Consistency of Constraint Networks
Induced by Automaton-Based
Constraint Specifications

Maria Andreina Francisco Rodriguez
Abstract

Consistency of Constraint Networks Induced by Automaton-Based Constraint Specifications

Maria Andreina Francisco Rodriguez

In this work we discuss the consistency of constraints for which the set of solutions can be recognised by a deterministic finite automaton. Such an automaton induces a decomposition of the constraint into a conjunction of constraints. Since the level of filtering for the conjunction of constraints is not known, at any point during search there might be only one possible solution but, since all impossible values might not have yet been removed, we could be wasting time looking at impossible combinations of values.

The so far most general result is that if the constraint hypergraph of such a decomposition is Berge-acyclic, then the decomposition provides hyper-arc consistency, which means that the decomposition achieves the best possible filtering.

We focus our work on constraint networks that have alpha-acyclic, centred-cyclic or sliding-cyclic hypergraph representations. For each of these kinds of constraints networks we show systematically the necessary conditions to achieve hyper-arc consistency.
For my loved ones, for being there all the way...
Acknowledgements

I would like to express my gratitude to Nicolas Beldiceanu and Romuald Debruyne of École des Mines de Nantes, France, for fruitful discussions on the topic.

I am also heartily thankful to my supervisor, Justin Pearson, whose encouragement, supervision, support and occasional cup of coffee made this work possible.

This thesis would not have been possible without Pierre Flener, whose teaching skills made me love constraint programming from the very beginning of his course. I’m really thankful he gave me the opportunity to work with him and Justin Pearson.

Last but not least, I would like to thank my parents and my boyfriend who have been caring and supportive every single day.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Constraints</td>
<td>4</td>
</tr>
<tr>
<td>3 The <em>Automaton</em> Constraint</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Modelling Constraints with Automata</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Modelling Constraints with Predicate Automata</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Modelling Constraints with Counter Automata</td>
<td>13</td>
</tr>
<tr>
<td>4 Hypergraphs</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Definition</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Acyclicity for Hypergraphs</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Other Hypergraph Definitions</td>
<td>18</td>
</tr>
<tr>
<td>5 $\alpha$-Acyclic Automata</td>
<td>20</td>
</tr>
<tr>
<td>5.1 The Constraint Hypergraph</td>
<td>20</td>
</tr>
<tr>
<td>5.2 Achieving HAC for $\alpha$-Acyclic Networks</td>
<td>21</td>
</tr>
<tr>
<td>6 Centred-Cyclic Automata</td>
<td>28</td>
</tr>
</tbody>
</table>
7 Sliding-Cyclic Automata
8 Conclusion
Bibliography
List of Figures

2.1 A sudoku puzzle ......................................................... 5
2.2 A partial solution to the sudoku puzzle in Figure 2.1 ......... 6

3.1 A deterministic finite state automaton for the regular expres-
sion $1^*2(1 + 2 + 3)^*$ .................................................. 9
3.2 A $k$-ary predicate automaton (with $k = 2$) for the $<_\text{lex}$ constraint 11
3.3 Predicate automaton with one counter .......................... 14

4.1 Example of a hypergraph .............................................. 17
4.2 Part of a Berge-acyclic constraint hypergraph of a predicate-
automaton-induced decomposition (with $k = 1$) ................. 18
4.3 A simple Berge-cyclic hypergraph ................................. 18

5.1 $\alpha$-acyclic constraint hypergraph induced by an automaton 21
5.2 Transition constraints of an $\alpha$-acyclic constraint hypergraph
induced by a predicate automaton ................................. 22
5.3 Signature constraints of an $\alpha$-acyclic constraint hypergraph
induced by an automaton ........................................ 23
5.4 Constraint hypergraph induced by the counter automaton of
the $\text{ith}_\text{-pos}_\text{different}_\text{from}_\text{zero}(\text{ith}, \text{pos}, (v_1, \ldots, v_n))$ constraint,
with counters $c$ and $d$ [BCR07] ................................. 24
5.5 Example of an $\alpha$-cyclic hypergraph ......................... 25
5.6 Predicate automaton of the $\text{ith}\_\text{pos}\_\text{different}\_\text{from}\_\text{zero}(\text{ith}, \text{pos}, \langle v_1, \ldots, v_n \rangle)$ constraint, with counters $c$ and $d$ [BCR07]. The labels 0 and 1 correspond to the values of the signature variables. ....... 26

6.1 Centred-cyclic constraint hypergraph induced by an automaton 29
6.2 Transition constraints of a centred-cyclic constraint hypergraph induced by an automaton .................. 30
6.3 Signature constraints of a centred-cyclic constraint hypergraph induced by an automaton .................... 31
6.4 Constraint hypergraph induced by the predicate automaton of the $\text{Maximum}(M, \langle v_1, \ldots, v_n \rangle)$ constraint .................. 32
6.5 Automaton of the $\text{Maximum}(M, \mathcal{V})$ constraint .................. 32
6.6 Underlying automaton of the $\text{Maximum}(M, \mathcal{V})$ constraint ... 33

7.1 Sliding-cyclic constraint hypergraph induced by an automaton 35
7.2 Transition constraints of a sliding-cyclic constraint hypergraph induced by an automaton .................. 36
7.3 Signature constraints of a sliding-cyclic constraint hypergraph induced by an automaton .................... 36
7.4 Automaton of the $\text{Change}(N, \mathcal{V}, <)$ constraint. The labels 0 and 1 correspond to the values of the signature variables. ... 37
7.5 Underlying automaton of the $\text{Change}(N, \mathcal{V}, <)$ constraint ... 37
7.6 Constraint hypergraph induced by the $\text{Change}(N, \mathcal{V}, <)$ constraint ................. 38
Chapter 1

Introduction

Global constraints are an important component in many modern constraint solvers. A global constraint does two things: from the modelling perspective, it allows a modeller to express commonly occurring combinatorial structures; from the solving perspective, it comes with a filtering algorithm that removes impossible domain values during search. There are global constraints for many combinatorial structures, such as scheduling [AB93, BLPN01], packing [CBM08], staff scheduling [BGP03], and so on [BCR07].

Although modern constraint solvers have many global constraints, often a constraint that one is looking for is not there. In the past, the choices where either to reformulate the problem or to write one’s own filtering algorithm.

In [BCP04], a framework is given where a global constraint can be specified in a relatively simple and high-level way by a (deterministic or non-deterministic) finite automaton. The idea is to describe what it means for the constraint to be satisfied in terms of the accepting paths of the automaton. Based on the automaton, the framework decomposes the specified new global constraint into a conjunction of already implemented (global) constraints. These constraints collectively give the semantics of the specified global constraint and provide the filtering.

It is so far known [BCP04, BCDP05] that if the constraint graph of a
decomposition (induced by an automaton) is Berge-acyclic [Ber70], then the
decomposition automatically provides hyper-arc consistency, that is the de-
composition achieves all the filtering that is possible. Beside Berge-acyclicity,
another ten patterns of constraint hypergraph structure are identified in the
current on-line version of the Global Constraint Catalogue\footnote{See http://www.emn.fr/z-info/sdemasse/gccat/sec3.6.5.html} [BCR07], but
little is known about the filtering strength of the (automaton-induced) de-
compositions that satisfy these structures.

In this thesis, we show how an $\alpha$-acyclic constraint hypergraph (see Chap-
ter 5) can be modified to provide hyper-arc consistency. Moreover, we
show that by adding implied constraints the so-called centred-cyclic net-
works of [BCR07] can also be modified to provide hyper-arc consistency
(see Chapter 6). Furthermore, we show that in the same way, adding im-
plied constraints to the sliding-cyclic networks can provide hyper-arc con-
sistency (see Chapter 7). This covers nine of the ten open hypergraph pat-
terns (namely $\alpha$-acyclic(2), $\alpha$-acyclic(3), centred-cyclic(1), centred-cyclic(2),
centred-cyclic(3), sliding-cyclic(1)(1), sliding-cyclic(1)(2), sliding-cyclic(1)(3)
and sliding-cyclic(2)(2)) and includes almost all of automaton-induced de-
compositions in the current on-line version of the Global Constraint Cata-
logue [BCR07] that are now known to provide hyper-arc consistency. The
Global Constraint Catalogue constraints at the moment 15 $\alpha$-acyclic, 19
centred-cyclic, 24 sliding-cyclic and 1 circular-sliding-cyclic constraints. The
circular-sliding-cyclic constraint is the only one not included in this work.

It was already observed in [BCDP05] that an $\alpha$-acyclic constraint hy-
pergraph can be made hyper-arc consistent by making all the constraints
pairwise consistent (see Definition 5.1), but no algorithm was given. Here
we show the connection (see Theorem 5.1) between achieving pairwise con-
sistency and doing a reachability analysis on an automaton.

There is also a large body of related work (e.g., [BKNW09, BHH+07,
BKN+10, QW07]) on decomposing global constraints to achieve hyper-arc
consistency. The work in this project can be seen as a more systematic approach to providing hyper-arc consistency via decompositions.
Chapter 2

Constraints

Constraint programming is an alternative approach which relies on a combination of techniques that deal with reasoning and computing. Its central notion are constraints. Informally, a constraint on a sequence of variables is a relation on their domains [Apt03]. It can be viewed as a requirement that states which combinations of values from the variables domain are admitted. In consequence, a constraint satisfaction problem consists on a finite set of constraints, each on a subset of a given set of variables.

We represent a constraint as a pair $R(S)$, where $S$ is a tuple of decision variables $\langle w_1, \ldots, w_n \rangle$ and $R$ is a set of tuples of length $n$ from some given domain. The tuple $S$ is often referred to as the scope of the constraint. We assume that in all scopes all variables are distinct.

Example 2.1. Consider a situation where the values taken by the variables on the sequence $x_1, x_2, \ldots, x_n$ must be different from each other. Then in constraint programming that restriction can be represented with the $\text{Alldifferent}(\langle x_1, x_2, \ldots, x_n \rangle)$ constraint, where the name of the constraint $R$ is $\text{Alldifferent}$ and the scope $S$ is the tuple $\langle x_1, x_2, \ldots, x_n \rangle$.

A constraint satisfaction problem is a way of expressing simultaneous requirements for values of variables. It can be defined then as a conjunction
Figure 2.1: A sudoku puzzle

Example 2.2. Consider the well-known Sudoku puzzle. The objective of the game is to fill a $9 \times 9$ grid with digits so that each column, row and each of the nine $3 \times 3$ sub-grids that compose the grid contains distinct digits from 1 to 9. Figure 2.1 shows an example of a Sudoku puzzle. The rules of the game can be specified as a set of 27 alldifferent constraints: 1 for each row, one for each column and one for each sub-grid.

A solution to a constraint $R(S)$ is some assignment to variables, $w_1 = d_1, \ldots, w_n = d_n$, such that the tuple $(d_1, \ldots, d_n)$ belongs to $R$. A solution to a constraint problem is an assignment to all its variables that satisfies all its constraints simultaneously.

We will make use of the notion of a partial solution. This may be defined as a mapping from some subset of the variables of the problem to values on its domains that do not violate the constraints of the problem. For example, Figure 2.2 shows a partial solution of the Sudoku problem in Figure 2.1.
Figure 2.2: A partial solution to the sudoku puzzle in Figure 2.1

Given a constraint $R(S)$ and a domain $D_i$ for each variable $w_i$, a supporting tuple for a value $d_k$ in the domain $D_k$ of variable $w_k$ is a tuple $(d_1, \ldots, d_k, \ldots, d_n)$ satisfying $R$ such that for all $i \neq k$ the value $d_i$ is in $D_i$. A given set of domains is said to be hyper-arc consistent [Apt03] for a constraint, if for every variable and every domain value there is a supporting tuple. Informally, every element $d_i$ of a domain $D_i$ participates in some solution to the constraint. The implementation in [BCP04] of the transition constraints prunes the domains of the variables until hyper-arc consistency is achieved. A given set of domains is said to be hyper-arc consistent or generalized arc consistent (GAC) for a constraint problem or a set of constraints if every value in the domain of every variable participates in some solution. Namely, a variable is hyper-arc consistent with a constraint if every value of the variable can be extended to the other variables of the constraint in such a way the constraint is satisfied. In general, hyper-arc consistency of the individual constraints of a problem is not sufficient to ensure hyper-arc consistency of the whole problem.

**Definition 2.1.** Let $Y$ and $Z$ be sets of attributes with $Z \subseteq Y$. Let $t$ be a tuple over $Y$, and let $r$ be a relation instance over $Y$. The projection onto $Z$ of $t$, denoted $t[Z]$, is the restriction of $t$ to $Z$. The projection onto $Z$ of $r$,
denoted $\pi_Z(r)$, is the set $\{t[Z] | t \in r\}$.

**Definition 2.2.** Let $r$ and $s$ be relations, with attributes $R$ and $S$, respectively. Then $r$ and $s$ are consistent if the projections of $r$ and $s$ onto their common attributes are the same [Fag83], that is, if $r[R \cap S] = s[R \cap S]$. 
Chapter 3

The Automaton Constraint

The Automaton\((A,V)\) constraint [BCP04] holds if the constraint described by the automaton \(A\) holds for the sequence of decision variables \(V\), that is if \(A\) accepts the sequence of values of \(V\). We define the Automaton constraint in three stages: first its particular case that is also known as the Regular constraint [Pes04], and then two orthogonal extensions, namely predicate automata and counter automata.

3.1 Modelling Constraints with Automata

Definition 3.1. A deterministic finite-state automaton (DFA) is a tuple \(\langle Q, \Sigma, S, F, \delta \rangle\) where \(Q\) is the set of states; \(\Sigma\) is the alphabet; \(S\) is a subset of \(Q\) denoting the start states; \(F\) is a subset of \(Q\) denoting the accepting states; and \(\delta\) is a function from \(Q \times \Sigma\) to \(Q\) denoting the transition function.

If \(\delta(\rho, \sigma) = \rho'\), then we say that there is a transition from state \(\rho\) to state \(\rho'\) that consumes alphabet symbol \(\sigma\); this is often written as:

\[
\rho \xrightarrow{\sigma} \rho'
\]

A sequence \(\sigma_1 \sigma_2 \cdots \sigma_{n-1} \sigma_n\) of alphabet symbols is accepted by the automaton
Figure 3.1: A deterministic finite state automaton for the regular expression $1^*2(1 + 2 + 3)^*$

*if there is a chain of transitions*

$$\rho_0 \xrightarrow{\sigma_1} \rho_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{n-1}} \rho_{n-1} \xrightarrow{\sigma_n} \rho_n$$

*such that $\rho_0 \in S$ and $\rho_n \in F$.*

One often uses pictures to define finite-state automata. For example, in Figure 3.1, we define an automaton with two states, $Q = \{\rho_s, \rho_t\}$, represented by circles, and an alphabet of three symbols, $\Sigma = \{1, 2, 3\}$, on the transitions. A start state is indicated by an arrow from the ‘start’ symbol, and an accepting state is represented by a double circle. The transition function is represented by the labelled arrows, that is $\delta(\rho, \sigma) = \rho'$ if there is an arrow from $\rho$ to $\rho'$ labelled with $\sigma$. For each state, there is one outgoing arrow per alphabet symbol; any missing arrow is assumed to go to an implicit non-accepting state, on which there is a self-looping arrow for every symbol of the alphabet, so that no accepting state is reachable from that state. For example, in Figure 3.1, the missing transition from state $\rho_s$ on symbol 3 goes to such an implicit non-accepting state.

The symbol sequences accepted by an automaton form a regular language. Hence any constraint (on a sequence of decision variables) whose extensional definition forms a regular language can be described by an automaton. In fact, any constraint on a finite sequence of decision variables that range over finite domains can be described by an automaton, since every finite language
is a regular language. For instance, the automaton in Figure 3.1 accepts the
language of the regular expression $1^*2(1 + 2 + 3)^*$.

The Automaton constraint discussed so far can be implemented either
via a specialised propagator [Pes04], or via decomposition into a conjunction
of constraints [BCP04]. For a given automaton, define a new constraint
$T(\rho, \rho', \sigma)$ extensionally by the following set:

$$\{\langle \rho, \rho', \sigma \rangle \mid \rho \xrightarrow{\sigma} \rho'\} \tag{3.1}$$

That is, $T(\rho, \rho', \sigma)$ is satisfied whenever there is a transition from state $\rho$ to
state $\rho'$ that consumes symbol $\sigma$. An Automaton constraint on a sequence
of $n$ decision variables, $v_1, \ldots, v_n$, is then decomposed into the following
conjunction of $n + 2$ constraints, called the transition constraints:

$$q_0 \in S \land T(q_0, q_1, v_1) \land \cdots \land T(q_{n-1}, q_n, v_n) \land q_n \in F \tag{3.2}$$

where $q_0, q_1, \ldots, q_{n-1}, q_n$ are new decision variables, called the state variables,
with domain $Q$. For contrast, we call $v_1, \ldots, v_n$ the problem variables.

The implementation of [BCP04] actually works unchanged for
non-deterministic finite-state automata, but we have elected to restrict our
focus to deterministic ones, in order to ease the notation.

### 3.2 Modelling Constraints with Predicate Automata

The automata in [BCP04] are more powerful than those in [Pes04]: when
used to describe constraints, the labels can be predicates, and all predicates
must be satisfied on an accepting path.

The definition presented here is parametrised by a suitable set of pred-
icates. Let $\text{Pred}_k$ be a set of $k$-ary predicates in some suitable language.
That is, a predicate takes a vector, \( V \), of \( k \) values and it is either true or false.

**Definition 3.2.** A \( k \)-ary-predicate DFA is a tuple \( \langle Q, \Sigma, \phi, S, F, \delta \rangle \), where \( Q, \Sigma, S, F, \) and \( \delta \) are exactly as for a deterministic finite-state automaton, and \( \phi \) is a function from \( \Sigma \) to \( \text{Pred}_k \). A sequence of \( k \)-ary vectors of values \( V_1 V_2 \cdots V_{n-1} V_n \) is accepted by the automaton if there exists a chain of transitions

\[
\rho_0 \xrightarrow{\sigma_1} \rho_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n-1}} \rho_{n-1} \xrightarrow{\sigma_n} \rho_n
\]

such that \( \rho_0 \in S \), \( \rho_n \in F \), and \( \phi(\sigma_i)(V_i) \) is true for all \( 1 \leq i \leq n \). Such a chain of transitions will often be written as

\[
\rho_0 \xrightarrow{\phi(\sigma_1)(V_1)} \rho_1 \xrightarrow{\phi(\sigma_2)(V_2)} \cdots \xrightarrow{\phi(\sigma_{n-1})(V_{n-1})} \rho_{n-1} \xrightarrow{\phi(\sigma_n)(V_n)} \rho_n
\]

Again, we often define \( k \)-ary predicate automata by pictures. The convention is similar to normal regular automata, except that the transition labels are predicates. We assume that each distinct predicate is associated with a distinct symbol of the alphabet \( \Sigma \) (as in Figure 3.1), and that the function \( \phi \) from symbols is defined by the predicate labels in the picture. For example, in Figure 3.2, the function \( \phi \) could be defined by lambda expressions as follows: \( \phi(1) = \lambda x, y : x = y \), \( \phi(2) = \lambda x, y : x < y \), and \( \phi(3) = \lambda x, y : x > y \). Consider the binary constraint (\( k = 2 \)) that vector \( \mathcal{A} \) be lexicographically less than vector \( \mathcal{B} \), which is denoted by \( \mathcal{A} \leq_{\text{lex}} \mathcal{B} \). For \( \mathcal{A} = \langle 1, 2, 5, 6 \rangle \) and
\( B = \langle 1, 3, 4, 7 \rangle \), the sequence \( \langle 1, 1 \rangle \langle 2, 3 \rangle \langle 5, 4 \rangle \langle 6, 7 \rangle \) of binary vectors, obtained by zipping \( A \) and \( B \) together, is accepted by the binary predicate automaton in Figure 3.2 because the transition chain

\[
\rho_s \xrightarrow{1=1} \rho_s \xrightarrow{2<3} \rho_t \xrightarrow{5>4} \rho_t \xrightarrow{6<7} \rho_t
\]

ends in the accepting state \( \rho_t \).

Given a predicate automaton \( \langle Q, \Sigma, \phi, S, F, \delta \rangle \), the automaton \( \langle Q, \Sigma, S, F, \delta \rangle \) is referred to as the underlying automaton of the predicate automaton. For example, the automaton in Figure 3.1 is the underlying automaton of the predicate automaton in Figure 3.2.

In [BCP04], constraints defined by predicate automata are implemented with the help of reification. The constraint \( T \) defined in (3.1) is used for the following transition constraints:

\[
q_0 \in S \land T(q_0, q_1, s_1) \land \cdots \land T(q_{n-1}, q_n, s_n) \land q_n \in F
\]

These transition constraints are like (3.2), but are expressed for new decision variables \( s_1, \ldots, s_n \), which are connected as follows to the problem variables via the predicates and reification: given an \( n \)-length sequence \( V_1, \ldots, V_n \) of \( k \)-ary vectors of problem variables, we add the following constraints, called the signature constraints:

\[
\bigwedge_{\sigma \in \Sigma} (s_i = \sigma \Leftrightarrow \phi(\sigma)(V_i))
\]

for all \( 1 \leq i \leq n \), where the \( s_i \) are called the signature variables, with domain \( \Sigma \). Hence \( \text{Pred}_k \) contains whatever can be implemented as reified constraints in the underlying constraint solver. For example, in Figure 3.2, the binary predicate automaton on the two vectors \( A = \langle a_1, \ldots, a_n \rangle \) and \( B = \langle b_1, \ldots, b_n \rangle \)
requires the transition constraints (3.3) and the signature constraints

\[(s_i = 1 \Leftrightarrow a_i = b_i) \land (s_i = 2 \Leftrightarrow a_i < b_i) \land (s_i = 3 \Leftrightarrow a_i > b_i)\]

for all \(1 \leq i \leq n\).

### 3.3 Modelling Constraints with Counter Automata

While the class of constraints that can be described by (predicate) automata is very large (currently, 63 of the 354 constraints of the on-line version of the Global Constraint Catalogue [BCR07] are described that way), it is often the case that (predicate) automata are very large or specific to a problem instance. The second extension in [BCP04] is the use of counters that are initialised at the start and evolve through counter-updating operations coupled to the transitions of the automaton. Such counter automata allow the capture of non-regular languages and yield (even for regular languages) automata that are much smaller if not instance-independent (and currently enable another 57 constraints of the catalogue to be described succinctly or generically). The two extensions are orthogonal and can be composed, so we define this second extension in isolation.

Again, we give a definition that is parametric on the class of counter-updating functions. Let \(\text{Update}_\ell\) be a set of \(\ell\)-ary counter-updating functions. That is, given a function \(\psi \in \text{Update}_\ell\) and a vector of counters \(\mathcal{C} \in \mathbb{N}^\ell\), we have that \(\psi(\mathcal{C})\) is a new vector in \(\mathbb{N}^\ell\).

**Definition 3.3.** An \(\ell\)-ary counter DFA is a tuple \(\langle Q, \Sigma, \mathcal{C}_0, S, F, \delta \rangle\) where \(Q, \Sigma, S,\) and \(F\) are exactly as for a deterministic finite-state automaton; vector \(\mathcal{C}_0\) has the initial values of the \(\ell\) counters; and \(\delta\) is a function from
$v_i \neq 0, \{c = c + 1\}$

\[ \text{start} \xrightarrow{} \rho_s : \phi \quad \text{where } \phi : c > 0 \]

Figure 3.3: Predicate automaton with one counter

$Q \times \Sigma$ to $Q \times \text{Cupate}_\ell$. If $\delta(\rho, \sigma) = (\rho', \psi)$ and $\psi(C) = C'$, then we write

$$(\rho, C) \xrightarrow{\sigma} (\rho', C')$$

A sequence $\sigma_1 \sigma_2 \cdots \sigma_{n-1} \sigma_n$ of alphabet symbols and the $\ell$-ary vector of counters $C$ are accepted by the automaton if there is a chain of transitions

$$(\rho_0, C_0) \xrightarrow{\sigma_1} (\rho_1, C_1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n-1}} (\rho_{n-1}, C_{n-1}) \xrightarrow{\sigma_n} (\rho_n, C)$$

such that $\rho_0 \in S$ and $\rho_n \in F$.

Unlike the counter automata in theoretical computer science (see, e.g., [LS05]), the counter automata here do not have access to the values of the counters during a run, but the values of the counters are updated by the transitions. This allows quite complicated constraints to be specified, especially constraints that concern the cardinality of certain sets.

In [BCP04], counter automata are decomposed into transition constraints that are slightly extended to include information about the values of the counters. Define a new constraint $T(\rho, \rho', C, C', \sigma)$ extensionally by the following set:

$$\{ (\rho, \rho', C, C', \sigma) \mid (\rho, C) \xrightarrow{\sigma} (\rho', C') \}$$

For example, the automaton in Figure 3.3 depicts an automaton with counters.
An Automaton constraint on a sequence of $n$ problem variables, $v_1, \ldots, v_n$, and a vector of $\ell$ counters, $C$, is then decomposed into the following conjunction of $n + 3$ transition constraints:

$$q_0 \in S \land T(q_0, q_1, C_0, C_1, v_1) \land \cdots \land T(q_{n-1}, q_n, C_{n-1}, C_n, v_n) \land q_n \in F \land C = C_n$$

(3.5)

where $q_0, q_1, \ldots, q_{n-1}, q_n$ are state variables, with domain $Q$, while $C_0, C_1, \ldots, C_{n-1}, C_n$ are vectors of integer decision variables, called counter variables, and $C_0$ has the initial values of the counters.
Chapter 4

Hypergraphs

4.1 Definition

Informally, a hypergraph can be seen as a generalization of a graph, where an edge can connect any number of vertices. Formally, a hypergraph [Ber70] is a pair \((V, E)\) where \(E\) is a set of subsets of \(V\). Figure 4.1 shows an example of a hypergraph.

While the edges of an ordinary, undirected graph (without self-loops) are pairs of nodes, hyperedges are arbitrary sets of nodes. In consequence, they can contain an arbitrary number of nodes.

For any hypergraph \((V, E)\), we say that it is connected if for every pair of vertices \(s, t \in V\), there exists a sequence of edges \(e_1, \ldots, e_\ell\) such that \(s \in e_1, t \in e_\ell\), and for all \(1 \leq i < \ell\) we have that \(e_i \cap e_{i+1}\) is non-empty.

4.2 Acyclicity for Hypergraphs

Given that hypergraphs are a generalization of graphs and since acyclic graphs are an important subclass of the later, it follows that acyclic hypergraphs are also an important subclass of hypergraphs.

**Definition 4.1.** A path is an alternating sequence of vertices and edges,
beginning and ending with a vertex, where each vertex is incident to both the edge that precedes it and the edge that follows it in the sequence, and where the vertices that precede and follow an edge are the end vertices of that edge.

A path that starts and ends at the same vertex but otherwise has no repeated vertices or edges, is called a \textit{cycle}. A graph is \textit{acyclic} if it contains no cycles.

Unlike for ordinary, undirected graphs, there are several natural, non-equivalent notions of acyclicity for hypergraphs. We will start by defining here Berge-acyclicity and we will present other degrees of cyclicity in the following chapters.

\textbf{Definition 4.2}. A Berge cycle \cite{BFMY83, Ber70} is a sequence $S_1, x_1, S_2, x_2, \ldots, S_n, x_n, S_{n+1}$ in a hypergraph $H$ such that: $x_1, \ldots, x_n$ are distinct vertices of $H$; $S_1, \ldots, S_n$ are distinct edges of $H$; the edges $S_{n+1}$ and $S_1$ are equal; for all $1 \leq i \leq n$ we have that $x_i$ is in $S_i \cap S_{i+1}$; and $n$ is greater than or equal to 2.

A hypergraph is considered \textit{Berge-cyclic} if it has a Berge cycle; it is considered \textit{Berge-acyclic} otherwise.
Figure 4.2: Part of a Berge-acyclic constraint hypergraph of a predicate-automaton-induced decomposition (with $k = 1$)

Figure 4.3: A simple Berge-cyclic hypergraph

**Example 4.1.** The hypergraph in Figure 4.3 is Berge-cyclic because it contains the Berge cycle $E_1, v_2, E_2, v_3, E_1$. The hypergraph in Figure 4.2 is Berge-acyclic.

In general, if there is a pair of edges of a hypergraph that share more than one node, then the hypergraph is Berge-cyclic.

### 4.3 Other Hypergraph Definitions

We would like to provide here some technical definitions that will be used in the following chapters in order to define other degrees of acyclicity for hypergraphs.

**Definition 4.3.** The set of **partial edges** generated by a set $M$ is obtained by intersecting the edges in set $E$ of a hypergraph $(V, E)$ with a set of vertices
$M \subset V$, that is, the set of edges $\{e \cap M \mid e \in E\} \setminus \{\emptyset\}$. This set of partial edges is said to be a vertex-generated set of partial edges. We sometimes refer to the original edges as full edges, in order to establish a clear difference with partial edges when it is needed.

**Definition 4.4.** Let $H$ be a connected, reduced set of partial (or full) edges of some hypergraph, and let $G$ and $F$ be edges of $H$. Define $Q$ to be $G \cap F$. We say that $Q$ is an articulation set of $H$ if the result of removing $Q$ from every edge of $H$, that is $\{e \setminus Q \mid e \in F\} \setminus \{\emptyset\}$, is not a connected set of partial edges.

The concept of an articulation set is a generalisation of an articulation point in a graph. For example, in Figure 4.3, the set $Q = E_1 \cap E_2$ is an articulation set.

**Definition 4.5.** A block of a reduced hypergraph is a connected vertex-generated set of partial edges with no articulation set. A block is said to be trivial if it contains zero or one edge. A reduced hypergraph is said to be $\alpha$-acyclic if all its blocks are trivial [BFMY83].
Chapter 5

\[\alpha\]-Acyclic Automata

5.1 The Constraint Hypergraph

Given a conjunction of constraints \(R_1(S_1) \land \ldots \land R_m(S_m)\), the set of scopes can be considered as a hypergraph \((\cup_{i=1}^{m} S_i, \{S_i\}_{i=1}^{m})\), where the tuples \(S_i\) are considered as sets. For example, the conjunction of constraints \(T(q_0, q_1, x_1) \land T(q_1, q_2, x_2)\) corresponds to the hypergraph:

\[
(\{q_0, q_1, q_2, x_1, x_2\}, \{\{q_0, q_1, x_1\}, \{q_1, q_2, x_2\}\})
\]

Such a hypergraph is referred to as a constraint hypergraph. In Figure 5.1, there is part of a constraint hypergraph of a predicate-automaton-induced decomposition. The transition constraints (3.3) are highlighted in Figure 5.2 and signature constraints (3.4) are highlighted in Figure 5.3.

It is well-known [GLS00] that the structure of the constraint hypergraph can determine how easy it is to solve a constraint problem. Here we review some of these well-known results and apply them to automaton-induced constraint decompositions. For example, a sufficient method for achieving hyper-arc consistency for a constraint set with a Berge-acyclic constraint hypergraph is to ensure each constraint of the hypergraph is hyper-arc con-
Figure 5.1: $\alpha$-acyclic constraint hypergraph induced by an automaton

sistent [JV88, J91].

In the on-line version of the Global Constraint Catalogue [BCR07], there are currently 38 of 120 constraint automata that induce Berge-acyclic constraint hypergraphs. Nevertheless, it can be seen from Figure 5.4 that, in general, if an automaton has counters, then the resulting hypergraph is Berge-cyclic. However, it is possible for an automaton with one counter and only one state to be manipulated in a way that its constraint hypergraph is transformed into a Berge-acyclic hypergraph by projecting away the state variables [Bel11]; we will see later on (in Theorem 5.1) that this is a special case of a more general result. Next, we generalise the known results to a wider class of automata with counters. More details and examples can be found in [BFMY83].

5.2 Achieving HAC for $\alpha$-Acyclic Networks

Without loss of generality, in the rest of the paper, we only consider constraint satisfaction problems whose constraint hypergraph is connected and reduced. The reduction of a hypergraph is obtained by removing each edge that is
Figure 5.2: Transition constraints of an $\alpha$-acyclic constraint hypergraph induced by a predicate automaton

properly contained in another edge. A hypergraph is called reduced if it is equal to its reduction.

In fact, if the constraint hypergraph is not connected, then it can be separated into its connected components and, since the connected component have disjoint sets of variables, each one of them can be considered as independent subproblems and solved separately. Moreover, if the hypergraph representation is not reduced, it is always possible to construct a new constraint equivalent to those related to the edges that are contained within each other. This way we would generate a constraint problem whose hypergraph representation is equivalent to the reduction of the original problem.

Intuitively, a hypergraph is $\alpha$-acyclic if removing articulation sets leaves the hypergraph disconnected; vertex-generated sets have to be considered so that the right edges are taken into account when looking at connected components.

Example 5.1. Consider again the hypergraph in Figure 4.3. It it is a reduced hypergraph since none of its edges are properly contained in any other edge. The set of edges $\{E_1, E_2\}$ is a connected set of full edges, but if we remove
the articulation set $Q = E_1 \cap E_2 = \{v_2, v_3\}$ from them, then we obtain the set of partial edges $\{F \setminus \{v_2, v_3\} \mid F \in \{E_1, E_2\} \setminus \emptyset\} = \{v_1, v_4\}$. Thus the only blocks are $\{E_1\}$ and $\{E_2\}$. Since each block contains at most one edge, these blocks are trivial.

**Example 5.2.** Consider again the hypergraph in Figure 5.4. It is a reduced hypergraph. It has many articulation sets, for example $\{s_1\}$, $\{s_2\}$, $\ldots$, $\{s_n\}$, $\{q_1, c_1, d_1\}$, $\{q_2, c_2, d_2\}$, $\ldots$, $\{q_{n-1}, c_{n-1}, d_{n-1}\}$. In order to verify that the hypergraph is $\alpha$-acyclic, all its blocks must be checked for triviality. For example, removing the articulation set $\{s_1\}$ from the hypergraph disconnects the partial edge $F_1 \setminus \{s_1\}$, which is a trivial block, from the rest of the hypergraph, which then still has many articulation sets. Removing any of the articulation sets of the form $\{q_i, c_i, d_i\}$ results in two disconnected components that both have articulation sets and hence are not blocks. So, any vertex-generated set of partial edges is either not a block or a trivial block.

**Example 5.3.** Consider the hypergraph in Figure 5.5. It doesn’t have any articulation sets, and in consequence, it is said to be $\alpha$-cyclic.

In the case of constraint sets with $\alpha$-acyclic constraint hypergraphs, it is
Figure 5.4: Constraint hypergraph induced by the counter automaton of the \textit{ith-pos-different-from-zero} constraint, with counters $c$ and $d$ [BCR07]

not straightforward to achieve hyper-arc consistency. But it can be done by using techniques from database theory [BFMY83].

We need some technical definitions. Given a constraint $R(\langle w_1, \ldots, w_n \rangle)$ and a tuple of distinct variables $\langle v_1, \ldots, v_k \rangle$ with $k < n$ such that for all $i$ the variable $v_i$ is equal to some variable $w_j$ in $\langle w_1, \ldots, w_n \rangle$, the projection of $R(\langle w_1, \ldots, w_n \rangle)$ onto $\langle v_1, \ldots, v_k \rangle$, denoted by $\text{proj}_{\langle v_1,\ldots,v_k \rangle}(R)$, is the restriction of $R$ to the variables $\langle v_1, \ldots, v_k \rangle$. Given a pair of variable tuples, $x = \langle x_1, \ldots, x_j \rangle$ and $y = \langle y_1, \ldots, y_\ell \rangle$, let $x \cap y$ denote the tuple that only contains the variables in the set-theoretic intersection, such that they appear in the same order as they appear in $x$ or $y$.

**Definition 5.1.** A pair of constraints, $R_1(S_1)$ and $R_2(S_2)$, are said to be pairwise consistent if $\text{proj}_{S_1 \cap S_2} R_1$ is equal to $\text{proj}_{S_1 \cap S_2} R_2$. A set of constraints is said to be pairwise consistent if every constraint pair in that set is pairwise consistent.

Let $\{R_1(S_1), \ldots, R_m(S_m)\}$ be a set of constraints with an $\alpha$-acyclic constraint hypergraph that are pairwise consistent, and let $R(S)$ be the con-
constraint representing the set of solutions of the conjunction of these constraints, where $S$ is a tuple containing all the variables in $S_1, \ldots, S_m$. It can be shown [BFMY83] that for all $i$ the projection $\text{proj}_{S_i} R(S)$ is equal to $R_i(S_i)$. Thus, all tuples in the constraints $R_i(S_i)$ participate in a solution. This, unlike for general constraint problems, makes the solutions very easy to compute. In particular, if all the constraints are hyper-arc consistent, then the conjunction of the constraints is hyper-arc consistent.

**Theorem 5.1.** Consider the transition constraints (3.5) from a counter-automaton-induced constraint decomposition:

$$ q_0 \in S \land T(q_0, q_1, C_0, C_1, v_1) \land \cdots \land T(q_{n-1}, q_n, C_{n-1}, C_n, v_n) \land q_n \in F \land C = C_n $$

Maintaining these transition constraints pairwise consistent is equivalent to doing reachability analysis on the corresponding automaton.

**Proof.** The transition constraint $T(q_{i-1}, q_i, C_{i-1}, C_i, v_i)$ codes the transition relation of the automaton. Achieving pairwise consistency on the pair $q_0 \in S$ and $T(q_0, q_1, C_0, C_1, v_1)$ forces the domain of $q_1$ and $C_1$ to include only states and counter values that are reachable after consuming one alphabet symbol of the automaton. Thus $T(q_0, q_1, C_0, C_1, v_1)$ can be replaced by a constraint $T_0$, 

---

**Figure 5.5:** Example of an $\alpha$-cyclic hypergraph
\[ v_i \neq 0, \{ c < ith \Rightarrow (d = d + 1, c = c + 1) \} \]

where \( \phi : ith = c, pos = d \)

Figure 5.6: Predicate automaton of the \( ith\_pos\_different\_from\_zero(ith, pos, \langle v_1, \ldots, v_n \rangle) \) constraint, with counters \( c \) and \( d \) [BCR07]. The labels 0 and 1 correspond to the values of the signature variables.

which is a sub-constraint of \( T \) that only contains the tuples corresponding to transitions that can happen in one step. Assume that hyper-arc consistency and pairwise consistency have been achieved up to step \( i \) by the constraints:

\[ q_0 \in S \land T(q_0, q_{i-1}, C_{i-1}, v_{i-1}) \land \cdots \land T(q_{i-1}, q_i, C_i, v_i) \]

By induction, the domains of \( q_i \) and \( C_i \) are all states and counter values that are reachable after \( i \) transitions in the automaton. Thus, achieving pairwise consistency on the pair \( T_i(q_{i-1}, q_i, C_{i-1}, C_i, v_i) \) and \( T(q_i, q_{i+1}, C_i, C_{i+1}, v_{i+1}) \) gives a new constraint \( T_{i+1} \) that contains only the subset of the tuples of \( T_i \) that can take part in a chain of transitions of length \( i + 1 \).

Example 5.4. Consider the constraint

\[ ith\_pos\_different\_from\_zero(ith, pos, \langle v_1, \ldots, v_n \rangle) \] [BCR07]

where \( ith \) is a constant, \( pos \) is a decision variable, and \( \langle v_1, \ldots, v_n \rangle \) is a sequence of decision variables. The constraint \( ith\_pos\_different\_from\_zero \) is satisfied when the element in the position \( pos \) of the sequence \( v \), that is the element \( v_{pos} \), is the \( ith \) non-zero element of the sequence. For example, \( ith\_pos\_different\_from\_zero(2, 4, \langle 0, 1, 0, 3, 5 \rangle) \) is satisfied. Its predicate au-
tomaton with two counters is shown in Figure 5.6 and part of its constraint hypergraph is shown in Figure 5.4. The signature constraints are defined as $s_i = 0 \iff v_i \neq 0$ and $s_i = 1 \iff v_i = 0$. For example, given the values $ith = 2$ and $pos = 4$, the sequence $\langle 0, 1, 0, 3, 5 \rangle$ is accepted by the following chain of transitions:

$$(\rho_s, \langle c = 0, d = 0 \rangle) \xrightarrow{1} (\rho_s, \langle c = 0, d = 1 \rangle) \xrightarrow{0} (\rho_s, \langle c = 1, d = 2 \rangle) \xrightarrow{1} (\rho_s, \langle c = 1, d = 3 \rangle) \xrightarrow{0} (\rho_s, \langle c = 2, d = 4 \rangle) \xrightarrow{0} (\rho_s, \langle c = 2, d = 4 \rangle)$$

As in the proof of Theorem 5.1, we denote the pairwise consistent transition relations obtained from the reachability analysis by $T_i$. The automaton has only one state, $\rho_s$, hence for any $i$ the constraint $T_i(q_{i-1}, q_i, \langle c_{i-1}, d_{i-1} \rangle, \langle c_i, d_i \rangle, s_i)$ is satisfied only when $q_{i-1} = q_i = \rho_s$.

Next, we analyse the counters $c$ and $d$. The analysis of values of the counters $c_i$ can be separated into two disjoint cases: $c_{i-1} \geq ith$ and $c_{i-1} < ith$. For the first case, when $c_{i-1} \geq ith$, the counters $c$ and $d$ are never updated because the guard $c < ith$ of the counter updates is not satisfied. In consequence, $T(q_{i-1}, q_i, \langle c_{i-1}, d_{i-1} \rangle, \langle c_i, d_i \rangle, s_i)$ is satisfied whenever $c_i = c_{i-1}$ and $d_i = d_{i-1}$, regardless of the value of $s_i$. For the second case, when $c_{i-1} < ith$, we have two cases depending on the value of the signature variable $s_i \in \{0, 1\}$. When $s_i = 0$ both counters are updated. Thus $T(q_{i-1}, q_i, \langle c_{i-1}, d_{i-1} \rangle, \langle c_i, d_i \rangle, s_i)$ is satisfied whenever $c_i = c_{i-1} + 1$ and $d_i = d_{i-1} + 1$. When $s_i = 1$ only $d$ is updated, while $c$ remains unchanged. Then $T(q_{i-1}, q_i, \langle c_{i-1}, d_{i-1} \rangle, \langle c_i, d_i \rangle, s_i)$ is satisfied if $c_i = c_{i-1}$ and $d_i = d_{i-1}$.

The constraints derived from the previous reachability analysis can be implemented straightforwardly to achieve hyper-arc consistency for the constraint $ith \_ pos \_ different \_ from \_ zero$. Note that these constraints must be enforced during search in order to maintain pairwise consistency.
Chapter 6

Centred-Cyclic Automata

We now analyse another type of constraint hypergraph that appears in the on-line version of the *Global Constraint Catalogue* [BCR07], namely centred-cyclic hypergraphs. We say that the constraint hypergraph of a predicate-automaton-induced decomposition is *centred-cyclic* if its signature constraints share at least one variable (vertex); see Figure 6.1 for an example. We show that it is possible to ensure hyper-arc consistency for this kind of constraint hypergraph. It can be verified that a centred-cyclic hypergraph is not \( \alpha \)-acyclic. Thus, to achieve hyper-arc consistency, new methods have to be used.

A centred-cyclic constraint decomposition has two groups of constraints: the transition constraints (Figure 6.2) and the signature constraints (Figure 6.3). Considered alone, the transition constraints are Berge-acyclic, while the signature constraints are \( \alpha \)-acyclic. In fact, if the signature constraints share exactly one variable, then they will be Berge-acyclic, and hyper-arc consistency alone of the signature constraints is enough to achieve hyper-arc consistency of the signature constraints. So, using results from the previous section, it is possible to achieve hyper-arc consistency on each group separately, but this is not enough to achieve hyper-arc consistency of the whole decomposition. The transition constraints and the signature constraints only
Consider the group of signature constraints as it was a single constraint. Since it is possible to establish hyper-arc consistency for the whole group, then we can consider this group as a constraint that is actually hyper-arc consistent. Moreover, the same reasoning can be applied to the group of transition constraints. This way we have two constraints that are hyper-arc consistent, and, as we explained in the previous chapter, it would be enough to achieve pairwise consistency between both of them in order to achieve hyper-arc consistency for the whole decomposition.

Using the previous reasoning, in order to ensure hyper-arc consistency for the whole centred-cyclic hypergraph, the projection of both subproblems onto their common variables must be the same: that is, the set of possible values for the sequence of signature variables \( s_1, \ldots, s_n \) must be a solution to both the signature constraints and the transition constraints. Thus it is sufficient to add to the decomposition an implied constraint \( \mathcal{I}(\langle s_1, \ldots, s_n \rangle) \) that is satisfied by a sequence of signature values \( \langle \sigma_1, \ldots, \sigma_n \rangle \) if and only
Figure 6.2. Transition constraints of a centred-cyclic constraint hypergraph induced by an automaton

if the corresponding transition sequence is allowed by the automaton and the signature constraints. This constraint must be enforced during search in order to prune sequences of values that are not allowed by either the signature constraints or the transition constraints, thus maintaining hyper-arc consistency.

Theorem 6.1. Given an automaton-induced decomposition, the constraint $I(\langle s_1, \ldots, s_n \rangle)$ can be computed directly from the underlying automaton.

Proof. Assuming that each predicate of the automaton is satisfiable for some assignment of the problem variables, the signature constraints (3.4) allow all possible values for each signature variable $s_i$. So the only restrictions on the values of the signature variables come from the transition constraints (3.3).

The transition constraints, together with the condition that $q_0$ and $q_n$ respectively belong to the start and accepting state sets, restrict the intermediate values to correspond to a chain of transitions from a start state to
Figure 6.3: Signature constraints of a centred-cyclic constraint hypergraph induced by an automaton

an accepting state. Thus the constraint $I$ must restrict the values of the signature variables to correspond to the edge labels of accepting chains of transitions of the underlying automaton.

By enforcing that only sequences that can be accepted by the automaton are partial solutions to both groups of signature and transition constraints simultaneously, we achieve hyper arc-consistency for the centred-cyclic constraint hypergraphs.

We now illustrate Theorem 6.1 on an example taken from [BCR07].

Example 6.1. The constraint $\text{Maximum}(M, V)$ restricts the decision variable $M$ to be the maximum of the sequence of decision variables $V$. For example, $\text{Maximum}(5, \langle 0, 1, 0, 3, 5 \rangle)$ holds since 5 is the greatest element in the sequence. Its predicate automaton is given in Figure 6.5 and induces the constraint hypergraph in Figure 6.4, which is centred-cyclic. Since the automaton has no counters, the constraint hypergraph of the transition constraints
Figure 6.4: Constraint hypergraph induced by the predicate automaton of the \( \text{Maximum}(M, \langle v_1, \ldots, v_n \rangle) \) constraint

\[
\begin{align*}
M > x & \quad M > x \\
\text{start} & \quad M = x \\
\rho_0 & \quad \rho_1
\end{align*}
\]

Figure 6.5: Automaton of the \( \text{Maximum}(M, V) \) constraint

is Berge-acyclic, as is the constraint hypergraph of the signature constraints, since they only share one variable, namely \( M \). Thus, from Theorem 6.1, the constraint \( \mathcal{I} \) corresponds to chains of transitions from ‘start’ to the accepting state \( \rho_1 \) in the underlying automaton given in Figure 6.6. Every sequence of signature values must contain a value other than 1, because every accepting path in Figure 6.6 has to pass through a transition labelled with a 2. In practice, it had been observed [Bel11] that the \( \text{Maximum} \) constraint behaved as if it was hyper-arc-consistent. In this particular case, since there is only one impossible assignment for the signature variables, after any \( s_i \) was assigned the value of 1, hyper-arc-consistency was achieved.

This result can be generalised to cases where the signature constraints
Figure 6.6: Underlying automaton of the $Maximum(M, \mathcal{V})$ constraint

share more than one variable.
Chapter 7

Sliding-Cyclic Automata

We now analyse another type of constraint hypergraph that appears in the on-line version of the *Global Constraint Catalogue* [BCR07], namely sliding-cyclic hypergraphs. We say that the constraint hypergraph of a predicate-automaton-induced decomposition is *sliding-cyclic* if its signature constraints are pairwise connected by at least one variable (vertex); see Figure 7.1 for an example. We show that it is also possible to ensure hyper-arc consistency for this kind of constraint hypergraph. It can be verified that a sliding-cyclic hypergraph is neither $\alpha$-acyclic nor centred-cyclic. In consequence, to achieve hyper-arc consistency we must use slightly different methods.

We will follow a similar reasoning as the one presented for the centred-cyclic hypergraphs.

A sliding-cyclic constraint decomposition has two groups of constraints: the transition constraints (see Figure 7.2) and the signature constraints (see Figure 7.3). Considered alone, the transition constraints are $\alpha$-acyclic, as are the signature constraints. Therefore, using results from the Chapter 5, it is possible to achieve hyper-arc consistency on each group separately, but this is not enough to achieve hyper-arc consistency of the whole decomposition. We must analyse the union of both groups of constraints. The transition constraints and the signature constraints only overlap in the sequence of
signature variables $s_1, \ldots, s_n$. Then we have here the same situation as with the centred-cyclic constraints presented in Chapter 6.

In order to ensure hyper-arc consistency for the whole hypergraph, the projection of both subproblems onto their common variables must be the same. Thus it is sufficient to add to the decomposition an implied constraint $I(\langle s_1, \ldots, s_n \rangle)$ that is satisfied by a sequence of signature values $\langle \sigma_1, \ldots, \sigma_n \rangle$ if and only if the corresponding transition sequence is allowed by the automaton and the signature constraints. Then we can derive the $I$ constraint applying Theorem 6.1.

**Example 7.1.** The constraint $\text{Change}(N, V, <)$ constraints the decision variable $N$ to be the amount of times that the constraint $<$ holds on consecutive variables of the sequence of decision variables $V$. For example, $\text{Change}(2, \langle 4, 5, 3, 2, 4 \rangle, <)$ holds since $4 < 5$ and $2 < 4$. Its predicate automaton is given in Figure 7.4 and induces the constraint hypergraph in Figure 7.6, which is sliding-cyclic.

Since the automaton has one counter, the constraint hypergraph of the transition constraints is $\alpha$-acyclic. Doing a reachability analysis as explained
Figure 7.2: Transition constraints of a sliding-cyclic constraint hypergraph induced by an automaton

Figure 7.3: Signature constraints of a sliding-cyclic constraint hypergraph induced by an automaton
\[ v_i \neq v_{i+1} \]

\[ \text{start} \xrightarrow{0} \rho_s : \phi \xrightarrow{1} v_i < v_{i+1} \Rightarrow c = c + 1 \]

where \( \phi : N = c \)

Figure 7.4: Automaton of the \( \text{Change}(N, V, <) \) constraint. The labels 0 and 1 correspond to the values of the signature variables.

\[ \text{start} \xrightarrow{0} \rho_s : \phi \xrightarrow{1} \]

where \( \phi : N = c \)

Figure 7.5: Underlying automaton of the \( \text{Change}(N, V, <) \) constraint

in Chapter 5 it can be observed that the automaton has only one state and in consequence domain consistency on the \( c_i \) variables is enough to ensure hyper-arc consistency. Moreover, since the signature constraints only share one variable, namely \( N \), constraint hypergraph of the signature constraints is Berge-acyclic. Berge-acyclicity is enough to warranty hyper-arc consistency.

Thus, from Theorem 6.1, the constraint \( \mathcal{I} \) corresponds to chains of transitions from ‘start’ to the accepting state \( \rho_1 \) in the underlying automaton given in Figure 7.5. Every sequence of signature values must contain exactly \( N \) variables with the value 1, because every accepting path in Figure 7.5 ends with \( N = c \), where \( c \) counts the amount of times the signature constraints take the value 1. In this particular case, all the possible assignments satisfy that \( \sum s_i = N \).

This result can be generalised to cases where the signature constraints share more than one variable, as well as the cases where transition constraints
Figure 7.6: Constraint hypergraph induced by the \( \text{Change}(N, V, <) \) constraint

that share more than one variable.
Chapter 8

Conclusion

We have used the notions and results of hypergraphs to derive properties of constraints given their hypergraph reformulation. In particular, we have shown that a constraint satisfaction problem whose hypergraph reformulation is $\alpha$-acyclic can be manipulated in order to achieve hyper-arc consistency. Also, we have shown a way to decompose centred-cyclic and sliding-cyclic constraint networks and evaluate their consistency.

A natural continuation of this work would be analysing the circular-sliding-cyclic constraint network. This constraint network is the only one not known to provide hyper-arc consistency at the moment.

Moreover, since maintaining pairwise consistency might add some extra constraints to the network, it would be interesting to compare in practice how the hyper-arc consistent networks behave in comparison with the original automata-generated networks. In this line of work, we would like to propose the possibility of extending the consistency algorithm presented in [Pes04].
Bibliography


