Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Theoretische Informatik
Prof. Martin Hofmann, PhD

Bachelor Thesis

Solving Size Constraints Using Graph Representation

Felix Reihl

Bearbeitungszeitraum:	29.05.2013 bis 01.08.2013
Aufgabensteller:	Dr. habil. Andreas Abel

Verfasst von Felix Reihl, geboren am 15. Februar 1991 in Starnberg.

Contents

Abstract

The purpose of this bachelor thesis is to find an algorithm that solves ordinal inequalities derived from size constraints. Such inequalities consist of rigid variables (plus offset), infinity, constants and flexible variables that are functions depending on rigid variables (plus offset). First, this is done for a restricted problem: constraints that include maximums and different dependencies on rigid variables are excluded. Then these restrictions are removed and the algorithm is adapted accordingly. The idea is to map the constraints to a graph, find the reflexive-transitive closure and then search for the solutions starting from known values.

Zusammenfassung

Diese Bachelorarbeit soll einen Algorithmus zum Lösen von ordinalen Ungleichungen aus size constraints finden. Solche Ungleichungen bestehen aus festen Variablen (plus Offset), Unendlich, Konstanten und flexiblen Variablen, die Funktionen von festen Variablen sind (plus Offset). Dafür wird das Problem zuerst eingeschränkt gelöst: Ungleichungen, die Maxima und unterschiedliche Abhängigkeiten von festen Variablen enthalten, werden vorerst nicht betrachtet. Nachdem die Einschränkungen entfernt wurden, wird der Algorithmus entsprechend angepasst. Der Ansatz besteht darin, die Ungleichungen auf einen Graphen abzubilden, dann die reflexiv-transitive Hülle zu bilden und anschließend von bekannten Werten aus die Lösungen zu finden.

1 Preface

1.1 Introduction

In proof assistant languages such as Coq and Agda, syntax-based termination checking is used. Since type-based termination checking provides several advantages, efforts are being made to implement it in proof assistants. A language which uses type-based termination for infinite structures is introduced in Abel and Pientka [2013], but the language expects the size values to be explicitly given in the code.

If these size values are not known, a set of size constraints has to be solved. With only finite structures being permitted, this set is trivially solvable, since it is expressable in Presburger arithmetic (see Enderton [2001, p. 188ff]). If, however, we assume the variables to be ordinal numbers $1, \ldots, \omega$, a more elaborate approach is needed.

The inference algorithm described in Barthe et al. [2006] provides a partial solution to this, but does not cover all possible forms of such constraints. A more general grammar is necessary.

1.2 Definition of the problem

The problem is defined by the following grammar:

i, j, k			rigid variables (fixed, unknown values)
n	$::=$	$0 \mid 1 \mid \ldots$	concrete natural number
v, w	$::=$	n	size value
	$\mid$	$i + n$	rigid + offset
	$\mid$	∞	infinite size
	$\mid$	$\max(\vec{v})$	maximum of $v_1, \ldots, v_n$
H	$::=$	$i < v \mid i \leq v$	hypothesis
Γ	$::=$	$\vec{H}$	context (list of hypotheses)
X, Y, Z			flexible variables (we want to solve)
a, b, c	$::=$	v	size expression
	$\mid$	$\max(\vec{a})$	maximum of $a_1, \ldots, a_n$
	$\mid$	$X(\vec{a}) + n$	unsolved function applied to arguments + offset
C	$::=$	$a < b \mid a \leq b$	inequality constraint

A *problem* consists of:

- A list of hypotheses $\vec{H} = \Gamma$. Hypotheses do not contain information about flexible variables and define the *context* of the problem.

- A list of *constraints* $\vec{C}$. They express the relations between the flexible variables, and between flexible and rigid variables. Any constraint that only contains rigid variables (plus offset) has to be implied by the hypotheses.
- A list of flexible variables $X_1, \ldots, X_n$, and for each variable X_ν its arity $\mathsf{ar}(X_\nu)$ and its polarity $\mathsf{pol}(X_\nu) \in \{+, -\}$. The polarity tells us whether we are looking for the least $(-)$ or biggest $(+)$ solution.

Notation 1. When X depends on i, j, k we will write $X\ i\ j\ k$ instead of $X(i, j, k)$.

A *solution* for $X_\nu\ i_1\ \ldots\ i_m$ (where $m = \mathsf{ar}(X_\nu)$) is a size value v that contains only rigid variables in $\{i_1, \ldots, i_m\}$.

A solution θ for the whole problem is a solution $\theta(X_\nu)$ for each flexible variable X_ν such that the constraints $\vec{C}$ are satisfied by θ in the context Γ.

Example. The following code snippet inserts an element at the right position of a sorted list.

```
data List +(i : Size) ++(A : Set)
{ nil
; cons [j < i] (x : A) (xs : List j A) : List i A
}

data Bool { true ; false }

fun T   : Set {}
fun leq : T -> T -> Bool

fun insert : [i : Size] |i| -> T -> List i T -> List (i + 1) T
                       {- termination order -}
{ insert i x nil            = cons _ x nil
                                {- X1 -}
; insert i x (cons i' y ys) = case leq x y
  { true  -> cons _ x (cons _ y ys)
               {- X2 -}  {- X3 -}
  ; false -> cons _ y (insert _ x ys)
               {- X4 -}    {- X5 -}
  }
}
```

The hypothesis is

$$i' < i$$

because i is the termination order.

We can derive the following size constraints and from the types given in the code:

- In the case `nil`:

Given `nil : List X1 T`

To show `cons X1 x nil : List (i+1) T`
The resulting constraint is $X_1 < i+1$ because `cons` has the condition `j < i`.

- In the case `cons`:
  - subcase `true`:

    To show `cons X2 x (cons X3 y ys) : List (i+1) T`
    Once more, because of `cons`' type, we receive the constraints $X_2 < i+1$ and $i' \leq X_3$.

    To show `cons X3 y ys : List X2 T`
    Because of the outer `cons`, we get $X_3 < X_2$.

  - subcase `false`:

    To show `cons X4 y (insert X5 x ys) : List (i+1) T`
    The constructor constraint $X_4 < i + 1$ is derived.

    Given `ys : List i' T`

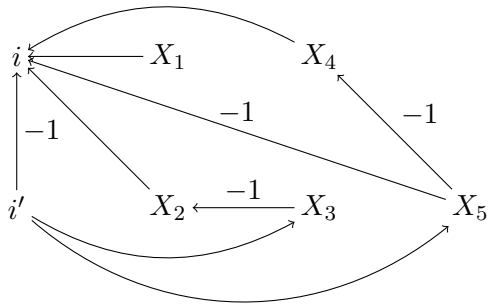
    To show `ys : List X5 T`
    We get the constraint $i' \leq X5$.

    Given `insert X5 x ys : List (X5+1) T`
    We conclude the subtyping constraint $X_5 < i$.

The resulting constraints can be unified to

$$\begin{aligned} X_1 &\leq i \\ X_2 &\leq i \\ i' &\leq X_3 \\ X_3 &< X_2 \\ X_4 &\leq i \\ i' &\leq X_5 \\ X_5 + 1 &\leq i \\ X_5 + 1 &\leq X_4 \end{aligned}$$

which leads to the following graph (see 2.2):



2 Simplified Approach

For a simplified approach, we assume that all flexible variables depend on the same tuple of rigid variables, e.g.

$$Xijk$$
$$Yijk$$
$$Zijk$$

Later on we will allow different signatures.

We also do not treat constraints of the form max(...) until later.

Our altered grammar now looks like this:

i, j, k			rigid variables (fixed, unknown values)
n	$::=$	$0 \mid 1 \mid \ldots$	concrete natural number
v, w	$::=$	n	size value
	$\mid$	$i + n$	rigid + offset
	$\mid$	∞	infinite size
H	$::=$	$i < v \mid i \leq v$	hypothesis
Γ	$::=$	$\vec{H}$	context (list of hypotheses)
X, Y, Z			flexible variables (we want to solve), each dependant on the same tuple of rigid variables
a, b, c	$::=$	v	size expression
C	$::=$	$a < b \mid a \leq b$	inequality constraint

First, we will unify the constraints to the canonical cases (2.1). Each of the unified constraints will be represented as an edge in the constraint graph (2.2). Afterwards we will define the composition of edges to find the transitive closure of the graph using the Floyd-Warshall-algorithm Cormen et al. [2009, p. 657f] (2.3). This allows to detect negative cycles that could imply a contradiction. To check whether our constraints are compatible with our hypotheses, we will construct another graph from the hypotheses and complete it. Now, we can test whether our hypotheses graph is compatible with our constraint graph (2.4). The next chapter covers the definition of the edges' algebraic structure (2.5).

Consequently, the constraint graph will be divided into the connected components (3.2). Starting at 0 or at ∞ – depending on whether we are looking for the least or

the biggest solution – we can set the values of nodes that are connected to those with known values by computing the least upper bound or the greatest lower bound.[1]

Finally we will remove our restrictions to the original problem and alter our algorithm to be applicable to the general problem (4).

2.1 Unification of the constraints

The constraints are of the form

$$a < b \mid a \leq b$$

where

$$a ::= n \mid i + n \mid \infty \mid X + n$$

$n, m \in \mathbb{N}$, i is a rigid variable, and X, Y are flexible variables. They can be unified to several distinct cases:

$$X < \infty \tag{2.1}$$
$$n \leq X \tag{2.2}$$
$$X \leq n \tag{2.3}$$
$$X + n \leq i \tag{2.4}$$
$$X \leq i + n \tag{2.5}$$
$$i + n \leq X \tag{2.6}$$
$$i \leq X + n \tag{2.7}$$
$$Y + n \leq X \tag{2.8}$$
$$Y \leq X + n \tag{2.9}$$
$$Y + n < X \tag{2.10}$$
$$Y < X + n \tag{2.11}$$

We will now treat constraints of the form $a \leq b$ and $a < b$ by iterating a for each possible b:

1. $a \leq \infty$: can be discarded because it does not contain any information.
2. $a < \infty$:
   a) $n < \infty$: can be discarded.
   b) $\infty < \infty$: is a contradiction.
   c) $i + n < \infty$: leads to $i < \infty$; it has to be checked if this is an implication of Γ.[2] If it is, the information is redundant and therefore can be discarded. Otherwise it is a contradiction.
   d) $X + n < \infty$: leads to $\boxed{X < \infty}$ (case 2.1).

[1] Nodes with known values are rigid variables and constants.

[2] We will do this by building a hypotheses graph and comparing the corresponding edges.

3. $a \leq n$:
   a) $m \leq n$: is either true or false. If it is true, discard, else it is a contradiction.
   b) $\infty \leq n$: is a contradiction.
   c) $i + m \leq n$:
      i. if $m > n$ then the statement is contradictionary because i is non-negative.
      ii. if $m \leq n$ the inequality may be written as $i \leq \underbrace{n - m}_{n'}$. If this is an implication of Γ, discard. Otherwise it is a contradiction.
   d) $X + m \leq n$:
      i. if $m > n$ then the statement is contradictionary because m is non-negative.
      ii. if $m \leq n$ the inequality may be written as $X \leq \underbrace{n - m}_{n'} \Rightarrow \boxed{X \leq n'}$ (case 2.3). If $n = m$ then $X \leq 0$ and therefore $\boxed{X = 0}$ can be assumed now.
4. $a < n$: if $n \neq 0$ then $a < n$ may be written as $a \leq n - 1$ and the respective cases can be treated as in the case $a \leq n$. Otherwise ($m < 0,\ \infty < 0,\ i + m < 0,\ X + m < 0$) it is a contradiction since all those values are defined as non-negative.
5. $a < i + n$:
   a) $\infty < i + n$: is a contradiction.
   b) $m < i + n$:
      i. if $m \geq n$ check if implied by Γ ($\rightarrow$ discard or contradiction).
      ii. if $m < n$, the statement can be discarded because $m < i + n$ is always true if $m < n$.
   c) $j + m < i + n$: check if implied by Γ ($\rightarrow$ discard or contradiction).
   d) $X + m < i + n$:
      i. if $m > n + 1$ then $X + \underbrace{m - n - 1}_{m'} + 1 < i$ is equivalent to $\boxed{X + m' \leq i}$ (case 2.4).
      ii. if $m \leq n + 1$ then $X < i + \underbrace{n - m + 1}_{n'} - 1$ is equivalent to $\boxed{X \leq i + n'}$ (case 2.5).
6. $a \leq i + n$:
   a) $\infty \leq i + n$: is a contradiction.
   b) $m \leq i + n$:

i. if $m \geq n$ check if implied by Γ ($\rightarrow$ discard or contradiction).

ii. if $m < n$, the statement can be discarded because $m \leq i + n$ is always true if $m < n$.

c) $j + m \leq i + n$: check if implied by Γ ($\rightarrow$ discard or contradiction).

d) $X + m \leq i + n$:

i. if $m > n$ then $X + \underbrace{m - n}_{m'} \leq i$ is kept: $\boxed{X + m' \leq i}$ (case 2.4).

ii. if $m \leq n$ then $X \leq i + \underbrace{n - m}_{n'}$ is kept: $\boxed{X \leq i + n'}$ (case 2.5).

7. $a \leq X + n$:

a) $\infty \leq X + n :\longrightarrow \boxed{X = \infty}$ can be solved.

b) $m \leq X + n$:

i. if $m > n$ the statement may be written as $\underbrace{m - n}_{m'} \leq X$: $\boxed{m' \leq X}$ (case 2.2).

ii. if $m \leq n$ the statement is equivalent to $0 \leq X + \underbrace{n - m}_{n'}$ which can be discarded because $X \geq 0$ and $n' \geq 0$.

c) $i + m \leq X + n$:

i. if $m > n$ the statement may be written as $i + \underbrace{m - n}_{m'} \leq X$: $\boxed{i + m' \leq X}$ (case 2.6).

ii. if $m \leq n$ the statement may be written as $i \leq X + \underbrace{n - m}_{n'}$: $\boxed{i \leq X + n'}$ (case 2.7).

d) $Y + m \leq X + n$:

i. if $m > n$ the statement may be written as $Y + \underbrace{m - n}_{m'} \leq X$: $\boxed{Y + m' \leq X}$ (case 2.8).

ii. if $m \leq n$ the statement may be written as $Y \leq X + \underbrace{n - m}_{n'}$: $\boxed{Y \leq X + n'}$ (case 2.9).

8. $a < X + n$:

a) $\infty < X + n$ is a contradiction.

b) $m < X + n$:

i. if $m > n+1$ the statement may be written as $\underbrace{m-n-1}_{m'}+1 < X$ which implies $\boxed{m' \leq X}$ (case 2.2).

ii. if $m \leq n+1$ the statement is equivalent to $0 < X + \underbrace{n-m+1}_{n'} - 1 \Rightarrow 0 \leq X + n'$ which can be discarded because $X \geq 0$ and $n' \geq 0$.

c) $i + m < X + n$:

i. if $m > n+1$ the statement may be written as $i + \underbrace{m-n-1}_{m'}+1 < X$ which implies $\boxed{i + m' \leq X}$ (case 2.6).

ii. if $m \leq n+1$ the statement is equivalent to $i < X + \underbrace{n-m+1}_{n'} - 1$ which implies $\boxed{i \leq X + n'}$ (case 2.7).

d) $Y + m < X + n$:

i. if $m > n$ the statement is equivalent as $Y + \underbrace{m-n}_{m'} < X \Rightarrow \boxed{Y + m' < X}$ (case 2.10).

ii. if $m \leq n$ the statement may be written to $Y < X + \underbrace{n-m}_{n'}$ which leads to $\boxed{Y < X + n'}$ (case 2.11).

We cannot use $m'' = m' + 1$ to change $<$ to $\leq$ (resp. $n'' = n' - 1$) because there is a difference between the inequalities when X is ∞.

2.2 Mapping of the constraints to a graph

We will now build a weighted graph $G = (V, E)$ with weights W from the constraints.

Definition 1. An *constraint graph* consists of

- *Vertices* $V = \{X_1, \ldots, X_\mu\} \cup \{i_1, \ldots, i_\nu\} \cup \{\infty\} \cup \{0\}$
- *Edges* $E \subseteq V \times V \times W$
- *Labels* $W \subseteq \{<, \leq\} \times (\mathbb{Z} \cup \{\infty\})$

Every constraint in the unified form may be written as an edge in the weighted graph. An edge $a \xrightarrow{\leq}_{n} b$ means $a \leq b + n$ and an edge $a \xrightarrow{<}_{n} b$ means $a < b + n$. We only have to consider the unified constraints ($\boxed{\text{boxed}}$ in the previous chapter):

case	constraint	resulting edge
2.1	$X < \infty$	$X \xrightarrow{<} \infty$
2.2	$n \leq X$	$0 \xrightarrow[-n]{\leq} X$
2.3	$X \leq n$	$0 \xrightarrow[+n]{\leq} X$
2.4	$X + n \leq i$	$X \xrightarrow[-n]{\leq} i$
2.5	$X \leq i + n$	$X \xrightarrow[+n]{\leq} i$
2.6	$i + n \leq X$	$i \xrightarrow[-n]{\leq} X$
2.7	$i \leq X + n$	$i \xrightarrow[+n]{\leq} X$
2.8	$Y + n \leq X$	$X \xrightarrow[-n]{\leq} Y$
2.9	$Y \leq X + n$	$X \xrightarrow[+n]{\leq} Y$
2.10	$Y + n < X$	$X \xrightarrow[-n]{<} Y$
2.11	$Y < X + n$	$X \xrightarrow[+n]{<} Y$

Since per definition every rigid variable i and every flexible variable X is non-negative and $\leq \infty$, we may insert the following edges into our graph as well[3]:

$$0 \xrightarrow[0]{\leq} X \quad 0 \xrightarrow[0]{\leq} i$$
$$X \xrightarrow[0]{\leq} \infty \quad i \xrightarrow[0]{\leq} \infty$$

Nodes that are not connected can be assumed to have edges with weight $(\leq, \infty)$ between them (in both directions)[4].

2.2.1 Reflexive closure

As we will see later 3.1, it is also reasonable to construct self-loops with weight $(\leq, 0)$ for every rigid and flexible variable. Such an edge does no add any information since $X \leq X + 0$ and $i \leq i + 0$ is always true.

$$X \circlearrowright^{\leq}_{0}$$

$$i \circlearrowright^{\leq}_{0}$$

This gives us the *reflexive closure* of the graph.

[3] because $0 \leq X \leq \infty$ and $0 \leq i \leq \infty$
[4] $a \leq b + \infty = \infty$ is always true.

2.3 Graph completion

2.3.1 Transitive closure

We want to complete the graph using the Floyd-Warshall-algorithm so that cycles and contradictions may be spotted easily. In the *transitive closure* we can find cycles by looking for self-loops.

As described in Cormen et al. [2009, p. 697] first, we need to number our nodes $\{1, \dots, n\}$. For $i, j, k \in \{1, \dots, n\}$ we define $t_{ij}^{(k)}$ as the label of the edge (i, j) after step k.

Then we set

$$t_{ij}^{(0)} = \begin{cases} (c, w) & \text{if there is an edge } (i,j) \text{ with label } (c,w) \text{ (comparator } c\text{, weight } w) \\ (\leq, 0) & \text{if } i = j \\ \infty & \text{else} \end{cases}$$

For $k \geq 1$:

$$t_{ij}^{(k)} = \min\left(t_{ij}^{(k-1)}, t_{ik}^{(k-1)} + t_{kj}^{(k-1)}\right)$$

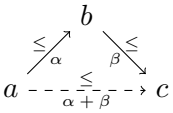
$+$ and $\min(\dots)$ will be defined later as compose ($\circ$) and meet ($\wedge$) (in chapter 2.5).

Once completed, $t_{ij}^{(n)}$ are elements of a matrix $T_{ij} \in (C \times W)^n$ that gives us the label (c, w) of an edge (i, j) in the transitive closure of the graph.

2.3.2 Composition

When composing edges (using *compose* $\circ$, see 2.5), two cases have to be distinguished:

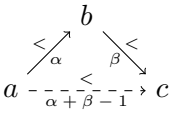
1. Both comparators are the same: In this case we can add an edge from a to b with weight $\alpha + \beta$.



   This is correct because

$$\begin{aligned} a &\leq b + \alpha \\ b + \alpha &\leq c + \beta + \alpha \\ a &\leq c + \alpha + \beta \end{aligned}$$

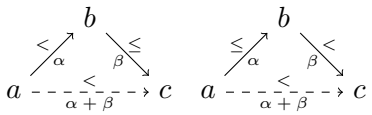
   The case $< / <$ can be treated similarily



The resulting edge has $\alpha + \beta - 1$ as weight because

$$\begin{aligned} a &< b + \alpha \\ b + \alpha &< c + \beta + \alpha \\ b + \alpha &\leq c + \beta + \alpha - 1 \\ a &< c + \alpha + \beta - 1 \end{aligned}$$

2. $\leq$ / $<$ or $<$ / $\leq$: If we have different comparators, the weights are also added but the resulting edge has $<$ as comparator.



This is correct because of

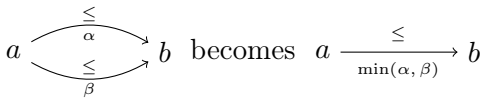
$$\begin{aligned} a &\leq b + \alpha \\ a - \alpha &\leq b < c + \beta \\ a &< c + \alpha + \beta \end{aligned}$$

and

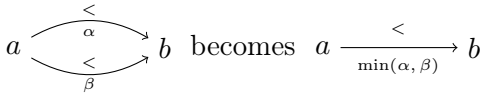
$$\begin{aligned} a &< b + \alpha \\ a - \alpha &< b \leq c + \beta \\ a &< c + \alpha + \beta \end{aligned}$$

Every time a new edge (a, b) with weight α is added, we check whether an edge from a to b already exists. If one exists, the two edges have to be compared (using *meet* $\wedge$, see 2.5) and the one with more information is kept; $\wedge$ returns the *minimum* of both edges.

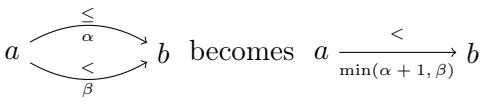
1. If both edges have $\leq$ or both edges have $<$ as comparators, the new edge is determined as follows:



and



2. If one edge has $\leq$ and the other $<$, the new edge is:



$\min(\alpha, \beta)$ can always be computed because $\alpha, \beta \in \mathbb{Z} \cup \{\infty\}$. Using this method, one of the two edges can always be eliminated because one of the statements always implies the other.

2.4 Consistency

When simplifying the constraints, we have to check if they are compatible with the hypotheses. For this purpose, we build another graph from the hypotheses and compare the edges to the corresponding edges in the constraint graph. If an edge in the hypotheses graph does not have a corresponding edge in the constraint graph[5] or the edge in the hypotheses graph is stricter than the one in the constraint graph, we proceed. If not, we have detected a contradiction.

Alternatively, we could look at all edges in the constraint graph that contain no flexible variables and compare them to their corresponding edges in the hypotheses graph.

Definition 2. Let a, b be nodes and let e, e' be edges from a to b with labels (c, w) and (c', w').

e is *stricter* than e' if and only if $(c, w) \wedge (c', w') = (c, w)$.

Example. Let the hypotheses be

$$i \leq j + 1$$
$$j \leq k$$

and from the constraints we shall conclude that

$$i \leq j$$

The resulting hypotheses graph looks like this:

$$i \xrightarrow[1]{\leq} j \xrightarrow[0]{\leq} k$$

and the resulting constraint graph has an edge

$$i \xrightarrow[0]{\leq} j$$

Now we have to search the corresponding edge in the hypotheses graph to every edge from the constraint graph that does not contain flexible variables. Since $i \xrightarrow[0]{\leq} j$ is stricter than $i \xrightarrow[1]{\leq} j$ we have arrived at a contradiction.

[5] respectively, the edge has weight ∞

2.4.1 Checking for negative cycles

In the completed graph, non-trivial loops exist at a vertex a if a is part of a circular path.

Redundant loops

- $a \overset{\leq}{\underset{\alpha}{\circlearrowleft}}$ can be left out if $\alpha \geq 0$ because then $a \leq a + \alpha$ is always true.
- $a \overset{<}{\underset{\alpha}{\circlearrowleft}}$ can be left out if $\alpha > 0$ because then $a < a + \alpha$ is always true.

Negative loops

- $0/i \overset{\leq}{\underset{\alpha}{\circlearrowleft}}$ is a contradiction if $\alpha < 0$.
- $X/0/i \overset{<}{\underset{\alpha}{\circlearrowleft}}$ is a contradiction if $\alpha \leq 0$.

Non-contradictionary, non-redundant loops

- $X \overset{\leq}{\underset{\alpha}{\circlearrowleft}}$ If $\alpha < 0$, this leads to $\boxed{X = \infty}$ because

$$X \leq X + \alpha$$

This can only be true if $X = \infty$. If an edge stating that $X < \infty$ exists, we produced a contradiction.

2.5 Algebraic structure

To formalize our graph structure, we will now analyse its algebraic properties. The edges can be seen as members of algebraic structures. Because of their properties, a bounded semilattice and a dioid can be constructed. We will show this by proving that the edges satisfy their definition.

2.5.1 Bounded semilattice

When comparing two edges with the same source and destination nodes – we call this operation *meet* ($\wedge$) – a *bounded meet semilattice* can be formed.

Let $L = C \times W$ be the set of the labels, where $W = \mathbb{Z} \cup \{\infty\}$ is the set of the weights and $C = \{\leq, <\}$ is the set of the comparators. Let $\wedge$ be the infix notation of meet : $L \times L \to L$.

$(L, \wedge)$ is a bounded semilattice if and only if the following holds:

- Associativity
$$\forall l, m, n \in L.\ l \wedge (m \wedge n) = (l \wedge m) \wedge n$$
- Commutativity
$$\forall l, m \in L.\ l \wedge m = m \wedge l$$
- Idempotency
$$\forall l \in L.\ l \wedge l = l$$
- Neutral element
$$\exists \top \in L.\ \forall l \in L.\ l \wedge \top = l$$

The operation *meet* is similar to a minimum, comparing two labels[6] and returning the one with more information. The edge with less information may now be elminitated because it is already implied.

Let $(c, w), (c', w') \in L$. Then $\wedge : L \times L \to L$ is defined by:
When both weights are ∞:

$$(\leq, \infty) \wedge (<, \infty) = (<, \infty)$$
$$(<, \infty) \wedge (\leq, \infty) = (<, \infty)$$
$$(c, \infty) \wedge (c, \infty) = (c, \infty)$$

If we *meet* an edge with ∞ as weight to an edge with finite weight, the resulting edge is the finite one. For $w \neq \infty$:

$$(c', \infty) \wedge (c, w) = (c, w)$$
$$(c, w) \wedge (c', \infty) = (c, w)$$

And for $w, w' \neq \infty$: If the two comparators are the same, the comparator stays the same and the resulting weight is the minimum of the two weights.

$$(<, w) \wedge (<, w') = (<, \min(w, w'))$$
$$(\leq, w) \wedge (\leq, w') = (\leq, \min(w, w'))$$

If the two comparators are different, we can simply change $<$ to $\leq$ by decrementing one of the weights and using the above definition (since none of the weights is ∞):

$$(\leq, w) \wedge (<, w') = (\leq, \min(w, w' - 1))$$
$$(<, w) \wedge (\leq, w') = (\leq, \min(w - 1, w'))$$

These minimums can always be computed because $w, w' \in \mathbb{Z}$.

Commutativity: per definition (since $\min(w, w') = \min(w', w)$)

[6] consisting of a comparator and a weight

Neutral element: The neutral element, called $\top$ (*top*), is an edge with label $(\leq, \infty)$.

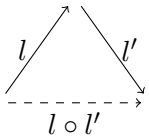
$$\top = (\leq, \infty)$$

Idempotency: per definition.

Associativity: can be shown using the definition.

2.5.2 Forming a dioid

We can expand this by defining a *semiring* with an idempotent + (in our case $\wedge$) – a *dioid*. We declare a second operation $\circ$ (*compose*) that takes two labels of edges that are connected (the destination node of the first is the source node of the second) and returns the label of the resulting edge.



A semiring $(L, \wedge, \circ)$ has to satisfy:

- $(L, \wedge)$ is a commutative monoid with an identity element $(\top)$
- $(L, \circ)$ is a monoid with an identity element (u)
- Distributivity
$$\forall l, m, n \in L.\ l \circ (m \wedge n) = (l \circ m) \wedge (l \circ n)$$
- Composing with $\top$ returns $\top$
$$\forall l \in L.\ l \circ \top = \top$$

$(L, \wedge)$ is a commutative monoid with identity element $\top$ because it is also a bounded semilattice.

Definition 3. Let $L = C \times W$, $W = \mathbb{Z} \cup \{\infty\}$ and $C = \{\leq, <\}$. Let $(c, w), (c', w') \in L$.
We define $\circ : L \times L \to L$ as follows.
Composition with $\top$ returns $\top$:

$$(\leq, w) \circ (\leq, \infty) = (\leq, \infty)$$
$$(\leq, \infty) \circ (\leq, w) = (\leq, \infty)$$

If c or c' is $<$:

$$(c, w) \circ (c', \infty) = (<, \infty)$$
$$(c, \infty) \circ (c', w') = (<, \infty)$$

Composing (c, w) with the neutral element $u = (\leq, 0)$ returns (c, w):

$$(c, w) \circ (\leq, 0) = (c, w)$$
$$(\leq, 0) \circ (c, w) = (c, w)$$

For $w, w' \neq \infty$:

$$(c, w) \circ (c, w') = (c, w + w'))$$
$$(\leq, w) \circ (<, w') = (<, w + w'))$$
$$(<, w) \circ (\leq, w') = (<, w + w')$$

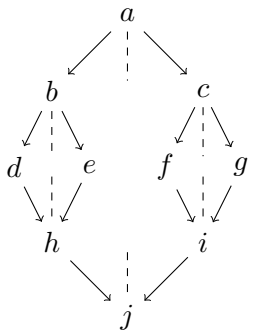
The only dioid axioms that are now left to show are the associativity of $\circ$ and the distributivity of $\wedge$ and $\circ$.

Associativity: can be shown using the definition.

Distributivity: can be shown using the definition.

3 Finding a solution

In the special case that our graph is a directed acyclic graph (DAG) we can find the solutions fairly straightforward.



Since a should be a rigid variable or a constant, we may simply set b to the value of $a - w$ or $a - w + 1$ depending on the comparator.[7] This works if we are looking for the least solution; if we are looking for the biggest solution we can simply start at the bottom. In this manner we can find the solution for every node if we know the values of the nodes on the other sides of all incoming (resp. outgoing) edges. Then we have to compare all incoming (resp. outgoing) edges and use the strictest one to determine the solution.

However, this form is only possible if the graph does not contain loops. Since we would have stopped early on if the loops were negative, only non-negative loops can remain.

3.1 Least upper bound and greatest lower bound

To compare all incoming (resp. outgoing) edges to a node and determine the strictest, we define the following terms:

Definition 4 (Least upper bound, greatest lower bound)**.**
Let $X, v_1, v_2 \in V$, $(X, v_1, l_1), (v, v_2, l_2), (v_1, v_2, l_{1,2}), (v_2, v_1, l_{2,1}) \in E$ and $l_1 = (c_1, w_1)$, $l_2 = (c_2, w_2)$.

- The *least upper bound* of X is

$$\bigsqcup X = \text{minlub}\{(X, v', l) \in E \mid v' \in V, l \neq (\leq, \infty)\}$$

[7] Let w be the weight of the edge between a and b.

where the minimum is computed as follows (let the comparators be $\leq$ for now[8]):

$$\text{minlub}\{(X, v_1, l_1), \ldots, (X, v_2, l_2)\} =$$
$$= \begin{cases} v_i + w_i & \text{if } v_i \text{ has no incoming edges from} \\ & \text{any } v_j,\ j \in \{1, \ldots, n\},\ i \neq j \\ & \text{if there is more than one node like this} \\ & \text{one of the } v_i + w_i \text{ is picked arbitrarily} \\ \text{undecidable} & \text{else} \end{cases}$$

- The *greatest lower bound* of X is

$$\bigsqcup X = \text{maxglb}\{(v', X, l) \in E \mid v' \in V, l \neq (\leq, \infty)\}$$

where the maximum is:

$$\text{maxglb}\{(v_1, X, l_1), \ldots, (v_2, X, l_2)\} =$$
$$= \begin{cases} v_i - w_i & \text{if } v_i \text{ has no outgoing edges from} \\ & \text{any } v_j,\ j \in \{1, \ldots, n\},\ i \neq j \\ & \text{if there is more than one node like this} \\ & \text{one of the } v_i - w_i \text{ is picked arbitrarily} \\ \text{undecidable} & \text{else} \end{cases}$$

If *undecidable* is returned, it is arbitrary which of the edges we pick.

3.2 Finding connected components

If our graph contains loops, we need to divide it into connected components. Since the graph is directed, we search for weakly connected components which means connected components in the underlying undirected graph. The nodes 0 and ∞ are omitted because they would connect every node. We can also ignore rigid variable nodes when looking for connected components.

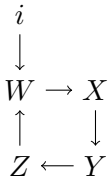
We do this by treating our graph as a graph forest when building it. Every time a new edge is added we check whether it connects two existing graphs and join them in this case. This can be done by using a *union find* algorithm on a *disjoint-set* data structure (see Cormen et al. [2009, p. 562ff]).

Thus, we can treat each connected component separately.

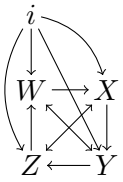
If more than one edge from outside of the connected component enters it, it is arbitrary which variable we set first. An incoming edge to a member of the loop implies incoming edges to all other members of the loop since we built the transitive closure of the graph.

[8] If the comparators are not $\leq$ we have to consider an offset by one.

Example.



Here we also get the edges e to b, e to c and e to d when completing the graph, so we can choose which of a, b, c, d to set first (assuming we already know e).



Example. Let the rigid variables be $\{i_1, i_2\}$ and the flexible variables be $\{X_1, X_2, X_3\}$, each depending on (i_1, i_2). $\mathsf{pol}(X_1) = \mathsf{pol}(X_1) = \mathsf{pol}(X_1) = -$.

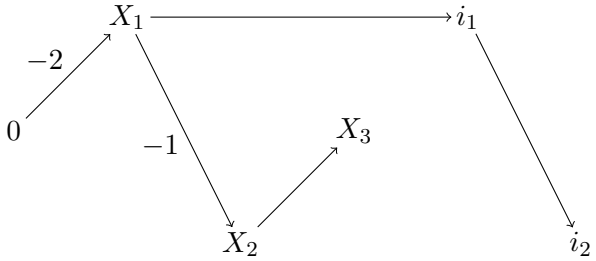
Let the hypotheses be

$$i_1 + 3 \leq i_2$$

and the constraints be

$$\begin{aligned} 2 &\leq X_1 \\ X_1 + 1 &\leq X_2 \\ X_2 &\leq X_3 \\ X_1 &\leq i_1 \\ i_1 &\leq i_2 \end{aligned}$$

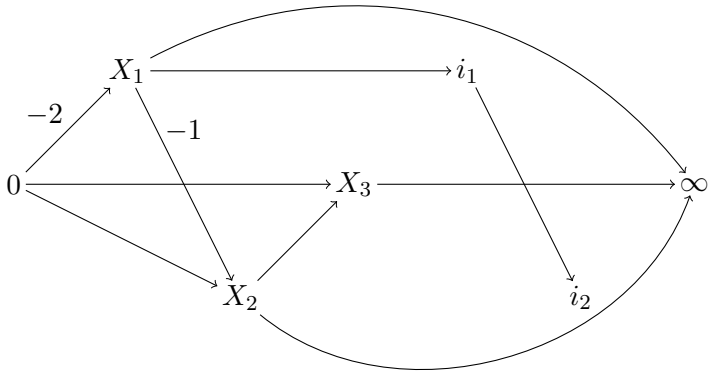
Our initial constraint graph looks like this [9]



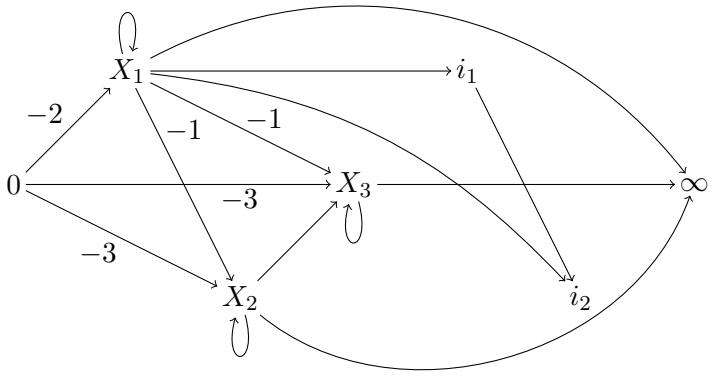
Now we add the edges from 0 to the flexible variables and from the flexible variables to ∞[10]:

[9] No label means $(\leq, 0)$

[10] Note that the edge from 0 to X_1 is not replaced since the existing edge is stricter



Next, the reflexive-transitive closure is built:



Thus, we need to check whether our hypotheses are compatible with the constraints. A graph is built from the hypotheses:

$$i_1 \xrightarrow{\quad -3 \quad} i_2$$

Since the corresponding edge in the constraint graph is weaker, the hypotheses are indeed compatible with the constraints.

All nodes are already in the same connected component[11], so we can now find the solutions. Since the polarity of all flexible variables is $-$, we start from 0.

- When entering the connected component, we arbitrarily[12] set $X_1 = 2$ because the greatest lower bound of X_1 is 2.

[11] even in the graph without 0 and ∞
[12] We could have set X_2 or X_3 as well

- Next, we set $X_2 = 3$ because the greatest lower bound of X_2 is $\max(0-(-3), 2-(-1))$.
- Similarily, $X_3 = 3$ because its greatest lower bound is $\max(0-(-3), 2-(-1), 3-0)$.

Had we been looking for the biggest solutions ($\mathsf{pol}(X_{1,2,3}) = +$), the solutions would have been $X_1 = i_1,\ X_2 = \infty,\ X_3 = \infty$.

4 Extended Problem

In our first approach, two restrictions to the original definition of the problem were made. We will now remove these and try to adapt our algorithm.

4.1 Allowing maximums

The first restriction was to limit the problem to hypotheses and constraints that are not of the form

$$\max(\dots)$$

What we have to do is

a) Simplify such constraints

b) Model them to a graph

Simplifying According to our grammar, such a constraint is of the form $a < b$ or $a \leq b$ where a, b is either a size expression, a maximum of size expressions or an unsolved variable (plus offset).

Several cases can be distinguished:

Let $m, n, m_1, \dots, m_r, n_1, \dots, n_t, o_1, \dots, o_s, p_1, \dots, o_u \in \mathbb{N}$ and let $i_1, \dots, i_s, j_1, \dots, j_u$ be rigid variables (for $r, s, t, u \in \mathbb{N}$).

1. Constant on the left side

   a) Only constants in the maximum:

   $$n \leq \max(m_1, \dots, m_r)$$
   $$n < \max(m_1, \dots, m_r)$$

   The right hand sides can be evaluated; the resulting inequality is either true or false. If true, we can discard it. Otherwise it is a contradiction.

   b) Constants and rigids (plus offsets) in the maximum:

   $$n \leq \max(m_1, \dots, m_r, i_1 + o_1, \dots, i_s + o_s)$$
   $$n < \max(m_1, \dots, m_r, i_1 + o_1, \dots, i_s + o_s)$$

   We can find the maximum of $m_1, \dots, m_r$ and replace $m_1, \dots, m_r$ by that maximum m.

   $$\boxed{n \leq \max(m, i_1 + o_1, \dots, i_s + o_s)}$$
   $$\boxed{n < \max(m, i_1 + o_1, \dots, i_s + o_s)}$$

c) Infinity in the maximum:

$$n \leq \max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s, \infty)$$
$$n < \max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s, \infty)$$

The maximums compute to ∞. The resulting inequality is therefore always true and we can discard it.

2. Constant on the right side: The first two cases work the same way as above. Later on, we will map this to multiple edges by interpreting

$$\max(m, i_1 + o_1, \ldots, i_s + o_s) \leq n$$

as a list of inequations ($m \leq n$ is either true ($\rightarrow$ discard) or false ($\rightarrow$ contradiction))

$$\boxed{i_1 + o_1 \leq n}$$
$$\ldots$$
$$\boxed{i_s + o_s \leq n}$$

and analog for $<$.

When we have ∞ in a maximum on the left side our resulting inequality is

$$\infty \leq n$$
$$\infty < n$$

which is a contradiction.

3. Rigid variable (plus offset) on the left side:

$$i + n \leq \max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s)$$
$$i + n < \max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s)$$

As stated earlier, we can find the maximum of $m_1, \ldots, m_r$ and replace $m_1, \ldots, m_r$ by that value m.

$$\boxed{i + n \leq \max(m, i_1 + o_1, \ldots, i_s + o_s)}$$

and anlog for $<$.

4. Rigid variable (plus offset) on the right side:

$$\max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s) \leq n + i$$
$$\max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s) < n + i$$

Once more, we replace $m_1, \ldots, m_r$ by their maximum m and divide it into multiple inequations.

$$\boxed{m \leq i + n}$$
$$\boxed{i_1 + o_1 \leq i + n}$$
$$\ldots$$
$$\boxed{i_s + o_s \leq i + n}$$

and in analogy for $<$.

5. Infinity on the left side:
   a) $\infty \leq \max(\ldots)$ is a true if and only if the maximum contains ∞. Otherwise it is a contradiction.
   b) $\infty < \max(\ldots)$ is a contradiction.
6. Infinity on the right side:
   a) $\max(\ldots) \leq \infty$ is always true and can therefore be discarded.
   b) $\max(\ldots) < \infty$ is true if and only if the maximum does *not* contain ∞. Otherwise it is a contradiction.
7. $\max(\ldots)$ on both sides

$$\max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s) \leq \max(n_1, \ldots, n_t, j_1 + p_1, \ldots, j_u + p_u)$$
$$\max(m_1, \ldots, m_r, i_1 + o_1, \ldots, i_s + o_s) < \max(n_1, \ldots, n_t, j_1 + p_1, \ldots, j_u + p_u)$$

The maximum of $m_1, \ldots, m_r$ is called m and the maximum of $n_1, \ldots, n_r$ is called n.

Such an inequality means that:

$$\boxed{m \leq \max(n, j_1 + p_1, \ldots, j_u + p_u)}$$
$$\boxed{i_1 + o_1 \leq \max(n, j_1 + p_1, \ldots, j_u + p_u)}$$
$$\ldots$$
$$\boxed{i_s + o_s \leq \max(n, j_1 + p_1, \ldots, j_u + p_u)}$$

and analog for $<$.

Modelling Constraints of the form

$$i + n \leq m$$
$$i + n < m$$

may simply be mapped to an edge in the graph $i \xrightarrow[m-n]{\leq} 0$ or resp. $i \xrightarrow[m-n]{<} 0$

Constraints of the form

$$n \leq \max(m, i_1 + o_1, \ldots, i_s + o_s)$$
$$n < \max(m, i_1 + o_1, \ldots, i_s + o_s)$$

or

$$i + n \leq \max(m, i_1 + o_1, \ldots, i_s + o_s)$$
$$i + n < \max(m, i_1 + o_1, \ldots, i_s + o_s)$$

are difficult to represent since n (or $i + n$) does not have to be $\leq$ ($<$) than all of the arguments of the maximum. Such an inequation means that n (or $i + n$) has to be $\leq$ ($<$) than at least one of the arguments of the maximum.

We will ignore such a constraint when building the graph. Later on, when the graph is completed, we can check if there are edges between n (or $i + n$) and at least one of the arguments. At least one of these edges has to be stricter than the corresponding condition stated by the inequality. If this is not the case, we have produced a contradiction.

4.2 Variable mapping

We also assumed that all flexible variables depend on the same rigid variables (e.g. $X\ i\ j\ k$, $Y\ i\ j\ k$, $Z\ i\ j\ k$). When we remove this constraint, we have to attend to the variable mapping. Every edge will also receive a function π that maps the variables of the source node onto those of the destination node.

As a result, it is not always possible to compare two of those edges. Therefore we have to allow multiple edges between two nodes. When finding a solution, we have to check if it satisfies all of them.

Let R be the set of the rigid variables and $i_1, \ldots, i_n$. We then define the mapping function as the permutation

$$\pi_{(\nu_1,\ldots,\nu_n)} : R^n \to R^n$$
$$(i_1, \ldots, i_n) \mapsto \pi_{(\nu_1,\ldots,\nu_n)}(i_1, \ldots, i_n) = (i_{\nu_1}, \ldots, i_{\nu_n})$$

Remark. $\nu_i \neq \nu_j$ for $i \neq j$ is not necessarily true. $\pi_{(1,\ldots,1)}$ would be a possible function.[13]

When mapping a constraint $X\ i_1\ \ldots\ i_n \leq Y\ i_{\nu_1}\ \ldots\ i_{\nu_n} + \alpha$ to the graph, we need to create an edge from X to Y with weight $(\leq, \alpha, \pi_{(\nu_1,\ldots,\nu_n)})$.

$$X \xrightarrow[\leq]{\pi_{(\nu_1,\ldots,\nu_n)}} Y$$

[13]Therefore our function is not a true *permutation*.

The case with $<$ as comparator works the same.

Example. Let $X\ i\ j\ k \leq Y\ i\ k\ j$ be a constraint. This means that the arguments (i, j, k) are mapped to (i, k, j). The mapping function is therefore

$$\begin{aligned} \pi_{(1,3,2)} &: R^3 \to R^3 \\ (i, j, k) &\mapsto \pi_{(1,3,2)}(i, j, k) = (i, k, j) \end{aligned}$$

and the resulting edge is

$$X \xrightarrow[\substack{0 \\ \leq}]{\pi_{(1,3,2)}} Y$$

4.2.1 Composition of edges with permutations

When we complete the graph and two edges have to be *composed*, $\circ$ has to be adjusted to treat permutation functions.

The permutation functions of two edges can simply be *composed* as $\pi \circ \pi'$, where $\circ$ is the function composition.

This works because [14]

$$\begin{aligned} X(i_1, \ldots, i_n) &\leq Y(\pi_{(\nu_1,\ldots,\nu_n)}(i_1, \ldots, i_n)) = Y(i_{\nu_1}, \ldots, i_{\nu_n}) \\ Y(i_{\nu_1}, \ldots, i_{\nu_n}) &\leq Z(i_{\nu_{\xi_1}}, \ldots, i_{\nu_{\xi_n}}) = Z(\pi_{(\nu_{\xi_1},\ldots,\nu_{\xi_n})}(i_1, \ldots, i_n)) \\ &= Z(\pi_{(\nu_1,\ldots,\nu_n)}(\pi_{(\xi_1,\ldots,\xi_n)}(i_1, \ldots, i_n))) \\ X(i_1, \ldots, i_n) &\leq Z(\pi_{(\nu_1,\ldots,\nu_n)} \circ \pi_{(\xi_1,\ldots,\xi_n)}(i_1, \ldots, i_n)) \end{aligned}$$

The rest of the labels (comparator and weight) is treated like before.

4.3 Finding a solution in the extended problem

Since we cannot necessarily compare two edges with permutation functions, both edges have to be considered when finding the solution. When a possible solution is found, it has to be checked if it satisfies all parallel[15] edges.

Example. Let the constraints be

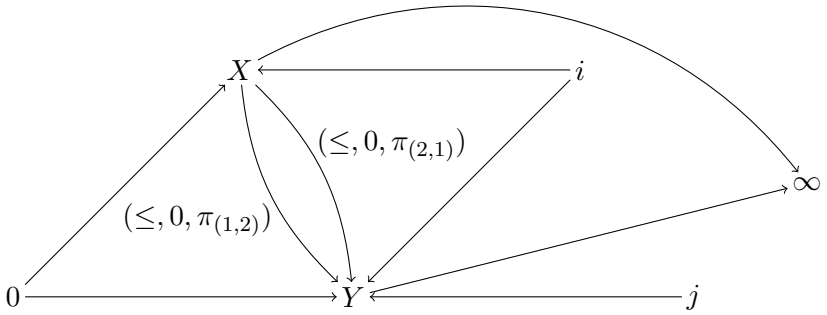
$$\begin{aligned} Xij &\leq Yji \\ Xij &\leq Yij \\ i &\leq Xij \\ i &\leq Yij \end{aligned}$$

and $\mathsf{pol}(X) = \mathsf{pol}(Y) = -$. The transitive closure of the graph looks like this[16]:

[14] labels are assumed to be $(\leq, 0)$ – the equations are valid with offsets as well

[15] with the same source and destination nodes

[16] No label means $(\leq, 0, \pi_{(1,2)})$.



The greatest lower bound for $Y\ i\ j$ is j, the least upper bound is ∞. The solution is therefore j since we are looking for the least solution.

The greatest lower bound for $X\ i\ j$ is 0 because the only incoming edge is from 0 with label $(\leq, 0, \pi_{(1,2)})$. The least upper bound for $X\ i\ j$ is either $Y\ i\ j = j$ or $Y\ j\ i = i$ (the second argument). Possible solutions for $X\ i\ j$ are therefore i and j.

5 Conclusion

The approach presented can be used to solve a general form of size constraints. If solutions exists, they are returned and contradictions between the constraints and the hypotheses and within the constraint inequalities are detected.

We simplified the constraints and mapped them to a graph. Then, the reflexive-transitive closure of the graph was formed. To check for consistency, we did the same with the hypotheses and compared the two graphs. Possible contradictions, such as negative cycles or violation of the hypotheses, are revealed. Afterwards we divided the constraint graph into connected components and determined the solutions in each of them separately.

Attending to maximums and the variable mapping of flexible variable signatures complicates the procedure a lot. Therefore it was reasonable to use a simplified approach and extend it later.

In an implementation, the computational time and space complexity largely depends on the data structures used. It should, however, be $\mathcal{O}(mn^3)$ where n is the number of flexible variable nodes in the largest connected component and m is the number of connected components. If treemaps instead of arrays are used, the complexity is $\mathcal{O}(mn^3 \log(n))$.

Acknowledgements

First and foremost, I want to thank my supervisor, Andreas Abel, for his support and guidance: Thank you!

Furthermore I would like to thank my girlfriend Lea, my family and my friends, especially Lukas, for their advice and patience.

Bibliography

A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *International Conference on Functional Programming (ICFP 2013)*, 2013. To appear.

G. Barthe, B. Grégoire, and F. Pastawski. CIC$\widehat{}$: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006. ISBN 3-540-48281-4.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

H. B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, 2001. ISBN 9780122384523.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe.

Ort, Datum	Unterschrift