Seek the first element, do the inclusion/exclusion, then from there seek the second element,. . .

This graph view has the advantage of being very simple by modeling a graph with only two sets in a way very similar to the mathematical definition of a graph. As demonstrated in chapter 7, it is however the less efficient model for a graph view.

6.5.2 The Out Neighbors Graph View

This second view (`OutAdjSetsGraphView`) uses one set variable for the set of nodes and one additional set variable for each node in the upper bound. Each of these additional sets model the set of "out neighbors" of each node. The "out neighbors" of a node n is the set of nodes n' which are the head of an arc (n, n') whose tail is n. These sets are stored in members `SetVar nodes` and `SetVarArray outN`.

The tells and iterators for nodes implemented in the same way as for `NodeArcSetsGraphView`. The arc iterator maintains the index and value of the current arc tail and uses an iterator over the set of adjacent nodes to get the arc heads. For tells and queries about the size of the graph, we add a integer variable member named `Size` with the following constraint:

$$Size = \sum_i \#OutNeigh(i)$$

The complexity of iteration over all the arcs is $O(m+n)$ where m and n are respectively the number of arcs and nodes of the iterated bound. For a batch tell, the inclusion or exclusion of several elements at once, the complexity is the same. However the seek time is reduced with respect to the nodes and arcs graph view: Seeking an arc takes $O(n + d)$ where d is the out-degree of the tail of the seeked arc. This is compared to $O(n + m)$ for the nodes and arcs graph view. As shown in section 7.4, in practice, arcs are included or excluded in small batches and this difference of asymptotic complexity really pays off.

The adjacency constraint is enforced by posting the `ArcImpliesEndNodes` propagator as in the previous view. Note that in this model, this adjacency constraint can be translated by introducing Boolean variables and posting a set of integer and set constraints:

$$\begin{aligned} &\forall n : OutNeigh(n) \neq \emptyset \Rightarrow n \in SN \\ &\forall n : OutNeigh(n) \subseteq SN \end{aligned}$$

It can be posted for all values n in the upper bound of the set of nodes SN using the Boolean negation, disjunction, reified constant in set, reified set cardinality and subset constraints. We show in chapter 7 that this reformulation is slightly less efficient than the `ArcImpliesEndNodes` generic propagator using the graph domain API.

This graph view as the advantage of allowing an easy formulation of some local properties of the graph:

- A node n is a sink : $OutNeigh(n) = \emptyset$
- A node n is a source: $\forall i \; OutNeigh(i) \neq \{n\}$
- The out degree of node n is 3: $\#OutNeigh(v) = 3$
- The graph is a matching: $Disjoint(OutNeigh) \wedge \forall i \; \#OutNeigh(i) \leq 1$

The `SetVar nodes` and `SetVarArray outN` public data members of the graph view can be used to model such constraints within Gecode.

In chapter 7, we show that this graph view is nearly as efficient as our dedicated graph domain implementation described in next section.

6.5.3 The Single Successor Graph View

This model, `SingleSuccGraphView`, is an integer model used for modeling paths [85, 58], cycles [6, 19] or trees [7]. Its particularity is that every node has a unique successor. It is a part of the model we presented in section 3.5.3 for the Knight's Tour problem, the variables modeling the next square of each square in the tour.

The tells and iterators are implemented in a way similar to the `OutAdjSetsGraphView`, the only difference lies in the underlying types for the iterators and the underlying methods for tells. The order and size of the graph are fixed as each node is included in the graph and has exactly one outgoing arc. The complexity of iterating over all nodes or arcs is linear in the size of the bound, $O(m + n)$. As each node has exactly one arc in the solution, it has at most one arc in the lower bound and the iteration complexity is bounded by $O(n)$ for the arcs of the lower bound. Note that the upper bound has no particular property an including or excluding one arc has complexity $O(n + d)$ has in the out neighbor sets view.

The adjacency constraint need not be enforced in this model as the node-set is constant.

6.6 A Dedicated Data Structure for Modeling Graph Domains

Instead of just translating graph domains into sets, we investigated whether it would be beneficial to have a dedicated data-structure which would allow a lower memory footprint and better timings for the operations available on graph domains.

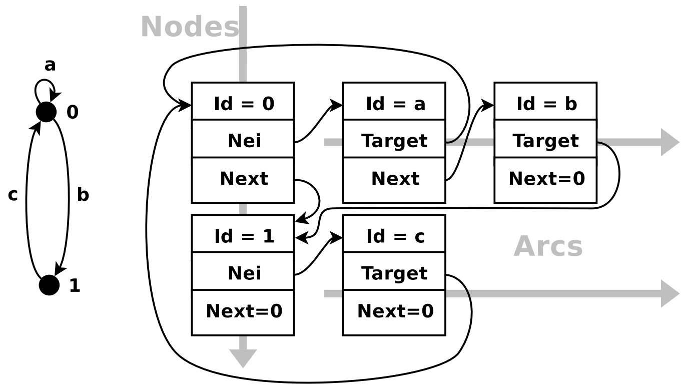


Figure 6.1: Graph data structure to represent the upper bound of a graph domain. Each of the nodes point to the list of their outward edges. The edges point to their head node (the tail is the node which point to the edge list).

6.6.1 Data Structure

Our data structure is an adjacency list. It is depicted in figure 6.1. Nodes are stored in a linked list (vertical in the figure). In each element of this list, we have a struct with an integer field storing the node id, a pointer to the next struct and a pointer to the out neighbors of that node. The structs in the out neighbors list have an integral id storing the edge id, a pointer to the next struct and a pointer to the head node. This structure is used to model the upper bound. The lower bound is represented by using the sign bit of the id of nodes and edges to indicate whenever it is also part of the lower bound.

Iteration over the nodes and arcs consists in following the pointers in the linked lists. A particularity of this model is that the iteration over the lower bound has the cost of iterating over the upper-bound. We can however stop before the end as we know the number of nodes and arcs of the lower bound. The worst-case complexity to access an arc (u, v) is $O(u + v)$.

The adjacency constraint need not be posted on this graph domain implementation as it is an invariant which is maintained by each tell operation and thus need not be enforced by a propagator. When nodes are removed from the upper bound they are marked and a complete pass over the upper bound is done to remove all incoming arcs to these nodes.

As is done for other variable types in Gecode, the implementation of the graph domain (`GraphVarImp`) is not exposed to the user which instead uses a graph view (`GraphVarView`) wrapping a pointer to this graph domain implementation.

6.6.2 Note about Memory Allocation

Gecode provides a memory allocator used to implement integer and set variable domains. These variables are implemented using linked lists of cells containing a range (pair of integers). The set domains use one such list for its upper bound and an other list for its lower bound. The cells are released by the variables when a domain modification results in less cells in the linked list (less gaps, when two intervals are merged into one). Those cells are then stored in a pool for later reuse. When a variable update needs to insert a cell into its linked list, it requests a free cell from the pool. If free cells are available, the operation does not need to perform a call to the underlying heap memory allocator. If not, then a large size of heap memory is requested at once. The allocator uses some strategy to dynamically determine the size of memory blocks allocated in the heap. The cell size of integer and set variables is computed for two integers and a pointer. That is 12 bytes on 32-bit machines and 16 on 64-bit machines. Our data structure uses cells of one integer and two pointers. That is 12 bytes on 32-bit machines and 20 bytes on 64 bits machines. On 32-bit architectures, our data-structure uses the same cell size as integer and set variables and can share memory cells with all the other variables in the CSP. The advantage of this sharing between graph and other domains could not be assessed by experiments. It seems that not calling the `dispose` method to give back the cells to the pool does not impede the running time or memory usage.

6.7 Additional support for branchings and propagators

This section covers three features of CP(Graph) which allow to ease the development of graph constraint propagators and search heuristics. The bounds graphs is the interface between graph domains and the Boost graph library. The second feature is a generic branching strategy which delegates the choice to a method of a bounds graph. Finally, we present scanners, a concise and efficient way to compare bounds of graph variables in propagators.

6.7.1 Bounds Graphs

One of the goals of CP(Graph) is to allow for the rapid development of new graph propagators. The `BoundsGraphs` class models both bounds of the graph domain and has a Boost Graph library compatible API. This API allows to apply the graph algorithms defined in Boost on the bounds of our graph views. They are the point where Gecode and Boost meet in our design. A `BoundsGraphs` instance has four data members: `UB` and `LB` are Boost graphs modeling the upper and lower bounds and `UB_v` and `LB_v` are mappings from node ids to Boost vertex descriptors used to access vertices of both Boost graphs. These mappings are currently implemented by vectors as node ids are kept consecutive in the universe graph of the CSP.

To suit the needs of all graph algorithms in the Boost library, we use bundled index properties in the vertex and edge objects implementing the Boost graph. These indexes allow to use vectors to implement additional properties of vertices and edges (such as a vertex color in DFS or a weight in Dijkstra's algorithm). The API provides members to reset node and arc indexes in each bound graph (*e.g.* `resetVertexIndexLB`).

The `update` member allows to keep the graphs synchronized with the graph view it operates on. The type of the underlying graph can be specified as a template parameter of the `BoundsGraphs` class. For instance, when information about incoming edges is as important as the information about outgoing edges, a bidirectional graph can be used. The latter is the default. If incoming edges and neighbors are not used, an unidirectional graph can be used. A undirected graph can also be used for non directed constraints.

This class also allows to modify the bound graphs during the propagation (contract edges, add temporary edges in the upper bound, ...). It allows to perform some modifications on the graphs and on the domain at the same time when these modifications are monotonic with respect to the domain. For instance, adding arcs to the upper bound cannot be performed on the domain, it can only be done in the Boost data structure.

The maintainability is increased as this class introduces one level of indirection between the propagator and the variable (like the views do) with an interface adapted to graph algorithms. The genericity of the Boost graph library allows to change the underlying graph implementation without changing the code using it. Given the broad range of algorithms available as part of this library, it is also much easier to prototype a propagator using the library than to write it from scratch.

6.7.2 Simple mechanism for branching strategies

CP(Graph) provides a simple schema to implement binary branching strategies based on one graph variable, the `UnaryGraphBranching` class. Two branching descriptions are provided: `GraphBDSingle` and `GraphBDMultiple`. The latter stores whether we branch on a node or edge, which node or edge we branch on and whether we should include or exclude this element in the graph domain on the left branch. The former is parametrized by the type (node or edge) about which we branch: `GraphBDSingle<int>` for nodes and `GraphBDSingle<pair<int,int> >` for edges. It uses slightly less space and should be preferred if the branching strategy only deals with one type of element of the graph. They are instantiated using:

- `GraphBDSingle<T>::GraphBDSingle(Branching*,T,bool)`
- `GraphBDMultiple::GraphBDMultiple(Branching*,pair<int,int>,bool)`

When instantiating a `GraphBDMultiple` for a node, the node id is passed in the first element of the pair.

A `UnaryGraphBranching` is used by defining a class deriving from `BoundsGraphs` and defining a `branch` method. This method uses the graphs stored in its members `UB` and `LB` to decide on which node or arc to branch next, allocates one branching description and returns a pointer to it. The branching is registered to the space by calling the following function from the space:

```
branch<GView, HeurBoundsGraph<GView> >(this,g);
```

If the user wants to develop other types of search strategies such as a strategy taking two graph variables into account or splitting the search space into more than two branches, it is possible to mimic the schema used in the `UnaryGraphBranching` or to fall back on the underlying mechanism in Gecode (the class `Branching`). One example is the `TernaryGraphBranching` developed by L. Quesada for his work on domReachability [92].

6.7.3 Scanners

Some constraint propagators for simple graph properties or relations simply need to compute the intersection, union and/or difference of two bounds of graph variables. To compute one of these, it is necessary to iterate over the two bounds. If several such combinations of bounds are needed they can be computed in a single pass instead of traversing the bounds several times. CP(Graph) provides a simple and efficient construct to scan all arcs of both graphs in one pass and collect the arcs which should be removed or added in the domains. We call that construct a scanner. The different scanners are used by instantiating a visitor which defines a method for each possible combination of status of the value with respect to both domains.

The "status" of an arc is a pair (s_1, s_2) where s_1 is the status of the arc in the first graph domain and s_2 the status of the arc in the second graph domain. As each arc might be (1) in ($\in \underline{D}(G)$), (2) out ($\notin \overline{D}(G)$) or (3) unknown ($\in \overline{D}(G) \setminus \underline{D}(G)$) in a domain, this gives nine possible status when dealing with two variables. The principle is then to collect in these methods the arcs (and/or nodes) which must be included or excluded from each graph domain. When the graph bounds are scanned, that info is used to prune the variables with iterator tells.

The first scanner has the following signature:

```
void scanTwoGraphsArcs(GV1 &g1, GV2 &g2, Visitor &visit)
```

It iterates on the union of the upper bounds of the two variables and calls one of eight methods for each arc according to the "status" of the arc in both domains. As we scan the union of the two upper bounds, the (out,out) status is never observed. In practice, for set based graph views, we need to instantiate an iterator for each of the four bounds and interleave the call to their "next" method. For the dedicated graph implementation it is simpler and more efficient to simply iterate on the upper bound and ask if the value is also in the lower bound as this information can be obtained in constant time. This scanner is used for the implementation of the induced subgraph constraint which was presented in chapter 3 and is detailed in chapter 4.

A second scanner is

```
void scanTwoGraphsCompleteNodeArcs (GV1 &g1, GV2 &g2,
                                    NodeValIter nVals, Visitor &visit);
```

It iterates on the Cartesian product of a set of nodes (`nVals`) with itself. In this scanner the nine methods of the visitor must be implemented as an arc can be absent from both domains. That scanner is used in the complement constraint.

Finally we also provide two simple scanners with only one graph view. `void scanGraphArcs(GV &g, Visitor &visit)` traverses the arcs of the upper bound of `g` and `void scanCompleteGraphNodeArcs(GV &g, Visitor &visit)` traverses the complete graph built over the set of nodes of the upper bound of `g`.

This technique has the advantage of providing a propagator implementation which is both efficient and declarative as the visitor only states the fate of elements based on their presence in the involved domains.

6.8 Conclusion

In this chapter we described the architecture of the Gecode library and its main features. We showed how graph domains can be implemented as an extension of this library, either by using set or integer variables as the underlying implementation or by using a dedicated data structure for the graph variables. We described how the kernel constraints are implemented as views and propagators. We also described some features such as several views and iterators which broaden the potential interactions between CP(Graph) and the rest of the Gecode library. Finally, we described some features to aid the development of new graph constraint propagators using the CP(Graph) extension to Gecode.

Chapter 7

Applications and Evaluation of the CP(Graph) Implementation

In this chapter, we present the application of CP(Graph) to a problem in biochemical networks analysis, the problem which triggered our research for a graph domain in CP. Then we present the combination of graph variables with map variables to model and solve graph pattern matching problems (published in [30]).

Then, we perform experiments to answer the following three questions. The first question asks whether graph-based CSPs should be translated to kernel graph constraints and other basic constraints or should we use global graph constraints. This question is answered for the *Connected* constraint presented in chapter 3. We show that the implementation of the global *Connected* constraint leads to faster and much more memory efficient programs. As in chapter 3, we also use this constraint in the context of the Knight's tour problem. We show that in that particular case a FD model using a stripped down FD model for connectedness is more running-time efficient than using the global path propagator. However, the memory usage is up to 20 times lower with the global constraint approach, allowing to solve larger problems if the memory is limited.

A second question concerns the relative efficiency of the proposed models for graph intervals. Memory and running times of different constraint programs are compared. The conclusion is that the graph variable type is slightly faster than the out-neighbors set model and can lead to a two-fold decrease of memory consumption. On the other hand, for some problems involving lots of small constraints, using a set model is more efficient. This limitation is addressed in the next question.

The last question asks for possible enhancements addressing the limitations of the graph variable type implementation: the practical running time incurred when accessing the first undetermined arc in the domain and the inadequacy of the chosen execution model with variables containing a large part of the CSP (we call it granularity of the variable). We show how arc descriptors allow to speed up some graph operations and discuss the addition of an update log to the graph variable implementation which would allow to apply sub-linear filtering algorithms for some constraints (almost linear in the number of changes instead of linear in the total size of the graph). This would adapt the execution model used for graph variables to a model similar to AC-2001 [15].

Appendix B, presents additional detailed information about each experiment.

7.1 A Metabolic Pathway Recovery Application

7.1.1 Metabolic Networks

The recent break-through of Systems Biology, the branch of Biology dedicated to the study of the molecular functioning of the organism as a whole, incurs a need for graph analysis methods and dedicated algorithms for Systems Biology.

Biochemical networks – networks composed of the building blocks of the cell and their interactions – are qualitative descriptions of the working of the cell. Such networks can be modeled as graphs. Metabolic networks are typical examples of such networks. They are composed of biochemical entities participating in reactions as substrates or products. Such a network can be modeled as a bipartite digraph whose nodes are the biochemical entities and reactions and whose edges link entities and reactions.

7.1.2 Pathways

Pathways are specific subsets of a metabolic network which are identified as functional processes of cells [81]. They can be used to study metabolic networks. As most pathways have a linear structure, one type of metabolic network analysis consists in finding simple paths in a metabolic graph [69, 73, 93, 118, 119].

The study of metabolic networks is constantly evolving and most of the problems are solved with dedicated algorithms. This dedicated approach has the benefit of yielding very efficient programs to solve network analysis problems. However its drawback is the difficulty of adapting a program to solve other problems or of combining programs to solve combinations of various analyses.

7.1.3 Applying CP(Graph) to Pathway Recovery

In [37, 35], we proposed to use constraint programming to solve constrained path finding problems in metabolic networks. The general kind of analysis we wish to perform with CP(Graph) is pathway recovery by constrained subgraph finding.

One potential application of this type of analysis lies in assisted explanation of DNA chip experiments. In such experiments, the behavior of a sane cell and a mutant are compared in a given context (the environment in which they live). This comparison is done at different times by extracting and amplifying the expressed RNA in the nucleus of the cells (this kills the cell). This RNA is then put on a DNA chip: an array of representative sequences of bases for a set of genes. The different RNA strands present in the cell bind to the chip in the locations which are specific to each of them. The micro array is scanned to measure the level of expression of each RNA strand which encodes for a given enzyme which in turn catalyzes a given set of reactions. The level of expression of RNA can be thresholded and interpreted in a binary decision as which reactions were active in the cell when its RNA was extracted. Given this set of reactions, biologists would like to know which processes were active in the cell.

If a program allows to recover known processes from subsets of their reactions, it could be adequate to discover the real processes given another set of reactions. Hence, such a program could approximate the real processes at work in a cell from DNA chip results. These computational results could then focus wet-lab experiments which are more expensive than in-silico experiments.

7.1.4 Previous work

Our current experiments focus on linear pathways by doing constrained shortest path finding.

In his thesis [27], Croes uses diverse constrained shortest path finding algorithm to find these linear pathways. Different problem formulations of point to point shortest path problems are compared and increasingly complex algorithms are used to solve them. A first setting consists in finding the point to point shortest path in a graph modeling the biochemical network; This can be done using breadth first search. A problem in this approach is that many shortest paths shortcut the real pathway by traversing pool metabolites, molecules like ATP or H_2O which are ubiquitous and are linked to many reactions. A proposed solution is to remove these nodes from the graph before performing the path finding.

A new problem is then that some pathways, such as glycolysis, use some of these metabolites as intermediates. In order to decrease the likelihood of selecting these nodes while still allowing to select them, each node is assigned a weight proportional to its degree. As pool metabolites have a very high degree, they are much less likely to be selected in the shortest paths. This problem requires a positive cost shortest path algorithm such as Dijkstra's. The rate of correct pathway recovery could be further increased: Some paths go through a reaction and its reverse reaction. This pair of reverse reactions models the reaction from substrates to products and from products to substrates. Most of the time, these reactions are observed in a single direction in each species. Hence paths containing both reactions should be excluded from the result set. The problem is extended to contain pairs of mutually exclusive reactions, and the algorithm needs to use backtracking to find the solution.

7.1.5 Our setting

The problems solved in [27] are point to point shortest path problems. We further extend this model by adding intermediate mandatory nodes in the path. The aim of this extension is to take DNA chip expression data into account. Remember this data is a level of expression for each gene at a given time in a complete cell and by thresholding this level, the data can be interpreted in terms of activated and inactivated genes. The genes encode for proteins which catalyze or inhibit reactions so the data can be interpreted in terms of mandatory reactions. If an information about forbidden nodes was extracted from the chip this would just reduce the size of the graph problem.

Let $n_1, ..., n_m$ the mandatory included reactions and $(r_{i1}, r_{i2}), 1 \leq i \leq t$ the mutually exclusive pairs of reactions, the CSP is: Minimize $Weight(G, w)$

s.t.

$$\begin{aligned} SubGraph(G, g) \wedge Path(G, n_1, n_m) \wedge \forall i \in [1, m] : n_i \in Nodes(G) \wedge \\ \forall i \in [1, t] : r_{i1} \notin Nodes(G) \vee r_{i2} \notin Nodes(G) \end{aligned} \tag{7.1}$$

Experiment 7.1. In our first experimental setting we run the path finding algorithm in metabolic graphs of increasing sizes. These graphs are computed by extracting a subgraph of the original metabolic bipartite digraph by incrementally growing a fringe starting by the included nodes while trying to respect the degree distribution. Then, we solve the constraint optimization problem presented above (7.1) using a first fail heuristic for path finding from [22]. This heuristic consists in selecting the node with lowest out-degree and choosing an outgoing arc which maximizes the in-degree of the target node. This problem is solved for the three longest linear metabolic pathways from [27]: lysine, glycolysis and heme. . All reaction nodes are selected as mandatory nodes. The weight is assigned to the nodes of the path: the weight of each node is its degree. Each reaction is doubled and its reverse reaction is mutually exclusive.

We performed this experiment with the cost-based filtering algorithm presented in section 4.6. The lower bounding works as follow: for each node n in $\underline{D}(G)$, compute the weight of the shortest path from n_1 to n_m going through n. The weight of each of these path is a lower bound of the weight of the solution, hence we constrain the solution to have a weight higher than the maximum computed weight.

The results are presented in Table 7.1, where the search is terminated after 10 minutes of computation. The growth of running time only allows to perform this experiment on small graphs. We are far from handling the metabolic graphs studied by bioinformaticians.

The problems with this heuristic are that it lacks a consideration for the costs and that the running time is greatly dependent on permutations of the node ids of the graph. This seems to indicate that the heuristic is not adapted to this problem.

We investigated the use of another heuristic, which would find shorter satisfying paths first and allow the lower bounding procedure to better prune to search tree. The very idea on which this shortest path metabolic analysis is based is that these pathways are close to shortest paths. Hence we developed a heuristic which explores that part of the network first. It uses the shortest path tree computed in the shorter path constraint propagator to select the edge extending the current path from the source towards the closest mandatory node along a shortest path.

	Time [ms]			Memory [KB]		
Order	Glyco	Heme	Lysine	Glyco	Heme	Lysine
50	0	50	-	6	10	-
100	80	1580	-	19	20	-
150	2500	4240	-	38	40	-
200	23520	-	-	64	-	-
250	-	-	-	-	-	-
300	-	-	-	-	-	-

Table 7.1: Use of a first fail TSPTW heuristic on the metabolic extraction problem

Pathway	Time [ms]	Mem [kB]	Commits	Failures
Glyco.	353230	53386	928	256
Lysine	82000	35655	201	54
Heme	97390	43849	253	66

Table 7.2: Results of experiment on the complete metabolic graph of the aMaze database. The running time, memory usage, size of the search tree and number of failures are presented.

Experiment 7.2. The shortest path tree based heuristic is used to solve the same problems as in experiment 7.1.

This heuristic has a tremendous impact on the computation time. The experiment results are presented in figure 7.1. This heuristic allows to find the constrained path in much larger graphs than the first fail heuristic. We present results for graphs up to 2000 nodes in that figure. The number of explored admissible solutions for these problems is very low (1 to 4). The number of failures is very low except for one small graph (order 150) and it corresponds to a peak in memory usage. In table 7.2, the running time, memory, number of explored spaces and number of failures are presented for the complete graph of 16316 nodes and 58114 edges. The running times begin to rise but the problem is solved in a reasonable time and amount of memory.

Experiment 7.3. The shortest path tree based heuristic is used to solve the same problems as in experiment 7.1 except that instead of selecting all reactions we alternatively select one reaction out of two consecutive reactions. Hence, the number of mandatory nodes is approximately half of that of experiment 7.2.

The running times for experiment 7.3 are presented in figure 7.2. The running times are slightly higher for glycolysis and heme but slightly lower for lysine. These results can be considered of the same order except for one instance of glycolysis in the graph of order 150 which cannot be solved in less than 10 minutes in experiment 7.3.

7.1.6 Conclusion

We note that the Gecode + CP(Graph) approach allows to handle reasonably large graphs (15946 nodes and 46430 edges) provided an appropriate heuristic and lower bounding procedure is used. Another important point is that the open-source nature of Gecode and CP(Graph) allows to easily share information between propagators and heuristics, something which would not be that easy when using a commercial solver in which propagators are black boxes.

Future work comprise two main aspects. The first aspect consists in finding which additional constraints are needed to recover known pathways as it was shown in [27] that non-constrained shortest paths are not able to recover all of them. A second aspect is an extension of this approach for discovering pathways containing branchings or cycles. A first formulation which would be interesting to test is the following: find the smallest graph containing all the seeds such that there is a seed from which all other nodes are reachable.

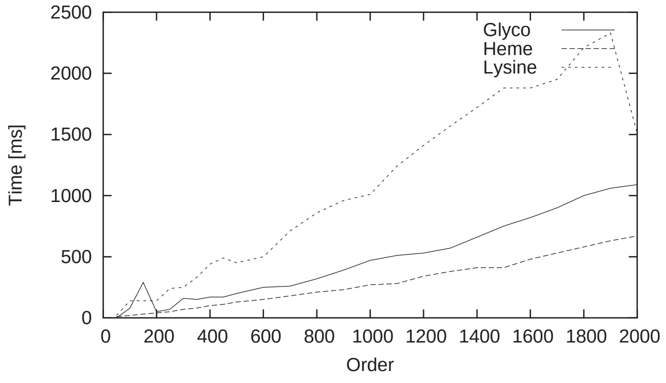
Given sn_s a set of nodes (seeds), minimize $Weight(G, w)$ subject to:

$$N \in sn_s \wedge sn_s \subseteq Nodes(G) \wedge Reachable(G, N, Nodes(G))$$

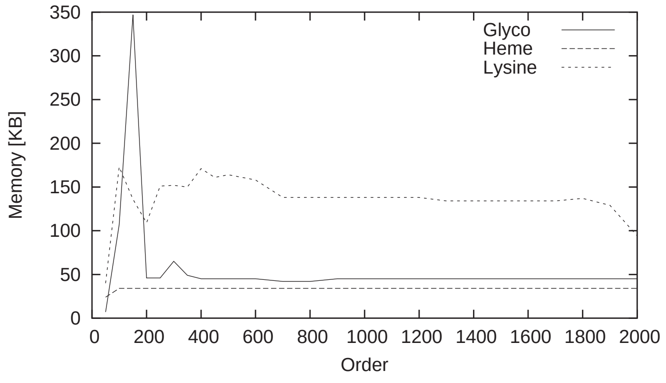
A second formulation in an undirected version of the metabolic graph could be the Steiner tree problem: find the lightest tree connecting the nodes of sn_s. Finally, the minimum weight induced subgraph (with mandatory nodes) has also an interest if edges are assigned a meaningful weight.

7.2 Modeling Graph Pattern Matching Problems

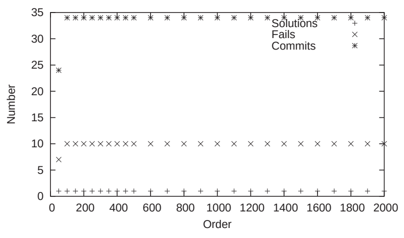
Graph pattern matching is a general term for problems where a graph p called the pattern and a graph t called the target are to be aligned or compared. These problems can be classified along three axes: monomorphism versus isomorphism, graph versus subgraph and exact versus approximate. Graph monomorphism is a problem where one wants to find a bijection m from the nodes of p to the nodes of t such that $(u, v) \in p \Rightarrow (m(u), m(v)) \in t$.



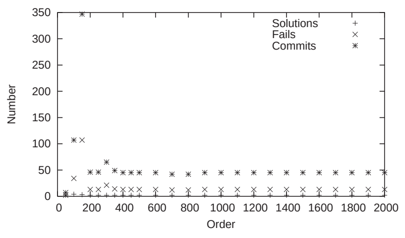
(a) Running times



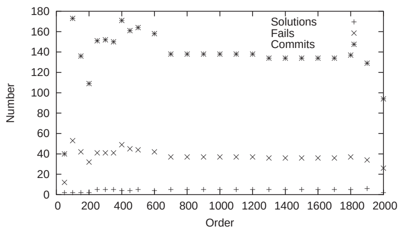
(b) Memory



(c) Statistics for Heme

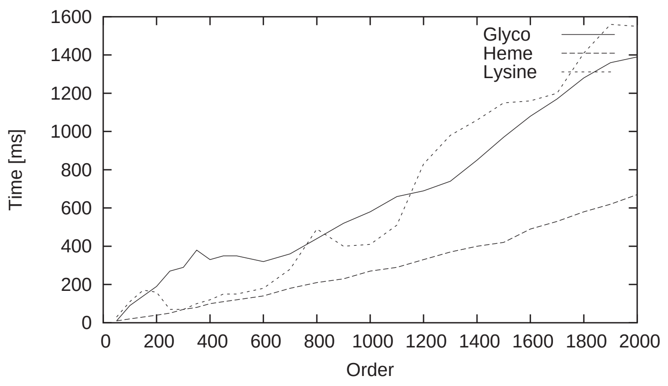


(d) Statistics for Glycolysis

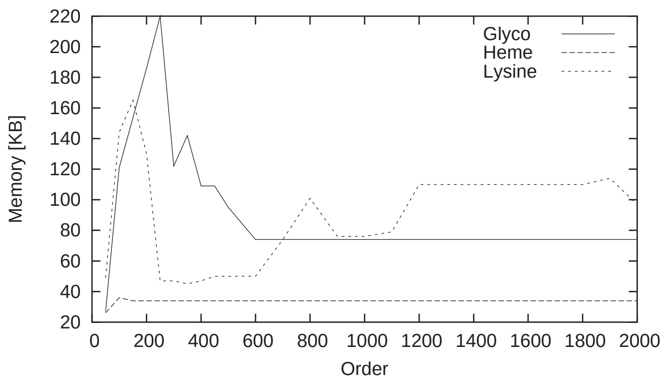


(e) Statistics for Lysine

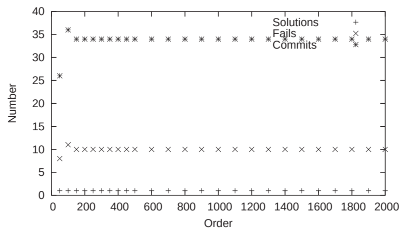
Figure 7.1: Use of a shortest path heuristic for pathway extraction in metabolic graphs



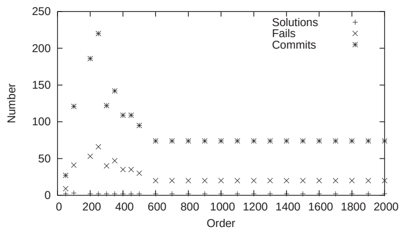
(a) Running times



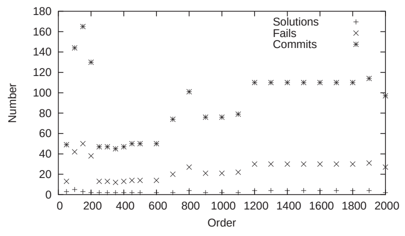
(b) Memory



(c) Statistics for Heme



(d) Statistics for Glycolysis



(e) Statistics for Lysine

Figure 7.2: Use of a shortest path heuristic for pathway extraction in metabolic graphs. Using half of the mandatory nodes of figure 7.1.

In graph isomorphism, m must satisfy $(u, v) \in p \Leftrightarrow (m(u), m(v)) \in t$ where the implication of monomorphism is replaced by an equivalence. In graph (iso/mono)-morphism each node of p must be associated with exactly one node of t and reciprocally. In subgraph (iso/mono)-morphism only a subgraph of t must be matched, *i.e.* some nodes of t might be unmatched, m is an injective function. The previous problems are exact pattern matching problems. The approach taken for approximate pattern matching in [126] was to specify optional parts in the pattern graph p, these parts need not be matched to the target graph t. In [30], Deville et al. introduced map variables and modeled these problems using the $Mono(P, T, M)$ constraint which holds if the map variable M is a bijection modeling a monomorphism from P to T.

One elegant feature of the modeling of these pattern matching problems with graph variables is that the opposition exact/approximate can be modeled by switching from fixed graph to non-fixed graph for the pattern graph variable:

- $Mono(p, t, M)$ models *exact* monomorphism of p to t.
- $Mono(P, t, M)$ models *approximate* monomorphism of $\overline{D}(P)$ to t (with optional part $\overline{D}(P) \setminus \underline{D}(P)$).

Similarly, the opposition graph/subgraph in these problems can be modeled by using a fixed or non-fixed graph variable for the target graph.

- $Mono(p, t, M)$ models exact *graph* monomorphism of p to t.
- $Mono(p, T, M)$ models exact *subgraph* monomorphism of p to $\overline{D}(T)$.

Additionally, $Mono(p, t, M) \wedge Mono(Compl(p), Compl(t), M)$ – where $Compl(G)$ is the complement graph of G – models an isomorphism from p to t.

Propagators for these problems and constraints have been implemented by S. Zampelli for his doctoral research. In [126], these models allow to solve more graph and subgraph pattern matching problems in a limit of 5 min per instance than when solved with a state of the art solver (`vflib`).

Future work includes breaking symmetries using automorphisms and an approach to the modeling of the maximum common subgraph problem using $Mono(P, T, M)$. This problem can be solved with two non-fixed graph variables by searching for a solution maximizing the size of the graphs. Also, richer patterns can be expressed in approximate pattern matching by imposing more constraints on the pattern. In the current definition of the approximation, P is constrained to be an induced subgraph of the initial pattern. Other constraints such as connected or tree could be investigated.

7.3 Comparison of $Connected(G)$ Implementations

In this section we compare an approach based on graph variables and global constraints with an approach based on integer variables and finite domain constraints for two problems: the connected subgraph and knight's tour problems presented first in section 3.5.3.

We show that in both cases, the global constraint approach is obviously much simpler to model than with basic constraints. We also show that the FD models produced by a generic constraint decomposition are by far outperformed by the global constraint approach (especially regarding memory consumption). Then we show that for larger problems such as the Knight's tour, while the global constraint approach performs better than a generic decomposition, a hand crafted model leads to a faster program. However this faster model still consumes about 20 times more memory which limits the size of problems which can be modeled.

7.3.1 The Generic $Connected(G)$ Decomposition

In chapter 3, we showed how the connected constraint can be modeled by computing the transitive closure of the graph using Boolean variables and constraints. The purpose of this section is to give an experimental comparison of that model with the *Connected* constraint we presented in Chapter 4.

Experiment 7.4. The first model compared in this experiment is the Boolean CSP presented in section 3.5.3 which uses Boolean variables and constraints. The second model uses a graph variable and a relaxed global connected constraint which performs only upper bound pruning in order to obtain the same level of consistency hence the same search trees.

The results are presented in Table 7.3. We use an adaptive recomputation distance of 2 and no fixed recomputation with a naive heuristic (include/exclude first unknown arc).

The running times obtained in both models are comparable for small graphs. The memory used by the Boolean model grows in $O(n^4 \log n)$ while it grows in $O(n^2 \log n)$ with the graph model. As soon as the graphs hit a certain size, the CP(Graph) model is much faster (graphs of order 20 lead to a two-fold speed-up). Moreover, the memory explosion strictly limits the

	Time [ms]		Memory [KB]	
n	Graph	Bool	Graph	Bool
All solutions				
3	0	0	5	30
4	0	0	6	98
5	60	90	7	267
6	2630	3790	11	771
7	219740	300850	15	1732
1000 solutions				
5	60	80	7	267
10	220	280	37	4420
15	470	990	98	39948
20	860	3140	215	207076
25	1350	10030	438	763629
30	1980	-	852	-
40	3940	-	2905	-
50	7100	-	5787	-
100	52960	-	85289	-

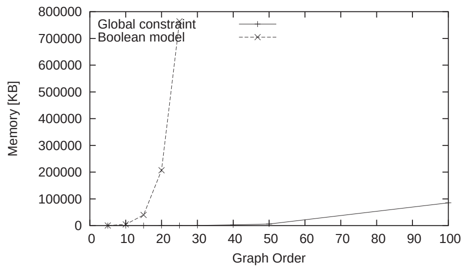
Table 7.3: Experiment 7.4: Comparison of the connected global constraint with the Boolean model for this constraint. "-" means the FD model ran out of memory.

use of the Boolean model to graphs of order less than 50: With a complete graph of fifty nodes, a single computation space takes up 1.4 GB of memory for enforcing connectedness using a Boolean model. Compared with the 120 KB used by the CP(Graph) model, this amounts to a 10^4 ratio.

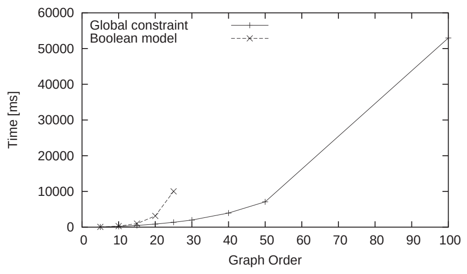
7.3.2 A Specific Decomposition for the Knight's Tour

We already mentioned the knight's tour in chapter 3. The knight's tour can be viewed as a graph problem and has been efficiently modeled using finite domains [120]. A knight's tour of a chess board is an Hamiltonian path in the graph of knight's moves, the tour is said to be reentrant if the end position is one knight's move away from the start position. In this case, both ends of the Hamiltonian path can be joined by an additional arc to form a cycle. This problem can be solved in linear time [24] and a finite model for solving it with constraints is part of the examples of Gecode.

We showed in chapter 3 that a path constraint can be decomposed into some constraints related to the degree of each node and a constraint to



(a) Memory



(b) Running times

Figure 7.3: Experiment 7.4: Plot of the data in table 7.3, search for 1000 solutions.

avoid cycles or ensure that the graph is connected (these are equivalent in this case). We showed that in the case of the Hamiltonian path it was easy to use the structure of the problem to come up with a very compact model for connectedness (line 3 of the FD model in experiment 7.5. In this section we show experimentally that this FD model for the Knight's tour leads faster to a first solution than the CP(Graph) model using a global path constraint. However, the former model uses about 20 times more memory than the CP(Graph) model.

Experiment 7.5. We compare two models for the Knight's tour problem:

- the CP(Graph) model:

$$Subgraph(G, KG_{s,s}) \wedge Nodes(KG_{s,s}) = Nodes(G) \wedge Path(G, n_{0,0}, n_{1,2})$$

- the FD model:

(1) $\forall n \in [0, s^2 - 1] : pred_n \in Neigh(n, KG_{s,s}) \wedge$
$succ_n \in Neigh(n, KG_{s,s}) \wedge jump_n \in [0, s^2 - 1] \wedge pred_n \neq succ_n$

(2) $\forall (u, v) \in Arcs(KG_{s,s}) : succ_u = v \Leftrightarrow pred_v = u \Leftrightarrow$
$jump_v = jump_u + 1$

(3) $\forall n, n' \in [0, s^2 - 1] : jump_n \neq jump_{n'}$

(4) $jump_0 = 0 \wedge jump_{2s+1} = 1$

These models are run with values of s in $\{10, 20, 30, 40\}$. One solution is sought with a naive heuristic. The recomputation distances are fixed:25 and adaptive:5.

The running times for the programs using a graph variable and a path constraint as well as for the FD model are given in table 7.4 and figure 7.4. We note that the very simple CP(Graph) model is able to solve the problem using no particular heuristic (the first available arc is included on the left branch and excluded otherwise).

The results obtained for the FD model with the same heuristic and search parameters as the graph model are also presented in table 7.4 (columns FD-AC and FD-FC). They show that this model uses much more memory but is much faster than the graph model when forward checking (FC) is used. Forward checking is a relaxed version of arc consistency where each value of each domain must be consistent with the variables already assigned (instead of all variables). When arc consistency is used in this model, it performs

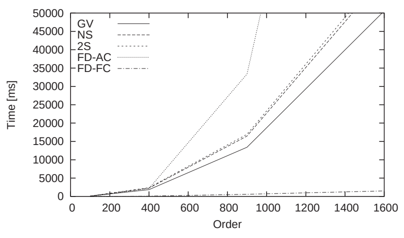
Order	Time [ms]				
	GV	NS	2S	FD-AC	FD-FC
100	90	100	110	10	10
400	1,850	2,260	2,390	2,180	90
900	13,410	16,460	16,950	33,380	580
1600	50,590	60,090	61,570	199,390	1,500
Order	Memory [KB]				
	GV	NS	2S	FD-AC	FD-FC
100	58	144	58	996	996
400	545	1,410	649	10,193	10,193
900	3,626	9,739	4,618	68,974	68,974
1600	10,451	25,157	11,347	188,947	188,947

Table 7.4: Comparison of the graph variable model and the FD model for the closed knight's tour. The first column, Order, is the number of squares on the board, the length of the board to the square. The three next columns are the three graph views: graph variable type (GV), out neighbors sets (NS), two sets (2S), the column FD-AC is the finite domain model with arc-consistency and the FD-AC column is with forward checking. Search parameters: defaults of the FD model, fixed recomputation distance: 25, adaptive recomputation distance: 8.

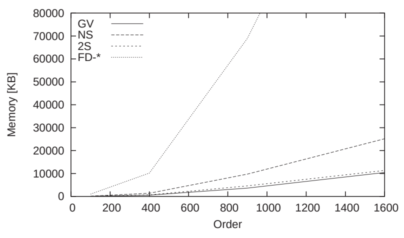
much worse than the graph model. The memory penalty comes from the large number of variables and propagators used in the FD model: $O(n^2)$ versus one in the graph model.

The first cause for the higher speed of the FD-FC model is the density of the search space: The path constraint as well as the arc consistency propagators of the FD model spend too much time on propagation. A second cause is the execution model: when dealing with a large graph, the CP(Graph) program traverses the whole graph for each constraint filtering algorithm even though the degree constraints operate very locally. The FD model is able to make good use of this locality as each degree constraint is only scheduled if an incident edge has been included or removed. This problem is dealt with in the next section.

We conclude that the CP(Graph) approach to the Knight's tour problem leads to a program using much less memory but spending much more time than the FD-FC approach. Hence the CP(Graph) approach allows to solve larger problems than the FD model as memory is often a more limiting factor than computation time.



(a) Running times



(b) Memory

Figure 7.4: Knight's tour problem: Plot of the data in table 7.4, search for 1 solution.

7.4 Comparing Implementations of Graph Intervals

In this section, we compare the efficiency of several graph interval implementations: In Chapter 6, four models for graph intervals are presented with a common API. We show that the graph domain API allows to make graph models interoperate in any constraint using this API. We give an example and use it for a first comparison of efficiency. Then we also look at CSPs using other constraints such as *Connected* or *Path* to compare the efficiency of the graph interval implementations on other problems.

7.4.1 The $Complement(G_1, G_2)$ constraint

The objective for designing and implementing a common graph domain API is to allow the interoperability of the models of this API: In this experiment we show that a single propagator implementation can be used independently of the implementation of the graph intervals it operates on. Moreover, we show that CP(Graph) allows to use different implementations of graph domains as different parameters of a single graph constraint.

The CSP consists of two graph variable and a complement constraint linking them. The graphs are initialized with complete upper bounds of order s (graph K_s).

$$Subgraph(G_1, K_s) \wedge Subgraph(G_2, K_s) \wedge Complement(G_1, G_2)$$

Experiment 7.6. We solve this CSP for different values of s: from 20 to 100 by steps of 20. The nine possible combinations of the three generic graph models (graph variable type, adjacency sets and 2-sets) are tested.

The DFS search engine is used with a fixed recomputation distance of 8 and an adaptive recomputation distance of 2. A naive lexicographical order heuristic which includes the first available arc in the left branch of the search tree is used to search for 1000 solutions of the CSP.

The figures in table 7.5 and figure 7.5 tell that the N-sets model performs better for this constraint closely followed by the graph implementation and way behind, the 2-sets. This big difference comes from the use of the iteration based adjacency propagator on the 2-sets model as removing this constraint leads to a much quicker program. The time curves on figure 7.5(a) are divided in 3 beams: the slowest is when the two sets are used for the branching (2S-*), the second is when that model is used with a graph or N sets. Finally the fastest group of combinations is composed of the graph and N sets mixed. Regarding memory, the heuristic we used favors the models using sets: as only 1000 solutions are searched, there are long sequences of consecutive arcs included or excluded from the variables and the list of interval data structure used by set variables represent these consecutive arcs as one interval.

7.4.2 The *Path* constraint.

Looking back at the results in table 7.4 and figure 7.4, we can see that on that problem, the graph variable type is up to 13% faster and uses up to 60% less memory than the adjacency sets model. On the other hand, the 2-sets model features the same low memory consumtion as the graph variable type but with a lower speed comparable to that of the adjacency model.

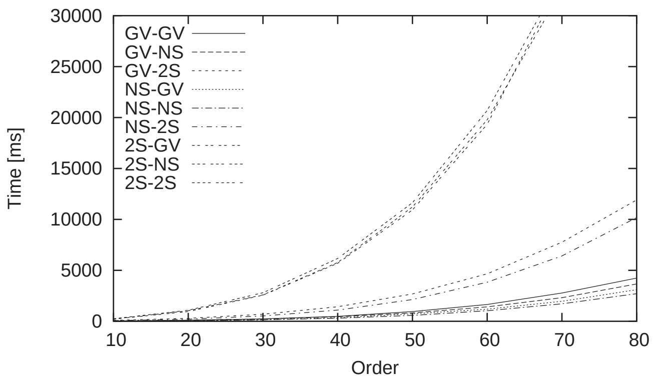
To summarize, we can say that the graph variable have slightly better performances than the adjacency model which in turn is more efficient than the 2-sets model.

7.5 Speeding Up Graph Variables

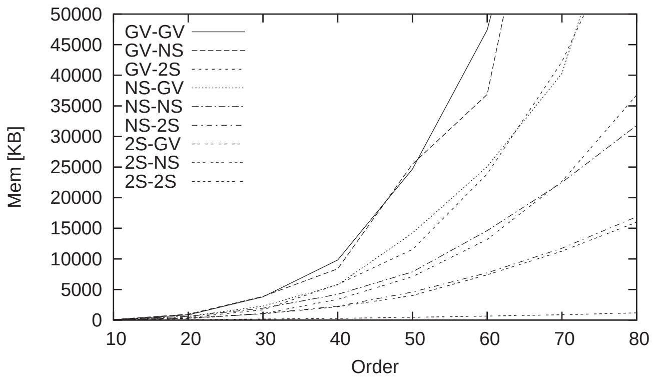
In section 7.3, we saw that the graph based model for the Knight's tour problem uses as much as 20x less memory than the FD model, but the latter is faster. The problem lies in the granularity of the variable types. While a coarser granularity results in less memory overhead (less variables and propagators to deal with), it results in a slower program given our execution model. There are two factors at play in this granularity problem: the time penalty for each operation on a variable's domain and the mechanics of propagator scheduling. In a nutshell, a graph propagator needs to traverse

	Time $[ms]$				
s:	20	40	60	80	100
GV-GV	100	490	1650	4240	9200
GV-NS	100	450	1420	3660	7970
GV-2S	290	1430	4660	11910	26410
NS-GV	70	340	1190	3100	6730
NS-NS	70	300	1030	2710	6080
NS-2S	210	1100	3830	10180	23060
2S-GV	960	5690	19330	51580	128750
2S-NS	1040	5770	19780	52970	126210
2S-2S	1090	6160	20700	54120	126050
	Memory $[KB]$				
s:	20	40	60	80	100
GV-GV	841	9820	47399	1525202	311258
GV-NS	962	8382	36873	1292299	269411
GV-2S	410	5829	23875	68166	171390
NS-GV	595	5758	25056	78240	156055
NS-NS	608	4216	14611	31801	60458
NS-2S	332	2272	7715	16887	34952
2S-GV	246	3387	13176	36783	93197
2S-NS	326	2171	7394	15990	30124
2S-2S	80	291	645	1168	1813

Table 7.5: Results for experiment 7.6 about $Complement(G_1, G_2)$: comparison of the running times and memory usage of all possible pairs of variable type among: the graph variable type (GV), the adjacency sets (NS) and the 2-sets (2S).



(a) Running times



(b) Memory

Figure 7.5: Results for experiment 7.6 about *Complement*(G_1, G_2): comparison of the running times and memory usage of all possible pairs of variable type among: the graph variable type (GV), the adjacency sets (NS) and the 2-sets (2S).

the complete graph bound to handle even a small change in its bound while an FD propagator can focus on a couple of nodes.

In this section, we present two potential solutions for overcoming this efficiency problems. The first, node and edge descriptors, aims at allowing a propagator to focus on a smaller part of a graph bound and to speed up some operations. The second solution consists in adapting the execution model to support and use update logs (called Δ in AC-2001 [15]). In this new setting, a propagator does not traverse the two bounds of the interval but only an update log which reflects the differences in the domain since the last invocation of the propagator.

7.5.1 Node and Edge Descriptors

The Graph domain API presented in chapter 6 is based on value iterators. The value iteration is more natural than a range iteration from the point of view of graph algorithms and data structures. However, range iteration and the implementation of set bounds as lists of ranges have a benefit over value iteration when graph algorithms are not involved: sequence compression. That is all consecutive arc values in a graph bound are represented in a $O(1)$ data structure: a range, an interval of numbers. In this section, we perform experiments which exhibit the benefit of sequence compression and develop a competitive feature, node and edges descriptors for the graph variable type.

Experiment 7.7. We enumerate 100,000 subgraphs of a complete graph of given order by using a CSP with one graph model and no constraint. The search heuristic is the naive lexicographical heuristic.

On figure 7.6, three curves present the running times of experiment 7.7. As in the previous experiment with the complement constraint, the graph variable implementation and the N sets model have comparable running times while the two sets model stays behind because of its implementation of the adjacency constraint which consumes lots of time in a CSP with few pruning.

The heuristic used in experiment 7.7 is to branch on the first arc of $\overline{D}(G) \setminus \underline{D}(G)$ in lexicographic order. The complexity to access the first value of $\overline{D}(G) \setminus \underline{D}(G)$ by using iterator difference is $O(m+n)$. This leads to an overhead of $O(m(m+n))$ worst case along a path of the search tree for using this heuristic.

This large overhead is easily decreased when using the set based model for graph domains. The unknown values iterator can use the cardinality of

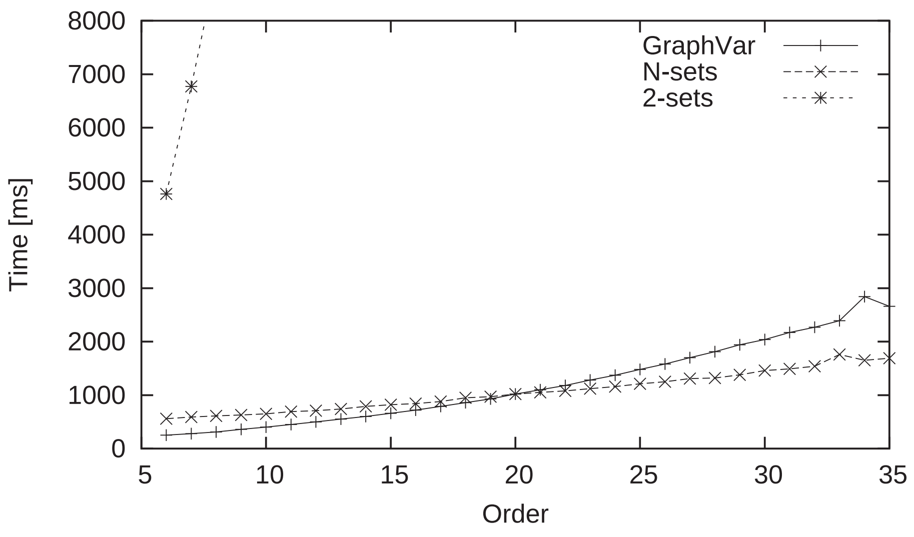


Figure 7.6: Compute 100000 subgraphs of a graph by using a branching based on bound iteration. The graph variable, N sets model, and 2 sets models are compared.

the sets to test if they are assigned in $O(1)$, leading to an access cost of $O(n)$ to get the right source node then $O(n)$ again to get to the right edge. The $O(n)$ search for the right node can be reduced to $O(1)$ thanks to the contractant property of filtering. As soon as the first sets are assigned the search strategy never needs to look at them again, so it can start the search beyond them. The total cost to access the next unknown arc is reduced to $O(n)$ worst case. That is $O(mn)$ worst case along a branch of the search tree.

This optimization can be applied to the graph domain data structure by using node and edge descriptors. If the branching strategy is allowed to start the search for the next unknown edge at the end of the glb prefix, then its amortized complexity drops to $O(m)$ along a path of the search tree: each edge is scanned only once. In order for them to be usable in this search heuristic, care must be taken to allow them to include or exclude an edge from the domain. For this, the edge descriptor points to the last glb element preceding the edge they refer to.

The copy mechanism of Gecode has to be taken into account as edge descriptors can be used across a copy of a space. We did not instrument the variable implementation to keep forwarding information for the edge descriptors. Instead we recompute the value of the edge descriptors the first time they are used after each copy.

Experiment 7.8. We experimentally compare the edge-descriptor-based lexicographical search heuristic with the built-in branching available in Gecode for arrays of sets. Experiment 7.7 is done again but instead of using the generic iteration based heuristic, the graph variable and set-based model use their own specialized heuristic.

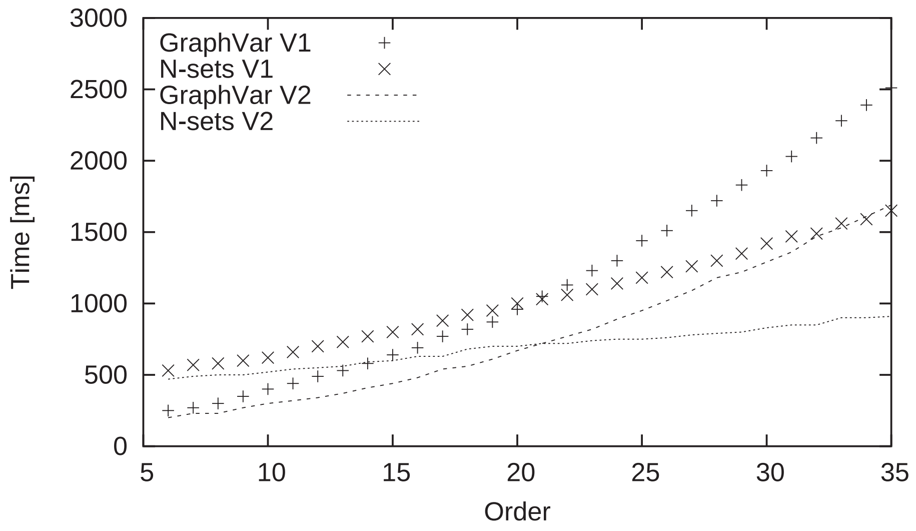
The results of experiment 7.8 are presented in figure 7.7(a) in which the new results are plotted along with the results of experiment 7.7. The old results are plotted with points and labeled V1 while the new results are plotted with lines and labeled V2. The running time for both models is clearly reduced. They still cross at the same point around order 22. The position of this crossing point depends on the number of solutions we search for. If only one solution is searched, the graph variable is always faster than the sets. On figure 7.8, we show the crossing point for 1000 nodes and one million edges, at 160 solutions. Two figures are presented: one with time against number of solutions and one with time against size of the graph. This shows that the graph variable type is faster than the set model for smaller graphs or smaller search tree.

This is explained by the cost of copy for the graph model: the more copy, the fastest the set model in this problem: The range list data structure of sets makes the graph representation more compact with the set model. Hence it is faster to copy; This effect is even stronger with large graphs and at the beginning of the search tree (we search for relatively few solutions). Moreover, the edge descriptor is recomputed at each copy hence the speedup due to edge descriptors is partially lost when copying the variable. We show in figure 7.7(b) that when using full recomputation the graph variable is always faster.

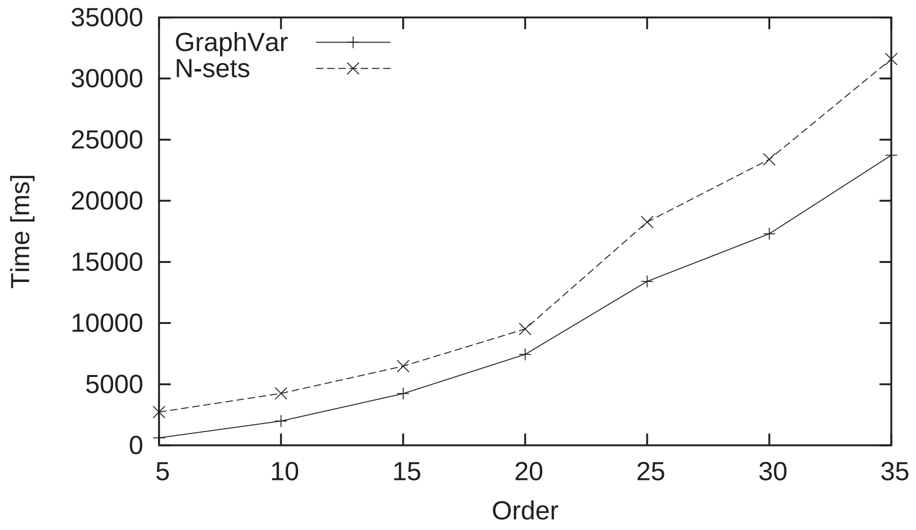
These experiments have shown that the use of edge descriptors can enhance the speed of seeking an edge in the graph variable type implementation. The edge descriptors even allow the coarse graph variables to perform better than finer grained set variables in this problem.

7.5.2 Breaking Granularity for Propagation: Update Logs

We have described how to speed up the traversal of the graph domain in order to perform a query or update in the graph variable implementation. In this section, we tackle another technical limitation related to the granularity of graph domains when used by constraint propagators.

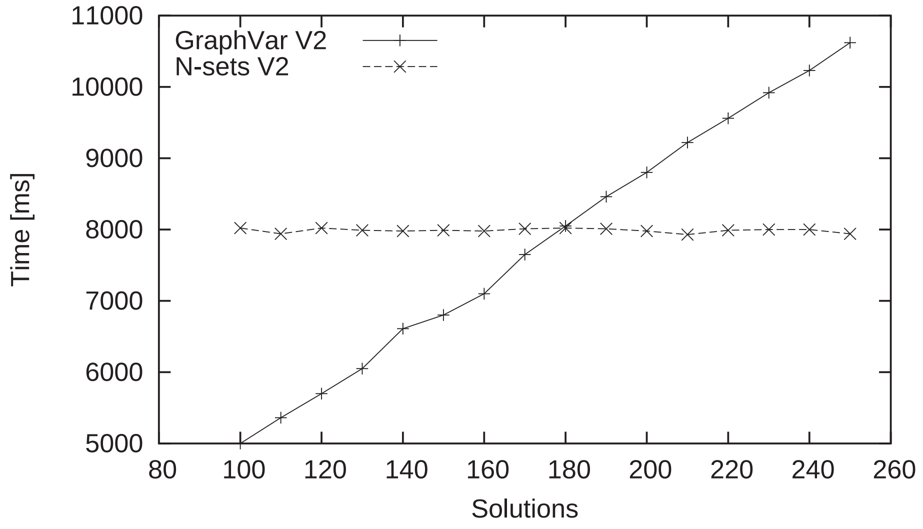


(a) Comparison of the new scheme with the old

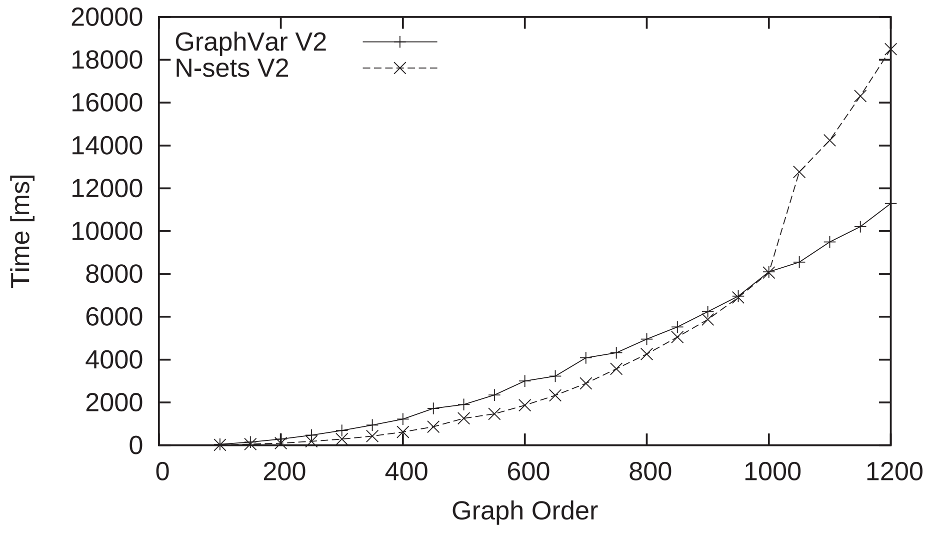


(b) Same experiment without copy (full recomputation)

Figure 7.7: Comparison of the iteration based branching (V1) with the built-in branching of sets and the edge-descriptor-based branching of graph variables (V2).

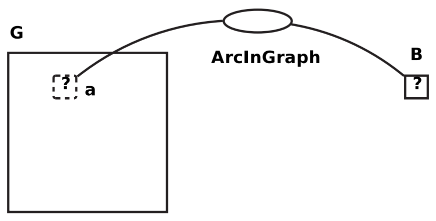


(a) Running time against number of solutions searched with a graph of order 1000 (one million of edges)

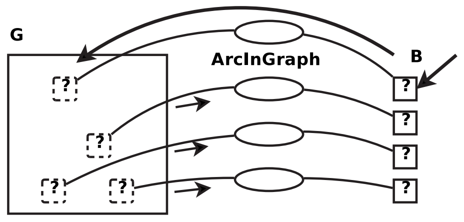


(b) Running time against order of the graph for 180 solutions

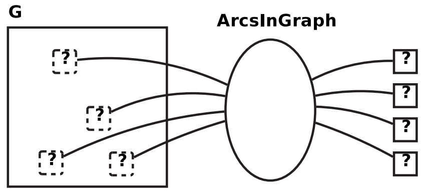
Figure 7.8: Comparison of the enumeration of subgraphs using edge descriptors for the graph variable and using the built-in set branching for the N sets model



(a) One *ArcInGraph* constraint



(b) Triggering of propagators with multiple *ArcInGraph* constraints



(c) Global constraint

Figure 7.9: Illustration of iteration and granularity on a simple example. The *ArcInGraph* constraint links a fixed arc a of a graph G with a Boolean variable B

Consider the reified constraint $ArcInGraph(G, a, B)$ which holds if the Boolean variable B denotes the presence of the fixed arc a in the graph G. This constraint is illustrated in figure 7.9(a). It is equivalent to $Arcs(G, SN) \wedge (B \Leftrightarrow a \in SN)$. The propagation rules of this constraint are very simple: if $B = 1$ then include a in G, if $B = 0$ then exclude a from G, if a belongs to $\underline{D}(G)$ then set $B = 1$ if it does not belong to $\overline{D}(G)$ then set $B = 0$. If there are conflicts, the constraint fails (*e.g.* $B = 0 \wedge a \in \underline{D}(G)$ or $B = 1 \wedge a \notin \overline{D}(G)$).

The edge descriptors can be used to speed up the query and tells related to the arc a in the graph but accessing arc a remains $O(n + d)$ in the worst case. This constraint is expected to be reasonably efficient as reified "isin" for sets has the same complexity, but we will see how to upgrade its query and update time to $O(1)$ by the end of this section.

Now consider that the user needs several such Boolean variables and posts several ($O(m)$) such constraints. The following situation is depicted in figure 7.9(b). When a Boolean variable is changed, its associated propagator is re-scheduled and eventually performs the update of the graph at a cost of $O(n + d)$. Then comes the granularity problem: All constraints subscribed to the graph variable are re-scheduled. Each of the *ArcInGraph* constraint is triggered and consumes $O(n + d)$ to check if it must propagate. The total cost associated to this update of one Boolean variable is $O(m(n+d))$. Even if the propagator can examine and update the graph variable in $O(1)$ this incurs an $O(m)$ cost for one update of a Boolean variable (plus the scheduling time).

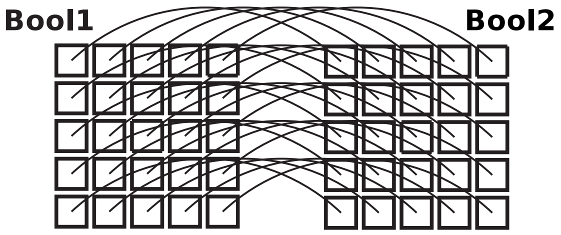
We performed an experiment to show this effect with $O(1)$ update time compared to a case where there is no such global triggering effect. In figure 7.10(a), the first CSP is simply two matrices of Booleans linked by individual equality constraints. In the second CSP presented in figure 7.10(b), each constraint is subscribed to all the Booleans of the matrix on the left. Each constraint only consults one precise Boolean variable which it can access in $O(1)$ but it is subscribed to all of them.

Experiment 7.9. The CSPs consist in two vectors of Booleans B and B' and a set of constraints $B_i = B'_i$ equating the Booleans in the vectors. Solutions are searched with a naive heuristic which operates on vector B. The CSP differ in the implementation of the Boolean equality constraint. In one CSP, the normal equality constraint is used, in the other one, the constraint $B_i = B'_i$ is subscribed to every variable in B'.

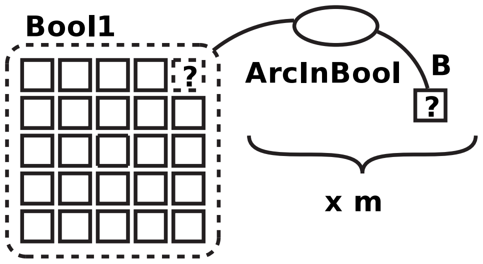
The experimental results are expected: the coarse model incurs an explosion of running time (see figure 7.10(c) where 100 solutions are searched).

One potential solution to avoid rescheduling $O(m)$ propagators is to use a unique global constraint as illustrated on figure 7.9(b). When the graph is modified it only re-schedules one propagator which must deal with all arcs and Boolean variables. It can do it in linear $O(m)$ time whatever the number of updates. When the Booleans are modified, Gecode does not provide the information of which variable has changed hence it also costs $O(m)$ to update the graph from the Booleans. Those two steps can be done in one pass over the two structures and we can use an edge descriptor and the count of unknown arcs in the graph to avoid scanning the prefix and postfix sequence of glb arcs. The update time stays $O(m)$ for k changes versus a theoretical $O(k)$ total cost.

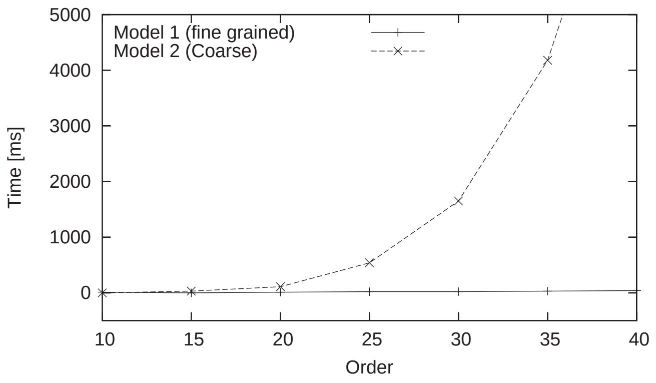
One solution to the problem of detecting which Boolean variable has changed is the use of daemons which are currently being integrated into Gecode by M. Lagerkvist at KTH. These daemons are entities which are not scheduled but directly triggered on the modification of a variable, such a daemon could be attached to each Boolean variable to inform the propagator of its modification.



(a) The model with the finest granularity



(b) Simulating graph granularity with $O(1)$ update



(c) experimental comparison

Figure 7.10: Illustration of iteration and granularity on a simple example. The *ArcInGraph* constraint links a fixed arc a of a graph G with a Boolean variable B

On the graph side, to avoid scanning the whole upper bound, it is possible to add a list of modifications, an update log, to the variable. Each propagator using this log would have a pointer to a position in this log and all modifications since it was last run would be available in the rest of the log. When a propagator is run it asks the variable to scan the rest of the log and obtains the k modification in $O(k)$. If the propagator is the oldest propagator subscribed to the variable, the beginning of the log can be freed.

To implement this for the upper bound a technical solution is to not release the memory cell which was occupied by the arc but append it to the update log. The `target` pointer of the `TargetEdge` struct can be used to mark this edge as deleted. Remember that edge descriptors cannot point to an arc of $\overline{D}(G) \setminus \underline{D}(G)$ as once deleted and released, this information is garbage.

The update log allows edge descriptors to point to the actual arc they want to query instead of on the last $\underline{D}(G)$ element before them. The query/update time then changes from $O(n + d)$ to $O(1)$. On copying the space, the update log would be emptied and the descriptors pointing to deleted arcs would be reset in the new space to their initial zero state.

To implement the update log of the lower bound, memory would need to be allocated to add new elements in the update log. Hence while the complexity for each query based on edge descriptors changes from $O(n+d)$ to $O(1)$ and while propagators which are interested in the whole list of changes see their complexity change from $O(m)$ to $O(k)$ (amortized $O(1)$ per change), the speed-up might only occur for large graphs because we need to allocate memory. Since this feature seems implementable and as it could speedup the incremental $Mono$ propagator for graph pattern matching, it will be further examined and implemented.

We showed that the edge descriptors allow to speed up the cost of some operations on graph variables but that the update log allows a complete shift of the propagators complexity. While the current execution model performs filtering from scratch every time a variable changed, this shift should allow an efficient use of randomized filtering (perform filtering from scratch only when it might be necessary) or incremental graph algorithms (perform filtering incrementally only when it is necessary). We believe that further research on the graph variable data structure could turn the current naive implementation into an very efficient model for graph domains.

7.6 Summary

In this chapter, we first applied the CP(Graph) implementation to a metabolic pathway recovery problem. This application suggests that the graph variable type is able to deal with large graphs (a graph of nore than 16000 nodes and 45000 arcs was used). Then we showed how the concept of non-fixed graph variables allows the modeling of a large class of graph pattern matching problems with a unique constraint.

We also performed experiments to assess the relative efficiency of the different graph interval models we proposed in chapters 3 and 6. These experiments suggest that the two most efficient models are the graph variable type and the adjacency sets model. The former is slightly more efficient but the latter is more convenient as it allows a direct access to the underlying out-neighbors sets.

Global constraints for graph properties have been compared with FD models in two CSPs: the connected subgraph problem and the Hamiltonian path problem. In both problems, the FD models consume several orders of magnitude more memory than the graph variable and global constraint approach. In the connected subgraph problem, the global constraint approach is up to twice as fast. In the Hamiltonian path problem, the finite domain model is faster. We suggested several explanations for this difference of efficiency among which the notion of granularity of large variable types like graphs and, to a smaller extent, sets.

We proposed two mechanisms to break the granularity barrier: node and edge descriptors allow to refer to precise elements of a graph bound and an update log allows a propagator to fetch the individual elements which were removed from the variable domain in constant time per change. This update log could allow the implementation of much faster filtering algorithms for very local constraints such as the degree constraints used in the Hamiltonian problem.

Chapter 8

Conclusion and Future Work

The objective of this thesis was the design, study and implementation of a computation domain in constraint programming. The main contributions of this thesis and perspectives on future work are presented below.

After the introduction and background chapters, in chapter 3 we have presented the graph interval domain abstraction used for graphs in this thesis. Three kernel graph constraints have been presented and used to decompose combinatorial graph problems into set and finite domain constraints. We also showed how graph intervals extend beyond graph CSPs and can be used in filtering algorithms of many finite domain constraints. This could lead to an implementation of the generic filtering algorithm of [5] using graph intervals and constraints. Also, other domain approximations could be investigated depending on the targeted type of applications (such as in [56] for sets). Such applications could for instance be in computational geometry or about finite state automatons.

In chapter 4, we presented and studied a list of constraints enforcing various graph properties. For each constraint we presented optimal filtering rules and their implementation as a filtering algorithm. We contributed several new constraints, new filtering rules for existing constraints and extensions or improvements of filtering algorithms. This list of filtering algorithms could lead to the design of incremental algorithms as suggested by the inclusion of an update log in the implementation of the graph variable type.

In chapter 5, we introduced two novel constraints on weighted spanning trees: the weight bounded spanning tree optimization constraint and the minimum spanning tree constraint. This work was mostly theoretic and could lead to further application to network design problems as suggested by the previous application of a small part of the filtering of MST to robust spanning tree design by Aron et al. in [1]. This extensive study of two complex graph constraints could also lead to the systematization or even automatization of graph constraint design and implementation.

In chapter 6, we presented a proof of concept implementation of the CP(Graph) computation domain over the Gecode constraint programming library. It provides several implementations of graph intervals either as set or integer based models or as a new variable type of Gecode, all gathered under a unified interface for propagators dealing with them. This proof of concept could be extended to better support incremental computations. It could also provide specialized implementations for some types of graphs.

Finally, in chapter 7, we applied our implementation of CP(Graph) to biochemical pathway analysis. We showed that this implementation supports graphs of sixteen thousands nodes and forty thousand arcs and is able to find constrained shortest paths in these graphs. We also conducted experiments to compare a finite domain model and a global graph constraint model for the connected subgraph enumeration problem. It showed the CP(Graph) model saves several order of magnitude of memory and is faster, allowing to deal with much larger graphs using graph variables instead of finite domain models. Comparing the global path constraint and finite domain model for the Hamiltonian path suggested that some enhancements could be made to the CP(Graph) implementation. Two such enhancements have been proposed including the use of an update log which changes the execution model of propagators to a model closer to that of AC-2001. These enhancements could lead to an efficient support for randomized and incremental filtering for graph constraints. This could in turn lead to a support of even larger graphs using the graph variable approach.

A broader perspective on future work would consist in an improved automatic exploitation of some structural properties of the CSPs. Additional properties arising from the combination of constraints can be exploited by adding implied constraints or using specialized filtering algorithms. The automatic identification of useful implied constraints or even the automatic generation of efficient specialized global filtering algorithms could be considered to automate the tuning of constraint programs. Some graph properties have many similarities and filtering algorithms for these properties use similar algorithmic components. Further work on filtering algorithms for these properties could lead to the identification of ways to systematize or even automatize the design of filtering algorithms.

Index of Graph Theory Definitions

Appendix A

Filtering Rules of Graph Constraints

In this appendix, we list formal pruning rules for each of the constraints in chapter 4. The format is the same as introduced in section 2.1.3.

$Subgraph(G_1, G_2)$

- $\underline{D}(G_2) \leftarrow \underline{D}(G_2) \cup \underline{D}(G_1)$
- $\overline{D}(G_1) \leftarrow \overline{D}(G_1) \cap \overline{D}(G_2)$

$Nodes(G_1) = Nodes(G_2)$

- $Nodes(\underline{D}(G_1)) \leftarrow Nodes(\underline{D}(G_2)) \cup Nodes(\underline{D}(G_1))$
- $Nodes(\overline{D}(G_1)) \leftarrow Nodes(\overline{D}(G_1)) \cap Nodes(\overline{D}(G_2))$
- Symmetrical rules for pruning G_2

$Complement(G_1, G_2)$
We abuse notation by denoting the bounds of $Arcs(D(G))$ as $D(G)$ to simplify the notation.

- Rules of $Nodes(G_1) = Nodes(G_2)$
- Filtering of the arcs:
- $\underline{D}(G_1) \leftarrow \underline{D}(G_1) \cup (Nodes(\underline{D}(G_2)) \times Nodes(\underline{D}(G_2))) \setminus \overline{D}(G_2)$
- $\overline{D}(G_1) \leftarrow \overline{D}(G_1) \setminus \underline{D}(G_2)$
- Symmetrical rules for pruning G_2

$Symmetric(G)$

- $Arcs(\underline{D}(G)) \leftarrow \{(u,v)|\{(u,v),(v,u)\} \cap Arcs(\underline{D}(G)) \neq \emptyset\}$
- $Arcs(\overline{D}(G)) \leftarrow \overline{D}(G_1) \setminus \{(u,v)|1 = \#(\{(u,v),(v,u)\} \cap Arcs(\underline{D}(G)))\}$
- Symmetrical rules for pruning G_2

$Undirected(G, G_u)$

- Rules of $Nodes(G_1) = Nodes(G_2)$
- Filtering for the arcs:
- $Arcs(\underline{D}(G_u)) \leftarrow \{(u,v)|\{(u,v),(v,u)\} \cap \underline{D}(G_u) \neq \emptyset\}$
- $Arcs(\overline{D}(G_u)) \leftarrow \overline{D}(G_u) \setminus \big\{(u,v)|\{(u,v),(v,u)\} \cap \overline{D}(G_u) = \emptyset\big\}$
- $Arcs(\overline{D}(G)) \leftarrow \overline{D}(G) \cap \overline{D}(G_u)$
- $Arcs(\underline{D}(G)) \leftarrow \underline{D}(G) \cup \big\{(u,v)|(u,v) \in \overline{D}(G) \cap \underline{D}(G_u) \wedge$
  $(v,u) \in \underline{D}(G_u) \setminus \overline{D}(G)\big\}$

$Connected(G)$
The graph G is an undirected graph: $Symmetric(G)$ holds (it can be enforced or can just hold as an invariant of the CSP).

- If $\underline{D}(G) = \emptyset$ then no filtering. Else:
- $\overline{D}(G) \leftarrow 1$ connected component of $\overline{D}(G)$ which contains $\underline{D}(G)$
- $\underline{D}(G) \leftarrow \underline{D}(G) \cup$ all bridges and cutnodes of $\overline{D}(G)$ on a path between nodes of $\underline{D}(G)$.

$StronglyConnected(G)$

- If $\underline{D}(G) = \emptyset$ then no filtering. Else:
- $\overline{D}(G) \leftarrow 1$ strongly connected component of $\overline{D}(G)$ which contains $\underline{D}(G)$
- $\underline{D}(G) \leftarrow \underline{D}(G) \cup \big\{x|\overline{D}(G) \setminus \{x\}$is not strongly connected$\big\}$ where x denotes either a node or an arc.

$DAG(G)$

- $Arcs(\overline{D}(G)) \leftarrow Arcs(\overline{D}(G)) \setminus \{(u,v)|(v,u) \in TC(\underline{D}(G))\}$

$Bipartite(G)$
Let t be a spanning forest of $\underline{D}(G)$. Let $C : Nodes(t) \rightarrow \{b, w\}$ be a arbitrary 2-coloring of t $(\forall(u,v) \in Arcs(t) : C(u) \neq C(v))$.

- $Arcs(\overline{D}(G)) \leftarrow Arcs(\overline{D}(G)) \setminus \{(u,v) | u \in Nodes(\underline{D}(G)) \wedge v \in Nodes(\underline{D}(G)) \wedge C(u) = C(v)\}$

$Weight(G, W, I)$
Let $minW([g_1, g_2])$ be the weight of the minimum weight graph in the interval. Similarly $maxW$ denotes the weight of the maximum weight graph in the interval. See section 4.5 for an algorithm to compute these.

- $\overline{D}(G) \leftarrow \overline{D}(G) \setminus \left\{x | minW\left([\underline{D}(G) \cup \{x\}, \overline{D}(G)]\right) > \overline{D}(I)\right\} \setminus \left\{x | maxW\left([\underline{D}(G) \cup \{x\}, \overline{D}(G)]\right) < \underline{D}(I)\right\}$
- $\underline{D}(G) \leftarrow \underline{D}(G) \cup \left\{x | minW\left([\underline{D}(G), \overline{D}(G) \setminus \{x\}]\right) > \overline{D}(I)\right\} \cup \left\{x | maxW\left([\underline{D}(G), \overline{D}(G) \setminus \{x\}]\right) < \underline{D}(I)\right\}$

$QuasiPath(G, n_1, n_2)$

- First, n_1 and n_2 are added to $\underline{D}(G)$
- All edges (x, n_1) and (n_2, x) are removed from $\overline{D}(G)$. Then,
- All edges of $\overline{D}(G)$ which are the only edge incident to a node of $\underline{D}(G)$ are added to $\underline{D}(G)$
- For each edge (u, v) of $\underline{D}(G)$, all other edges (u, x) and (x, v) are removed from $\overline{D}(G)$

$Path(G, n_1, n_2)$ (Undirected)
The graph G is an undirected graph: $Symmetric(G)$ holds (it can be enforced or can just hold as an invariant of the CSP).

- All bridges and cutnodes of $\overline{D}(G)$ not on a path from n_1 to n_2 are removed from $\overline{D}(G)$
- $\overline{D}(G)$ is set to its connected components containing n_1 and n_2
- All bridges and cutnodes of $\overline{D}(G)$ on a path from n_1 to n_2 are added to $\underline{D}(G)$
- The rules of the $QuasiPath(G, n_1, n_2)$ constraint

$Path(G, n_1, n_2)$ (Directed)
Let $TC(\overline{D}(G))$ denote the set of arcs in the transitive closure of $\overline{D}(G)$,

- Remove from $\overline{D}(G)$ each edge $(u, v) \in \overline{D}(G)$ such that $(n_1, u) \notin TC(\overline{D}(G)) \vee (v, n_2) \notin TC(\overline{D}(G))$
  To do this, you can add a temporary arc (n_2, n_1) to $\overline{D}(G)$ and keep only the strongly connected component containing n_1 and n_2. Remove the arc (n_2, n_1).
- Remove from $\overline{D}(G)$ each edge $(u, v) \in \overline{D}(G)$ such that
  $\exists n \in Nodes(\underline{D}(G)) : (n, u) \notin TC(\overline{D}(G)) \wedge (v, n) \notin TC(\overline{D}(G))$
  To do this, in the SCC-reduced graph of $\overline{D}(G)$, remove all arcs which jump over a node of $\underline{D}(G)$.
- Include all nodes and edges which dominate a node of $\underline{D}(G)$ from n_1 in $\overline{D}(G)$.
- Include all nodes and edges which dominate a node of $\underline{D}(G)$ from n_2 in the reverse of $\overline{D}(G)$.

$Path(G, n_1, n_2, w, i)$
Let $d(x, y)$ denote the length of the shortest path from x to y according to lengths in w.

- Remove from $\overline{D}(G)$ all arcs (u, v) such that $d(n_1, u) + w(u, v) + d(v, n_2) > i$
- Perform the filtering of $Path(G, n_1, n_2)$.

Appendix B

Experimental Setting

The experiments were performed on a computer equipped with an Intel Xeon 2.66GHz processor and 2Go of RAM (durer.info.ucl.ac.be as of Aug. 2006). Gecode version 1.0.1 and CP(Graph) version 0.9.1 were used.
Gecode version 1.0.1 can be obtained via SVN at `https://svn.gecode.org/svn/gecode/tags/release-1.0.1`.
CP(Graph) version 0.9.1 can be obtained via SVN at `https://savane.info.ucl.ac.be/svn/cp_graph_map/branches/0.9.1/graph`.

The details of all experiments are listed below.

Experiment 7.1 The script for the experiment is available at `examples/experiments/cpgraph-path_metab.cc`.
The graphs are available in `examples/experiments/graphs.zip`.
The program is run as `./cpgraph-path_metab g200_croes_ecoli_glyco [options]`. This loads the graph in `g200_croes_ecoli_glyco_graph.txt` with mandatory nodes in `g200_croes_ecoli_glyco_seeds.txt` and pairs of reverse reactions in `g200_croes_ecoli_glyco_revs.txt`. For info, the actual pathway is present in `g200_croes_ecoli_glyco_sol.txt`. The program is built on the `Example` class so any option can be specified (such as recomputation distances). The default recomputation distances of `c_d=8` and `a_d=2` have been used to produce the result presented in table 7.1.

Experiment 7.2 The script for the experiment is available at `examples/experiments/cpgraph-path_metab_node_SPT.cc`.
It is run in the same fashion as for experiment 7.1.

Experiment 7.3 The script for the experiment is available at `examples/experiments/cpgraph-path_metab_node_SPT.cc`.
This experiment consists in removing one node out of two (except the first and the last) in the files `*_seeds.txt`. A script `split_seeds.py` is provided that does it (it saves the original files with a `.orig` extension).

Experiment 7.4 The script for the Boolean model of the connected constraint is `examples/experiments/connected.cc`. The script for the CP(Graph) model of the connected constraint is `examples/experiments/cpgraph-connected.cc`. Both scripts are run with the order of the graph as first argument and `Example` options after that. The CP(Graph) model also compares two different implementations of the connected constraint: `CheckConnected` rebuilds the bounds graphs at each propagator invocation while `IncrCheckConnected` keeps the bounds graphs and update it at each invocation.

Experiment 7.5 The script for the integer model of the Knight's tour is `examples/knights.cc` in the Gecode tree. The script for the CP(Graph) model of the Knight's tour is `examples/experiments/cpgraph-knights.cc`.
Both scripts are run with the order of the graph as first argument. For the CP(Graph) script, the two last arguments must be the view to use (`GraphVarView`, `OutAdjSetsGraphView`, `NodeArcSetsGraphView`) and for the graph variable implementation, 1 to use the classic iterator based branching and 2 to use the node descriptor based branching.

Experiment 7.6 The script for the complement experiment is `examples/experiments/cpgraph-complement.cc`.
The script is run with the order of the graph as first argument. The two last arguments must be the views to use for the two graph intervals (`GraphVarView`, `OutAdjSetsGraphView`, `NodeArcSetsGraphView`).

Experiment 7.7 and 7.8 The script for the subgraph enumeration experiment is `examples/experiments/cpgraph-gen.cc`.
The script is run with the order of the graph as first argument. The next argument must be the view to use for the graph interval (`GraphVarView`, `OutAdjSetsGraphView`, `NodeArcSetsGraphView`). Finally a number 1, 2 or 3 specifies which branching or implementation to use. 1 is the iterator based branching. 2 and 3 use the specialized branching and 3 runs the test with adjacency graphs twice: once with a set based adjacency constraint and once with an iterator based adjacency constraint.

Experiment 7.9 The script for the subgraph enumeration experiment is `examples/experiments/cpgraph-granularity.cc`.
The script is run with the order of the graph as first argument. It compares the normal Boolean model with simple equality propagators and the same CSP but with a modified *ArcInGraph* propagator which subscribes to the whole vector.

Appendix C

CP(Graph) Code Examples

C.1 Modeling a Graph Problem

The following program illustrates the definition of a new search heuristic using the Gecode `Branching` class and the definition of a CP(Graph) CSP.

C.1.1 A search heuristic

On line 7, a branching description `ArcBD` is defined for describing a choice made on an arc

On line 11, a new branching – a class implementing a search strategy – is implemented. This branching does no fancy computations. It simply includes the first available arc in the left branch and excludes it in the right branch. Its particularity is that it is defined for undirected graphs: it includes or excludes two arcs at the same time: the chosen arc and its reverse. The branching maintains the invariant that the graph is symmetric. As the connected constraint maintains this invariant too, the *Symmetric* constraint need not be posted.

Lines 18 to 30 define constructors for space copy (cloning) and for posting the branching. These are not really specific to this branching.

On line 31, the `branch` method is defined. This method call is triggered by the search engine when a space has reached a fixed point and new alternatives for this space have to be defined. This method is responsible for deciding how many alternatives are present for this choice point. In this case the number is 2 if the graph variable is not fixed and 0 otherwise. On line 32, the `iter_arc_Unk()` method is called on the graph view to produce a `UnkArcIterator`, an iterator on the unknown arcs of the graph interval (the arcs in $\overline{D}(G) \setminus \underline{D}(G)$). If there is such an arc, a choice is made to include it in the left branch on line 34.

On line 45, the `description` method returns the branching description computed for the current choice point.

On line 47, the `commit` method actually performs the choice by posting the appropriate constraint to turn this space into a child space corresponding to the alternative numbered `a`. On lines 50-55, the branching description is recomputed if it was not passed as an argument. On lines 57-64, the inclusion of the two arcs is performed for the left branch (`lbd->action` was set to `true` on constructing the `ArcBD` and the left alternative is `a==0`). The two arcs are included in the lower bound and the result of this inclusion is tested to check if it lead to a CSP failure on line 62.

On line 85, the `copy` method calls the cloning constructor. This method is used by the Gecode kernel during space cloning.

C.1.2 The CSP

The CSP is defined as a class extending the class `Example` which extends `Space`. The graph view type and the propagator type are template parameters of this class. On line 88, the constructor initializes the upper bound of the graph variable `g1` as a complete directed graph. Then all loops are removed to make it a model for a complete undirected graph (lines 90-94). On line 95 the connectedness constraint is posted on the graph view and on line 96, the search strategy is specified.

The rest of the class is typical of a Gecode space, in the constructor for cloning the space, the `update` method must be called on the graph variable as on any other type of variable. The `copy` method calls the copy constructor, and the `print` method is called by the search engine for each solution. This last method is necessary only for the predefined search engines of Gecode; If a new search engine is implemented it could call other methods of the space to use the solution.

Finally, in the main function, an Option object is constructed, the command line arguments are parsed and two calls to the search engine are made with two different propagators for the connectedness constraint. Both calls are done with the `GraphVarView`, the view over the graph variable type, but other views, implementing different graph interval models could be substituted (such as `OutAdjSetsGraphView` or `NodeArcSetsGraphView` for instance).

```
#include "examples/support.hh"
#include "graphutils.icc"
```

```
#include "graph.hh"
using namespace Gecode::Graph;
// branching descriptor for arcs
typedef GraphBDSingle<std::pair<int,int> > ArcBD;

// This branching uses the naive strategy and posts the constraint for the arc
// and its reverse
template<class GView>
class UndirectedBranching: public Branching {
        GView g;
        ArcBD * bd;

    public:
        /// Constructor for cloning \a b
        UndirectedBranching(Space* home, bool share, UndirectedBranching& b):
            Branching(home,share,b),bd(NULL){
                g.update(home,share,b.g);
            }

        void static post(Space* home, const GView& g){
            (void) new (home) UndirectedBranching<GView>(home,g);
        }
        /// Constructor for creation
        UndirectedBranching(Space* home, const GView& g):
            Branching(home),g(g),bd(NULL){}
        /// Perform branching
        virtual unsigned int branch(void){
            typename GView::UnkArcIterator i=g.iter_arcs_Unk();
            if (i()){
                bd = new ArcBD(this, i.val(), true);
                return 2;
            }
            return 0;
        }

        /// Return branching descriptor (of type Gecode::Graph::GraphBDSingle)
        virtual BranchingDesc* description(void) {return bd;}
        /// Perform commit for alternative \a a and branching description \a d
        virtual ExecStatus commit(Space* home, unsigned int a,
                BranchingDesc* d){
            ArcBD* lbd ;
            if (d){
                lbd = static_cast<ArcBD*>(d);
            } else {
                branch();
                lbd = bd;
            }

            int u,v;
            boost::tie(u,v) = lbd->elem;
            if (a==1^lbd->action) {
                ModEvent e1 = g._arcIn(home,u,v);
                ModEvent e2 = g._arcIn(home,v,u);
                if (me_failed(e1)|| me_failed(e2)){
                    return ES_FAILED;
                } else {return ES_OK;}
            } else {
                ModEvent e1 = g._arcOut(home,u,v);
                ModEvent e2 = g._arcOut(home,v,u);
                if (me_failed(e1)||me_failed(e2)){
                    return ES_FAILED;
                } else {return ES_OK;}
            }
            return ES_FAILED;
        }
        /// Perform cloning
        virtual Actor* copy(Space* home, bool share) {
            return new (home) UndirectedBranching(home,share,*this);
        }
};

/** \brief Example to test a connectedness constraint
 * \ingroup Examples
 * */
// counts number of solutions.
// Used in CPGraphConnected::print
int sol=0;

template <class GraphView, template <class> class ConnPropag>
class CPGraphConnected: public Example {
    private:
        GraphView g1;
    public:
        // Constructor
        CPGraphConnected(const Options& opt): g1(this,opt.size){
            // remove loops
            pair<int,int> loops[opt.size];
            for (unsigned int i = 0; i<opt.size; i++)
                loops[i] = make_pair(i,i);
            StlToGecodeValIterator<pair<int,int>*> rem (loops,loops+opt.size);
            GECODE_ME_FAIL(this,g1._arcsOut(this,rem));
            GECODE_ES_FAIL(this, ConnPropag<GraphView>::post(this, g1));
            UndirectedBranching<GraphView>::post(this,g1);
        }
        /// Constructor for cloning \a s
        CPGraphConnected(bool share, CPGraphConnected& s) :
            Example(share,s){
                g1.update(this, share, s.g1);
        }
        /// Copying during cloning
        virtual Space*
            copy(bool share) {
                return new CPGraphConnected(share,*this);
            }
        /// Print the solution
        virtual void
            print(void) {
                std::cout << "Solution " << ++sol<< ':' <<
                    std::endl<< g1 << std::endl;
            }
};

int
main(int argc, char** argv) {
    Options opt("CPGraphConnected");
    opt.icl          = ICL_DOM;
    opt.solutions         = 0;
    opt.parse(argc,argv);
    if (opt.size == 0) {opt.size =1;}
    Example::run<
        CPGraphConnected<GraphVarView, CheckConnected>,DFS>(opt);
    sol = 0;
    Example::run<
        CPGraphConnected<GraphVarView, IncrUBConnected>,DFS>(opt);
    return 0;
}
```

C.2 Implementing a Constraint Propagator

We present a propagator which prunes the upper bound of the graph for the *Connected*(G) constraint. This propagator uses the bounds graphs and calls a simple DFS from the Boost library to mark nodes of the upper bound in the same connected component as a node of the lower bound.

The class `UBConnected` is a propagator for the connectedness constraint. The `propagate` method is where the filtering algorithm is implemented. The `post` method is called to add the constraint to a CSP. The `cost` and `copy` methods are called by the Gecode kernel during scheduling and space cloning.

Lines 33-60 contains various predicates and helper functions which are used later.

On line 65, the `UBConnectedBoundsGraphs` class is defined. This class inherits from `BoundsGraphs` and implements the filtering algorithm in its `filter_upper_bound` method. That method spans lines 81 to 136. First, on lines 88-91, the order of the upper bound is checked and the propagator is stopped with failure of the CSP if the graph is empty.

On lines 92-98, a node of the lower bound is sought and if it is not found, the propagator stops with no modification. Otherwise an empty visitor (l. 100), and a color map associating a color to each vertex (l. 101-102) are instantiated. The color map is initialized (l. 104-107) and depth-first search is performed in the upper bound (`UB`) from the start vertex `s` (l. 105).

On lines 110-120, iterator adapters are instantiated in order to iterate on every remaining white vertex in the upper bound. On line 123, the actual filtering is done by calling the `_nodesOut` method of the graph view. The result of this filtering is passed to the caller to be dealt with.

On lines 132-155, the classical methods of every propagator are implemented. They consist in calling `update`, `cancel` and `subscribe` on each variable of the scope of the propagator. The `cost` method defines a scheduling priority between propagators on the same variable type.

In the `propagate` method, updating of the bounds graphs is done on lines 157-161. For the call to the filtering algorithm on line 162, the macro `GECODE_ME_CHECK` returns immediately if the returned value is a failure. Finally, if the graph is fixed (assigned) then subsumption of the propagator is signaled and otherwise, fixed point is signaled. Note that here we have a fixed point, otherwise we would return `ES_NOFIX`.

```
#include "boost/iterator/filter_iterator.hpp"
#include "boost/iterator/transform_iterator.hpp"
#include "boost/graph/depth_first_search.hpp"
#include "boost/graph/connected_components.hpp"                5
#include "boost/property_map.hpp"


namespace Gecode { namespace Graph {
                                                               10
template <class> class UBConnectedBoundsGraphs;
```

```
/*
 *
 * Upper bound pruning for connected with incremental boundsGraphs
 *
 */
template <class GView>
class UBConnected : public Propagator {
    protected:
        GView g;
        UBConnectedBoundsGraphs<GView> *bg;
    public :
        UBConnected(Space* home, bool share, UBConnected& p);
        ~UBConnected(void);
        UBConnected(Space* home, GView &g);
        virtual Actor*       copy(Space*,bool);
        virtual PropCost     cost(void) const;
        virtual ExecStatus   propagate(Space*);
        static ExecStatus post(Space* home, GView &g) ;
};

typedef Gecode::Graph::Graph::vertex_descriptor Vdesc ;
typedef std::map<Vdesc,boost::default_color_type> CMT;
/// used as internal of UBConnectedBoundsGraphs::filter_upper_bound
/// iswhite : unary predicate
struct is_white: public unary_function<Vdesc,bool> {
    boost::associative_property_map<CMT> & map;
    is_white(boost::associative_property_map<CMT> & m):map(m) {}
    bool operator()(Vdesc v)const {
        return get(map,v) == boost::white_color;}
};
/// used as internal of UBConnectedBoundsGraphs::filter_upper_bound
/// returns id of vertex v in graph UB
int _get_id_(Vdesc v, const Gecode::Graph::Graph & UB ){
    return UB[v].id;
}
/// used as internal of UBConnectedBoundsGraphs::filter_upper_bound
/// returns id from some object Desc
template <class Desc>
    struct get_id: public unary_function<Desc,int> {
        Gecode::Graph::Graph & UB;
        get_id(Gecode::Graph::Graph & UB):UB(UB) {}
        int operator()(Desc v)const;
    };
typedef pair<Vdesc,boost::default_color_type> CMapDesc;
template<>
```

```
    int get_id<CMapDesc>::operator()(CMapDesc v) const {
        return _get_id_(v.first,UB);}
template<>
    int get_id<Vdesc>::operator()(Vdesc v) const { return _get_id_(v,UB);}

/// The class for the bounds graphs
template <class GView>
    struct UBConnectedBoundsGraphs: public BoundsGraphs<GView> {
        GView g;
        using BoundsGraphs<GView>::UB;
        using BoundsGraphs<GView>::LB;
        using BoundsGraphs<GView>::UB_v;
        using BoundsGraphs<GView>::LB_v;
        using BoundsGraphs<GView>::numNodes;
        using BoundsGraphs<GView>::arcOut;
        using BoundsGraphs<GView>::nodeOut;
        using BoundsGraphs<GView>::safe_add_nodeLB;
        using BoundsGraphs<GView>::update_bounds;
        using BoundsGraphs<GView>::check_consistent;
        UBConnectedBoundsGraphs(GView &g):BoundsGraphs<GView>(g), g(g){
            assert(check_consistent());
        }

        ModEvent filter_upper_bound(Space *home){
            typedef Gecode::Graph::Graph::vertex_descriptor Vdesc ;
            typedef Gecode::Graph::Graph::vertex_iterator Viter ;
            typedef std::map<Vdesc,boost::default_color_type> CMT;
            Vdesc s;
            Viter vi, vi_end;

            if (g.lubOrder()==0){
                // fails on empty graph
                return ME_GRAPH_FAILED;
            }
            typename GView::GlbNodeIterator it = g.iter_nodes_LB();
            // search for a starting point;
            if (!it()){
                return ME_GRAPH_NONE;
            } else {
                s = UB_v[it.val()];
            }// we have a starting point , do dfs

            boost::dfs_visitor<boost::null_visitor> t;// empty visitor
            CMT v2c;// one color per node
            boost::associative_property_map<CMT> cmap(v2c);
            // set all nodes white
            boost::tie(vi,vi_end) = vertices(UB);
            for (;vi!=vi_end; ++vi){
                put(cmap,*vi,boost::white_color);
            }
            boost::depth_first_visit(UB,s,t, cmap);

            // depth first done; do the pruning
            boost::tie(vi,vi_end) = vertices(UB);
            is_white iw(cmap);
            typedef boost::filter_iterator<is_white, Viter> WhiteIter;
            WhiteIter beginW(iw,vi,vi_end);
            WhiteIter endW(iw,vi_end,vi_end);
            get_id<Vdesc> gid(UB);
            typedef boost::transform_iterator<get_id<Vdesc>, WhiteIter>
                RemoveIter;
            RemoveIter begin(beginW,gid);
            RemoveIter end(endW,gid);
            StlToGecodeValIterator<RemoveIter> rem(begin,end);
            // remove nodes
            return g._nodesOut(home,rem);
        }
    };

/*
 *
 * Upper bound pruning for connected with incremental boundsGraphs
 *
 */
template <class GView>
    UBConnected<GView>::UBConnected(Space* home, bool share,
        UBConnected& p): Propagator(home,share,p), bg(NULL){
            g.update(home,share,p.g);
        }
template <class GView>
    UBConnected<GView>::~UBConnected(void){
        if (bg) { delete bg; }
        g.cancel(this, Gecode::Graph::PC_GRAPH_ANY);
    }
template <class GView>
    UBConnected<GView>::UBConnected(Space* home, GView &g):
        Propagator(home,true), g(g), bg(NULL){
            g.subscribe(home,this, Gecode::Graph::PC_GRAPH_ANY);
        }
template <class GView>
```

```
    Actor*      UBConnected<GView>::copy(Space* home,bool share){
        return new (home) UBConnected(home,share,*this);
    }
template <class GView>
    PropCost    UBConnected<GView>::cost(void) const{
        return Gecode::PC_QUADRATIC_LO;
    }
template <class GView>
    ExecStatus  UBConnected<GView>::propagate(Space* home){
        if (bg == NULL){
            bg = new UBConnectedBoundsGraphs<GView>(g);
        } else {
            bg->update_bounds();
        }
        GECODE_ME_CHECK(bg->filter_upper_bound(home));
        if (g.assigned())
            return ES_SUBSUMED;
        else {
            return ES_FIX;
        }
    }
template <class GView>
    ExecStatus UBConnected<GView>::post(Space* home, GView &g) {
        (void) new (home) UBConnected(home,g);
        return ES_OK;
    }
template<class GView>
    void connected(Space *home, GView g, IntConLevel){
        GECODE_ES_FAIL(home,UBConnected<GView>::post(home, g));
    }
}}
```

Appendix D

Status of the Implementation

In version 0.9.1 of CP(Graph), available at https://savane.info.ucl.ac.be/svn/cp_graph_map/branches/0.9.1/graph, the following models and constraint propagators are implemented.

D.1 Models and Views

The following graph models and views are implemented:

- The out-adjacency model (also called N-sets) for graph intervals is implemented as the `OutAdjSetsGraphView` in file `view/outadjsets.icc`.
- The nodes and arcs set model (also called 2-sets) for graph intervals is implemented as the `NodeArcSetsGraphView` in file `view/nodearcsets.icc`.
- The single successor graph model for graph intervals is implemented as the `SingleSuccGraphView` in file `view/intsucc.icc`.
- The graph variable data structure is implemented as `GraphVarImpl` in file `var/imp.icc` and `var/imp.cc`. A graph view for dealing with this variable type is implement as `GraphVarView` in file `view/graphvar.icc`.
- Specializations of the node set view over graph domains are implemented in files `nodeset.icc` and `nodesetgraph.icc`.

D.2 Propagators and Support

The following support for propagators and constraint propagators are implemented:

- Scanners and iterator converters between STL iterators and Gecode iterators are available in file `view/iter.icc`.
- Bounds graphs are implemented in file `view/boundsgraphs.icc`.
- A base class for binary graph propagators is available in file `binarysimple.hh`.
- The propagator for the $Complement(G_1, G_2)$ constraint is `ComplementPropag`. It is declared in `binarysimple.hh` and implemented in `binarysimple.icc`. This constraint can be posted by using the templated function `complement(Space * home, GDV1 &g1, GDV2 &g2)`.
- The propagator for the $InducedSubgraph(G_1, G_2)$ constraint is `InducedSubGraphPropag`. It is declared in `binarysimple.hh` and implemented in `binarysimple.icc`. This constraint can be posted by using the templated function `inducedSubgraph(Space * home, GDV1 &g1, GDV2 &g2)`.
- The propagator for the $Subgraph(G_1, G_2)$ constraint is `SubgraphPropag`. It is declared in `binarysimple.hh` and implemented in `binarysimple.icc`. This constraint can be posted by using the templated function `subgraph(Space * home, GDV1 &g1, GDV2 &g2)`.
- The propagator for the $QuasiPath(G, n_1, n_2)$ constraint is `PathDegreePropag`. It is declared and implemented in `path/pathdegree.icc`. This constraint can be posted by using the templated function `pathdegree(Space *home, GView &g, int start, int end)` of `path.hh` .
- The propagator for the $Path(G, n_1, n_2, W, I)$ constraint for the edge-weight case is `PathCostPropag`. It is declared and implemented in `path/path.icc`. This constraint can be posted by using the templated function `path(Space* home, GView &g, int start, int end, const map <pair<int,int>,int> &edgecosts, IntVar w)` of `path.hh`.

- The propagator for the $Path(G, n_1, n_2, W, I)$ constraint for the node-weight case is `PathCostPropagNodes`. It is declared and implemented in `path/path.icc`. This constraint can be posted by using the templated function `path(Space* home, GView &g, int start, int end, const vector<int> &nodecosts, IntVar w, pair<int,int> * hint=NULL)` of `path.hh`. If hint is not null, the value it points is set to the first undetermined arc in the shortest path tree. This hint was used to implement the shortest path heuristic of experiment 7.2.
- The propagator for the $Path(G, n_1, n_2)$ constraint is `PathPropag`. It is declared and implemented in `path/path.icc`. This constraint can be posted by using the templated function `path(Space* home, GView &g, int start, int end)` of `path.hh`.
- The propagator for the $Connected(G)$ constraint is `IncrCheckconnected`. It is declared and implemented in `path/connected.icc`. This constraint can be posted by using the templated function `connected(Space* home, GView &g)` of `path/connected.icc`.
- The propagator for the $Tree(G)$ constraint is `UndirectedTree`. It is declared and implemented in `path/connected.icc`.

Bibliography

[1] I.D. Aron and P. Van Hentenryck. A constraint satisfaction approach to the robust spanning tree with interval data. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 18–25, 2002.

[2] F. Azevedo and P. Barahona. Modelling digital circuits problems with set constraints. In J. Lloyd et al., editor, *Proceedings of the 1st International Conference on Computational Logic (CL)*, pages 414–428. Springer, 2000.

[3] N. Beldiceanu. Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005.

[4] N. Beldiceanu, M. Carlsson, J-X. Rampon, and C. Truchet. Graph invariants as necessary conditions for global constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, pages 92–106. Springer, 2005.

[5] N. Beldiceanu, M. Carlsson, J-X. Rampon, C. Truchet, S. Demassey, and T. Petit. Using graph properties for global constraints for necessary conditions and filtering. In *Proceedings of the 5th Swedish Constraint Network Workshop (SweConsNet)*, 2006.

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

[7] N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *Proceedings of the 2nd International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3524 of *LNCS*, pages 64–78. Springer, 2005.

[8] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Uppsala Universitet, April 2006.

[9] N. Beldiceanu, I. Katriel, and S. Thiel. Filtering algorithms for the same constraint. In *Proceedings of the 1st International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3011 of *LNCS*, pages 65–79. Springer, 2004.

[10] N. Beldiceanu and E. Poder. The period constraint. In *Proceedings of 20th International Conference on Logic Programming (ICLP)*, volume 3132 of *LNCS*, pages 329–342. Springer, 2004.

[11] F. Benhamou. Heterogeneous constraint solving. In *Proceedings of ALP'96*, pages 62–76. Springer-Verlag, 1996.

[12] C. Bessiere, E. Hebrard, B. Hnich, and Z. Kiziltan. The range constraint: Algorithms and implementation. In *Proceedings of the Third International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'06)*, volume LNCS 3990, 2006.

[13] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *LNCS*, pages 138–152. Springer, 2004.

[14] C. Bessiere and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 398–404, 1997.

[15] C. Bessière and JC. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 309–315, 2001.

[16] G. Birkhoff. Lattice theory. In *American Mathematical Society Colloquium Publications*, volume 25(3), 1967.

[17] The Boost graph library. `http://www.boost.org/libs/graph/`.

[18] O. Boruvka. O jistém problému minimálním. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.

[19] E. Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes.* PhD thesis, Université Paris 13, Paris, 1999.

[20] K.N Brown, P. Prosser, J-C. Beck, and C.W. Wu. Exploring the use of constraint programming for enforcing connectivity during graph generation. In *Proceedings of the 5th Workshop on Modelling and Solving Problems with Constraints, held at the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 2005. http://lia.deis.unibo.it/~zk/Pubs/MSPC2005proceedings.pdf.

[21] H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. In *Proceedings of 10e Journée nationale sur la résolution pratique de problèmes NP-complets (JNPC)*, pages 107–121, 2004.

[22] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Proceedings of the 14th International Conference on Logic Programming.* The MIT Press, 1997.

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

[24] A. Conrad, T. Hindrichs, H. Morsy, and I. Wegener. Solution of the knight's hamiltonian path problem on chessboards. *Discrete Applied Mathematics*, 50:125–134, 1994.

[25] T.T. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 1990.

[26] B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In *Descriptive complexity and finite models*, pages 38–62, Providence, 1997. American Mathematical Society .

[27] D. Croes. *Recherche de chemins dans le réseau métabolique et mesure de la distance métabolique entre enzymes.* PhD thesis, ULB, Bruxelles, 2005.

[28] C. Demetrescu. Engineering (dynamic) shortest path algorithms (a round trip between theory and experiments). In *DIKU Summer School on Shortest Paths*, 2005.

[29] Y. Deville. CLP(BioNet) : Towards a CLP framework for the analysis of biochemical networks. In *Invited talk at the 3rd Swedish Constraint Network workshop (SweConsNet)*, 2004.

[30] Y. Deville, G. Dooms, S. Zampelli, and P. Dupont. Cp(graph+map) for approximate graph matching. In *1st International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD)*, 2005.

[31] D. Diaz and P. Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 6, 2001.

[32] E.W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. Springer Berlin, 1959.

[33] M. Dincbas and P. Van Hentenryck. Extended unification algorithms for the integration of functional programming into logic programming. *Journal of Logic Programming (JLP)*, 4 (3):199–227, 1987.

[34] B. Dixon, M. Rauch, and R.E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[35] G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *Proceedings of 5èmes Journées Ouvertes Biologie Informatique Mathématiques (JOBIM)*, 2004.

[36] G. Dooms, Y. Deville, and P. Dupont. A Mozart implementation of CP(BioNet). In P. Van Roy, editor, *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *LNCS*, pages 237–250. Springer, 2004.

[37] G. Dooms, Y. Deville, and P. Dupont. Recherche de chemins contraints dans les réseaux biochimiques. In F. Mesnard, editor, *Programmation en logique avec contraintes, actes des Journées Francophones de Programmation en Logique et de programmation par Contraintes (JFPLC)*, pages 109–128. Hermes Science, June 2004.

[38] I. Dumitrescu. *Constrained Path and Cycle Problems.* PhD thesis, University of Melbourne, 2002.

[39] J. Dundacek. Constraint logic programming for graphs, June 1996. Master's thesis.

[40] D. Eppstein. Finding the k smallest spanning trees. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *LNCS*, pages 38–47. Springer, 1990.

[41] D. Eppstein. Setting parameters by example. *SIAM Journal on Computing*, 32(3):643–653, 2003.

[42] F. Focacci, A. Lodi, and M. Milano. Solving tsp with time windows with constraints. In *Proceedings of the 16th International Conference on Logic Programming (ICLP)*, 1999.

[43] F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming: An hybrid approach. In R. Dechter, editor, *Principles and Practice of Constraint Programming*, pages 187–201. SpringerVerlag, 2000.

[44] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, February 2003.

[45] I. Fudos and C.M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics (TOG)*, 16(2):179–216, 1997.

[46] M. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[47] N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 302–309. IEEE Computer Society, 1996.

[48] The gecode library. `http://www.gecode.org/`.

[49] T. Gellermann, M. Sellmann, and R. Wright. Shorter path constraints for the resource constrained shortest path problem. In *Proceedings of the 2nd International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3524 of *LNCS*, 2005.

[50] I.P. Gent, P. Prosser, B.M. Smith, and W. Wei. Supertree construction with constraint programming. pages 837–841, 2003.

[51] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 862–871, 2004.

[52] L. Georgiadis and R.E. Tarjan. Finding dominators revisited: extended abstract. In *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 869–878. Society for Industrial and Applied Mathematics, 2004.

[53] C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *Proceedings of the 3rd Workshop on Constraint Logic Programming (WCLP'93)*, 1993.

[54] C. Gervet. *Set Intervals in Constraint-Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de Franche-Compté, 1995.

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

[56] C. Gervet and P. Van Hentenryck. Length-lex ordering for set CSPs. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006.

[57] J.L. Gross and J. Yellen, editors. *Handbook of graph theory*. Crc Press, 2004.

[58] M. Grönkvist. A constraint programming model for tail assignment, integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In *Proceedings of the 1st International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3011 of *LNCS*. Springer, 2004.

[59] S. Gualandi and B. Tranchero. Concurrent constraint programming-based path planning for uninhabited air vehicles. In *Proceedings of SPIE's Defense & Security Symposium*, 2004.

[60] W.W. Hager and Y. Krylyuk. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics*, 12(4):500–523, 1999.

[61] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. 2:100–107, 1968.

[62] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 607–615, 1995.

[63] P. J. Hawkins, V. Lagoon, and P. J. Stuckey. Solving set constraint satisfaction problems using robdds. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[64] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[65] P. Van Hentenryck. Constraint and integer programming in OPL. *INFORMS Journal on Computing*, 14(4):345–372, 2002.

[66] D. S. Hochbaum and A. Chen. Performance analysis and best implementations of old and new algorithms for the open - pit mining problem. *Operations Research*, page 40. to appear.

[67] R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics.* Elsevier, 1992.

[68] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[69] H. Jeong, B. Tombor, R. Albert, Z.N. Oltvai, and A.L. Barabasi. The large-scale organization of metabolic networks. *Nature*, 406:651–654, 2000.

[70] David R. Karger, R. Motwani, and G.D.S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[71] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[72] I. Katriel. Algorithmic topics in constraint programming: Course notes, 2005.

[73] B.J. Kim, C.N. Yoon, S.K. Han, and H. Jeong. Path finding in scale-free networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 65:27101–27104, 2002.

[74] V. King. A simpler minimum spanning tree verification algorithm. In *Proceedings of the 4th International Workshop on Algorithms and Data*

Structures (WADS), volume 955 of *LNCS*, pages 440–448. Springer, 1995.

[75] J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[76] J.L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[77] B. Legeard, F. Ambert, and H. Zidoum. CLPS: Un langage de PLC ensembliste. In *Journées Prototypes, Journées Francophones de Programmation en Logique et de programmation par Contraintes (JF-PLC)*, 1995.

[78] C. Lepape, L. Perron, J-C. Régin, and P. Shaw. A robust and parallel solving of a network design problem. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *LNCS*, pages 633–648. Springer, 2002.

[79] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[80] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[81] K. Minoru, G. Susumu, K. Shuichi, and N. Akihiro. The KEGG databases at GenomeNet. *Nucleic Acids Research*, 30(1):42–46, 2002.

[82] T. Müller. Practical investigation of constraints with graph views. In *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL)*, 1999.

[83] T. Müller and M. Müller. Finite set constraints in Oz. In F. Bry, B. Freitag, and D. Seipel, editors, *Proceedings of the 13th Workshop Logische Programmierung*, pages 104–115, 1997.

[84] Van Hentenryck P., Saraswat V., and Deville Y. The design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3), 1998.

[85] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows. *Transportation Science*, 32:12–29, 1996.

[86] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-ackermann time. In *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC)*, volume 3827 of *LNCS*, pages 964–973. Springer, 2005.

[87] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[88] R.C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, November 1957.

[89] J-F. Puget. A C++ implementation of CLP. In *Proceedings of the 2nd Singapore International Conference on Intelligent Systems*, 1994.

[90] J.F. Puget. Finite set intervals. In *Proceedings of the Workshop on Set Constraints, held in Conjunction with the 2nd International Conference on Constraint Programming (CP)*, 1996.

[91] L. Quesada. mail to g. dooms, May 2006.

[92] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL).* Springer, 2006.

[93] M.J. Sternberg R. Alves, R.A. Chaleil. Evolution of enzymes in metabolism: a network perspective. *Journal of Molecular Biology*, 320:751–770, 2002.

[94] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science A*, 158:233–277, May 1996.

[95] J. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17:200–207, 1970.

[96] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 184–191. ACM Press, 2004.

[97] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[98] D.E. Rutherford. *Introduction to Lattice Theory*. Hafner Publishing Company, New-York, 1965.

[99] J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 362–367, 1994.

[100] J-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, pages 209–215, 1996.

[101] J-C. Régin. The symmetric alldiff constraint. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 420–425, 1999.

[102] J-C. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7 (3-4):387–405, 2002.

[103] J-C. Régin. Tutorial at the 10th international conference on principles and practice of constraint programming (cp), 2004. http://www.constraint-programming.com/people/regin/papers/modelincp.pdf.

[104] A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *LNCS*, pages 604–618. Springer, 2004.

[105] C. Schulte. Programming constraint services. In *Lecture Notes in Artificial Intelligence*. Springer, 2002.

[106] C. Schulte and P.J. Stuckey. Speeding up constraint propagation. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *LNCS*, pages 619–633. Springer, 2004.

[107] C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In *Proceedings of the 5th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)*, 2005.

[108] Christian Schulte and Peter J. Stuckey. When Do Bounds and Domain Propagation Lead to the Same Search Space? *Transactions on Programming Languages and Systems*, 27(3):388–425, May 2005.

[109] M. Sellmann. Cost-based filtering for shorter path constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *LNCS*, pages 694–708. Springer, 2003.

[110] M. Sellmann. *Reduction Techniques in Constraint Programming and Combinatorial Optimization*. PhD thesis, University of Paderborn, 2003.

[111] J. G. Siek, L-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 399–414. ACM Press, 1999.

[112] B. Smith and S. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 646–654, 1995.

[113] G. Smolka. The Oz programming model. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.

[114] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[115] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[116] R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[117] S. Thiel. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. PhD thesis, Universität des Saarlandes, 2004.

[118] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, L. Wernisch, D. Gilbert, and S.J. Wodak. Representing and analyzing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10):921–35, 2000.

[119] J. van Helden, L. Wernisch, D. Gilbert, and S.J. Wodak. Graph-based analysis of metabolic networks. In *Bioinformatics and genome analysis*, pages 245–274. Springer, 2002.

[120] P. Van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, 1996.

[121] W.-J. van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4–5):347–373, 2005.

[122] W.J. van Hoeve. The alldifferent constraint: A survey. In *Proceedings of the 6th Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[123] J. van Leeuwen. *Handbook of Theoretical Computer Science Vol. A: Algorithms and Complexity.* Elsevier, 1990.

[124] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, 1997.

[125] J. Westbrook and R.E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[126] S. Zampelli, Y. Deville, and P. Dupont. Approximate constrained subgraph matching. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 832–836. Springer, 2005.