MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Verification of binarised neural networks using ASP

Bachelor's Thesis

JINDŘICH MATUŠKA

Brno, Spring 2024

MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Verification of binarised neural networks using ASP

Bachelor's Thesis

JINDŘICH MATUŠKA

Advisor: RNDr. Samuel Pastva, PhD.

Department of Computer Systems and Communications

Brno, Spring 2024

MUNI
FI

Declaration

Advisor: RNDr. Samuel Pastva, PhD.

Acknowledgements

My thanks go to my family, which has provided me with a firm ground for my whole life, to my friends whom I could spend my free time and who were distracting me from the never-ending study duties, to all my teachers, who had patience with me and helped me grow, and finally to my advisor, who has provided me with this assignment and was my support when writing this thesis.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Abstract

Deep neural networks are state-of-the-art technology. Using them in critical real-life applications carries a risk of failure. For this, verification of their properties is needed. This thesis explores the possibility for the use of answer set programming paradigm in this task. It implements a quantitative verificator for binarized neural networks, a special case of deep neural networks, using this paradigm. It also demonstrates the use of this verificator on a network trained on the MNIST dataset. The verificator proves to be especially good when evaluating highly robust networks.

Keywords

verification, binarized neural networks, answer set programming, Clingo, robustness

Contents

List of Tables

List of Definitions

Thesis assignment

ASP (Answer Set Programming) is a form of constraint programming designed for solving various NP-complete search problems. It is often used as an alternative to SAT/SMT solvers or symbolic algorithms based on BDDs (Binary Decision Diagrams). Meanwhile, SAT/SMT and BDDs are one of the key tools in current verification and validation workflows for binarised neural networks (BNNs) [1]. Conceptually, the goal of this thesis is therefore to explore the possibilities of applying ASP to reason about the behaviour of binarised neural networks. This should supplement the existing SAT/SMT/BDD-based approaches.

More concretely, the student should familiarise themselves with the problem of BNN robustness and the ASP method in general. They should then formulate an ASP encoding of the BNN robustness problem such that the robustness of a network can be validated using a suitable ASP solver (e.g. clingo [2]). The student should then create a prototype implementation of this encoding that will be tested on a collection of reasonable examples (such as the ones from [1]).

1 Introduction

2 Binarised neural networks

Binary neural networks are a type of Deep neural networks where instead of floating point numbers, binary values are used as inputs and outputs of layers. This reduction of available values can lead to high reduction in the computation time and energy consumption while achieving near state-of-the-art results [3].

Notation

Symbol	Definition
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of whole numbers
$\mathbb{B}$	set of ± 1-binarised values, $\mathbb{B} = \{-1, 1\}$
K^m	set of vectors on K of lenght m

Table 2.1: Used notation

2.1 Definition of Binarised neural network

2.1.1 Deep neural network

General neural network is a multilayer perceptron. It consists of perceptrons (neurons) in layers, these are further split into the input layer, the output layer and possibly multiple hidden layers. Each perceptron computes its inner potential ξ based on the outputs of the previous layer, its output is then determined by an activation function σ.

For the computation of the inner potential ξ weighted sum is used. There are many different used activation functions σ such as *unit step function*, *logistic sigmoid*, *hyperbolic tangens* or *ReLU*.

Multiple of these perceptrons are then assorted into layers. The input layer is not consisting of perceptrons but is straight composed of the inputs. The output layer is often represented by perceptrons with activation function from other layers, such as *argmax* for single-choice classification and *softmax* for the classification using the probability.

Further follows a more formal definition of the deep neural network corresponding to the one from [4].

Definition 2.1.1 (Perceptron)**.** Perceptron p is a function from vector of k real numbers to a real number. The function p is a composition of an inner potential ξ and an activation function σ. The inner potential ξ is weighted sum parametrized by static vector of real numbers $\vec{w}$ of size k and real bias b. The activation function σ can be instantized by any real-valued function.

$$p : \mathbb{R}^k \to \mathbb{R}$$

$$\xi : \mathbb{R}^m \to \mathbb{R},\ \sigma : \mathbb{R} \to \mathbb{R}$$

$$\xi(\vec{x}) = b + \sum_{i=1}^{m} w_i \cdot x_i$$

$$p = \sigma \circ \xi$$

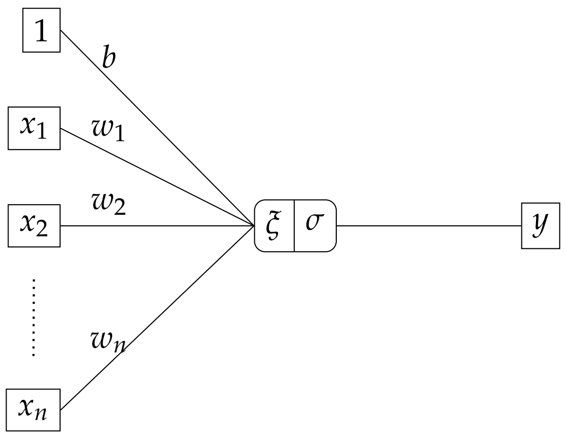

Figure 2.1: Schema of a perceptron

Definition 2.1.2 (Single-layer perceptron)**.** Single-layer perceptron is a function t from vector of k real numbers to vector of l real numbers where each value in the result vector is computed by a signle perceptron.

$$t : \mathbb{R}^m \to \mathbb{R}^n$$

$$t(\vec{x}) = (p_1(\vec{x}), p_2(\vec{x}), \ldots, p_n(\vec{x}))$$

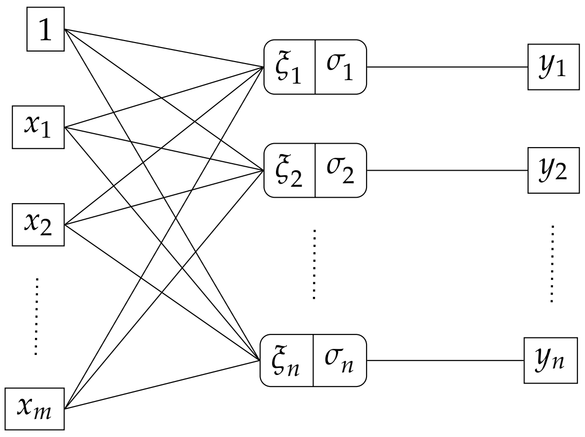

Figure 2.2: Schema of a single-layer perceptron

Definition 2.1.3 (Multi-layer perceptron)**.** Multi-layer perceptron (also called deep neural network) is a convolution of Single-layer perceptrons. The last applied layer t_{d+1} is called the output layer, all other layers are called hidden layers. The input of a multi-layer perceptron is called the input layer.

$$\mathcal{N} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_{d+1}}$$

$$\mathcal{N} = t_{d+1} \circ t_d \circ \ldots \circ t_1$$

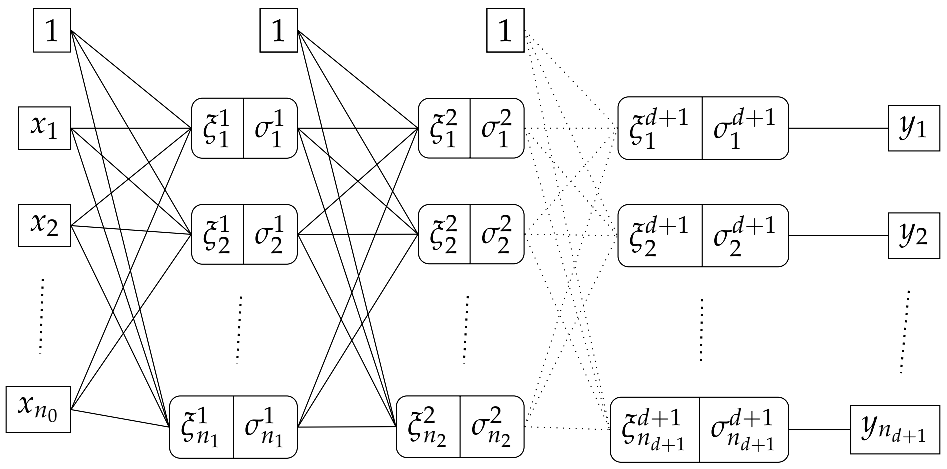
1
1
1
x_1
x_2
x_{n_0}
ξ_1^1 σ_1^1
ξ_2^1 σ_2^1
$\xi_{n_1}^1$ $\sigma_{n_1}^1$
ξ_1^2 σ_1^2
ξ_2^2 σ_2^2
$\xi_{n_2}^2$ $\sigma_{n_2}^2$
ξ_1^{d+1} σ_1^{d+1}
ξ_2^{d+1} σ_2^{d+1}
$\xi_{n_{d+1}}^{d+1}$ $\sigma_{n_{d+1}}^{d+1}$
y_1
y_2
$y_{n_{d+1}}$

Figure 2.3: Schema of a multi-layer perceptron

2.1.2 Binarised neural network

As the computation of general multi-layer perceptron depends on slow multiplication of floating-point numbers, came with an idea of binarized perceptron. This type of perceptron constrains the input vector, Binarized perceptron constraints the input space and vector of weights to vectors of binary values -1 and 1. The activation function is usually a heavyside step function H.

$$H(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

The computation of this constrained perceptron is faster than of the general perceptron as multiplication of two ± 1-binarized values can be done with a single *XOR* gate (-1 is equivalent to 1, 1 is equivalent to 0).

From binarized perceptrons, multi-layered perceptrons can be built similiarly to the general perceptron. In the case of the categorisation problem, argmax layer consisting of weighted sums and *argmax* operator is used as the output layer to find the maximal output.

Further follows formal definition of binarised neural network. I also show that every binarised perceptron defined using parameters as in the definition from [1] can be translated into binarised perceptron according to my definition and vice versa. This shows the equivalence of my definition to that of [1].

Definition 2.1.4 (Binarised perceptron)**.** Binarised perceptron $p^{\mathbb{B}}$ is a function from vector of m ± 1-binarised values to a single ± 1-binarised value. The function $p^{\mathbb{B}}$ is a composition of an inner potential ξ and a heavyside step function H. The inner potential ξ is weighted sum parametrized by static vector of ± 1-binarised values $\vec{w}$ of size k, and real bias b.

$$p^{\mathbb{B}} : \mathbb{B}^k \to \mathbb{B}$$
$$\xi : \mathbb{B}^k \to \mathbb{R},\ H : \mathbb{R} \to \mathbb{B}$$
$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot x_i$$
$$p^{\mathbb{B}} = H \circ \xi$$

Definition 2.1.5 (Binarised perceptron with batch normalization)**.** Binarised perceptron with batch normalization $\hat{p}^{\mathbb{B}}$ is a function from vector of m ± 1-binarised values to a single ± 1-binarised value. The function $p^{\mathbb{B}}$ is a composition of an inner potential ξ, batch normalization function ρ a heavyside step function H. The inner potential ξ is weighted sum parametrized by static vector of ± 1-binarised values $\vec{w}$ of size k, and real bias b. The batch normalization function ρ is a function on real numbers parametrized by real values α, γ, μ, σ.

$$\hat{p}^{\mathbb{B}} : \mathbb{B}^k \to \mathbb{B}$$
$$\xi : \mathbb{B}^k \to \mathbb{R},\ \rho : \mathbb{R} \to \mathbb{R},\ H : \mathbb{R} \to \mathbb{B}$$
$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot x_i$$
$$\rho(x) = \alpha \cdot \left(\frac{x - \mu}{\sigma} \right) + \gamma$$

$$\hat{p}^{\mathbb{B}} = H \circ \rho \circ \xi$$

Lemma 2.1.1. *For every Binarised perceptron there is equivalent Binarised perceptron with batch normalization.*

Proof. If the parameters of batch normalization function ρ are set to be $\alpha = \sigma = 1, \gamma = \mu = 0$, function ρ is identity function. With parameters of inner potential ξ unchanged, the following holds

$$p^{\mathbb{B}} = H \circ \xi = H \circ \mathrm{id} \circ \xi = H \circ \rho \circ \xi = \hat{p}^{\mathbb{B}}$$

$\square$

Lemma 2.1.2. *For every Binarised perceptron with batch normalization there is equivalent Binarised perceptron.*

Proof. The idea behind this construction comes from [1].

The value of $\hat{p}^{\mathbb{B}}(\vec{x})$ is only determined by the sign of expression $(\rho \circ \xi)(\vec{x})$. Lets thus analyse the inequality $(\rho \circ \xi)(\vec{x}) \geq 0$.

$$(\rho \circ \xi)(\vec{x}) = \alpha \cdot \left(\frac{b + \sum_{i=1}^{k} w_i \cdot x_i - \mu}{\sigma} \right) + \gamma$$

If $\alpha = 0$, then the expression $\rho \circ \xi$ is a constant function. In that case the perceptron is equivalent to the one with its bias bigger than the length of input vector for positive constant perceptron or with bias lower than the length of input vector for negative constant perceptron.

If $\alpha \neq 0$, the expression can be divided by the term $\frac{\alpha}{\sigma}$. In the case of this term being negative, the inequality switches and has to be corrected by further multiplying by -1.

$$(\rho \circ \xi)(\vec{x}) \cdot \frac{\sigma}{\alpha} = b + \sum_{i=1}^{k} w_i \cdot x_i - \mu + \frac{\sigma \cdot \gamma}{\alpha}$$

$$(\rho \circ \xi)(\vec{x}) \cdot \frac{\sigma}{\alpha} = (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i$$

$$\frac{\alpha}{\sigma} > 0 : (\rho \circ \xi)(\vec{x}) \geq 0 \iff (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i \geq 0$$

$$\frac{\alpha}{\sigma} < 0 : (\rho \circ \xi)(\vec{x}) \geq 0 \iff (b - \mu + \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} w_i \cdot x_i \leq 0$$

$$\iff (-b + \mu - \frac{\sigma \cdot \gamma}{\alpha}) + \sum_{i=1}^{k} -w_i \cdot x_i \geq 0$$

As can be seen, in the positive case the expression breaks into two parts, the new bias $(b - \mu + \frac{\sigma \cdot \gamma}{\alpha})$ and unchanged weighted sum $\sum_{i=1}^{k} w_i \cdot x_i$. In the negative case, both the bias and vector of weights are negative.

The perceptron without batch normalization for the positive resp. negative case consists of bias $(b - \mu + \frac{\sigma \cdot \gamma}{\alpha})$ resp. $(-b + \mu - \frac{\sigma \cdot \gamma}{\alpha})$ and weight vector of $\vec{w}$ resp. $-\vec{w}$. □

Remark. The proof of lemma above is constructive and is used for encoding of the quantitative verification problem as ASP problem.

Remark. As $\sum_{i=1}^{k} w_i \cdot x_i$ is always a whole number, bottom whole part of bias $\lfloor b \rfloor$ can be used in place of bias in inner layers of BNN. In the output layer however the fractional part can make difference when choosing the maximal input.

Definition 2.1.6 (Binarised single-layer perceptron)**.** Binarised single-layer perceptron is a function $t^{\mathbb{B}}$ from vector of m ± 1-binarised numbers to vector of n ± 1-binarised numbers where each value in the result vector is computed by a signle perceptron.

$$t^{\mathbb{B}} : \mathbb{B}^m \to \mathbb{B}^n$$
$$t^{\mathbb{B}}(\vec{x}) = (p_1^{\mathbb{B}}(\vec{x}), p_2^{\mathbb{B}}(\vec{x}), \ldots, p_n^{\mathbb{B}}(\vec{x}))$$

Definition 2.1.7 (Argmax layer)**.** Argmax layer is a function t^{am} which returns the mask of maximal value after the weighted sum. This mask has form of a one-hot vector, where only the first position with the maximal value after the weighted sum is assigned value 1, all the other positions are assigned value 0.

$$t^{am} : \mathbb{B}^m \to \{0, 1\}^n$$
$$t^{am}(\vec{x}) = y, y_k = 1 \iff k = \arg\max_{i=1}^{n}(\xi_i(\vec{x}))$$

Definition 2.1.8 (Binarised multi-layer perceptron)**.** Binarised multi-layer perceptron (also called binarised neural network) is a convolution of binarised single-layer perceptrons. The last applied layer t^{am} is called the output layer and takes form of argmax layer, all other layers are called hidden layers. The input of a binarised multi-layer perceptron is called the input layer.

$$\mathcal{N}^{\mathbb{B}} : \mathbb{B}^{n_0} \to \{0,1\}^{n_{d+1}}$$
$$\mathcal{N}^{\mathbb{B}} = t^{am} \circ t_d^{\mathbb{B}} \circ \ldots \circ t_1^{\mathbb{B}}$$

2.2 Robustness of Binarised neural network

There are two types of robustness problems. The qualitative robustness is a problem to determine wether the neural network gives the same (true) output for all inputs in some input region. The quantitative rubustness on the other hand determines the part of this input region, which has the same output as some chosen input of this region.

Further I provide formal definition of robustness on functions in general.

2.2.1 Definition of robustness

Definition 2.2.1 (Quantitative robustness of function)**.** Let F be a function, $F : P \to Q$. Let I be an input region of F, that is $I \subseteq P$. Let w be a function, $w : P \to \mathbb{R}, \forall x \in I : w(x) > 0$. Let h_p be a function for some base input $p \in P$, $h_p : Q \to \mathbb{R}$, that satisfies

$$h_p(F(p)) = 0$$

$$\forall x \in Q. 0 \leq h_p(x) \leq 1$$

Let $Q_{d,h_p}(I)$ be equal to

$$Q_{w,h_p}(I) = \frac{\int_I w(i) \cdot h_p(F(i))\, di}{\int_I w(i)\, di} \quad \text{or} \quad Q_{w,h_p}(I) = \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)}$$

if the input region is non-discrete or discrete respectively. Then $Q_{w,h_p}(I)$ is called the quantitative robustness of function F with a weight function w and evaluation function h_p on input region I.

For empty input region $I = \emptyset$ or non-discrete input regions I such that $\int_I w(i)di = 0$, quantitative robustness is equal to 0.

The quantitative robustness is an average of values aquired by aplying the evaluation function h_p on the input region weighted by the weight function w. The lower the value of robustness is, the more the function is robust on input region.

The weight function can be used to make part of the inputs more prominent. For instance, by the use of weight function such as $w(x) = \frac{1}{1+||p-x||}$, inputs closer to the base input will have higher weight.

The evaluation function can be used to encode dissimiliarity of an input from the base input p. That could prove useful in case of external metric.

The simplest example of the weight function is a constant function $w_1(x) = 1$. With constant function, every input is assigned the same weight, which leads to the quantitative robustness being an average of evaluation function h_p applied over all inputs from the input region.

More complex weight functions can be used to give some areas of the input region higher priority. Such weight function may reflect requirement for the robustness closer to the base input p.

Lemma 2.2.1. *For input region consisting only of the base input $I = \{p\}$, the quantitative rubustness is equal to 0*

Proof.

$$Q_{w,h_p}(\{p\}) = \frac{w(p) \cdot h_p(p)}{w(p)} = \frac{w(p)}{w(p)} \cdot 0 = 0$$

□

Lemma 2.2.2. *Quantitative robustness is between 0 and 1 for every possible combination of functions and input regions.*

Proof. Let me first show that the quantitative robustness is nonnegative. Both w and h_p are nonnegative on P and Q respectively, thus $w(x) \cdot h_p(y)$ is nonnegative for all $(x, y) \in P \times Q$. For that also values of integrals and summations over both $w(x)$ and $w(x) \cdot h_p(y)$ are nonnegative. The fraction of two nonnegative values is nonnegative and the quantitative robustness is nonnegative.

By definition holds $0 \leq h_p(y) \leq 1$, thus also $0 \leq h_p(F(x)) \leq 1$. As $w(x) \geq 0$,

$$0 = w(x) \cdot 0 \leq w(x) \cdot h_p(F(x)) \leq w(x) \cdot 1$$

Again, integration or summation over any subset $I \subseteq P$ can be applied onto the latter two expressions.

$$0 \leq \int_I w(i) \cdot h_p(F(i))\, di \leq \int_I w(i)\, di$$

$$0 \leq \sum_{i \in I} w(i) \cdot h_p(F(i)) \leq \sum_{i \in I} w(i)$$

If the right-hand side term is equal to 0, the statement holds by definition of qualitative robustness. Else by division by the right-hand side term (nonnegative), following holds

$$0 \leq \frac{\int_I w(i) \cdot h_p(F(i))\, di}{\int_I w(i)\, di} \leq 1$$

$$0 \leq \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)} \leq 1$$

$\square$

While the quantitative robustness does tell us something about how much of the input region is evaluated wrong (or even how much is it wrong), the qualitative robustness does only say if there is any wrong output. This may seem less useful, however it can lead to lower computational expenses.

Definition 2.2.2 (Qualitative robustness of function F on input region I)**.** Function $F : P \to Q$ is (qualitatively) robust on input region $I \subseteq P$ with respect to evaluation function h_p if and only if $Q_{w,h_p}(I) = 0$.

Lemma 2.2.3. *The property of qualitative robustness is independent of the function w.*

Proof. For empty input region, the lemma holds trivially by definition.

Otherwise as weight function w is by definition positive, all statements of the following chain are equivalent.

$$Q_{w,h_p}(I) = \frac{\sum_{i \in I} w(i) \cdot h_p(F(i))}{\sum_{i \in I} w(i)} = 0$$

Since $\sum_{i \in I} w(i) > 0$:

$$\sum_{i \in I} w(i) \cdot h_p(F(i)) = 0$$

As both w and h_p are non-negative

$$\forall i \in I.\, w(i) \cdot h_p(F(i)) = 0$$

$$\forall i \in I.\, w(i) = 0 \vee h_p(F(i)) = 0$$

From the definition $\forall x \in P.\, w(x) > 0$

$$\forall i \in I.\, h_p(F(i)) = 0$$

This statement is independent of the function w, thus the lemma holds for the discrete variation.

The non-discrete variant can be proven similarly. □

The robustness according to my definition can be contraintuitive when applied to functions that are not continuous on the input interval. An extreme example of such function is the Dirichlet function $D : \mathbb{R} \to \mathbb{R}$.

$$D(x) = \begin{cases} 1 & \text{if } x \in \mathbb{Q} \\ 0 & \text{if } x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

Lemma 2.2.4. *The Dirichlet function D is robust on any interval $\langle k, l \rangle$ where k, l are rational numbers, $k \neq l$.*

Proof. As shown in the Example 3.1.1 of [5], the Dirichlet function has Lebesgue integral on interval $[0, 1]$ with a value equal to 0.

Lets prove that for every rational number $a \in \mathbb{Q}$ and real number $b \in \mathbb{R}$ following statements are equivalent:

1. b is rational

2. $a + b$ is rational

1. $\implies$ 2.:
Since both a and b are rational, by definition they can be written as a fraction of integers

$$a = \frac{p}{q}, b = \frac{p'}{q'}$$

The sum can be expressed as a fraction of integers, thus is also rational.

$$a + b = \frac{pq' + p'q}{qq'}$$

2. $\implies$ 1.:
The b can be written using the rational a and the sum of a and b.

$$b = (a + b) + (-a)$$

Now since both $(-a)$ and $(a + b)$ are rational, b was already proven to be rational in the opposite side implication.

Lets show that the integral of Dirichlet function D is equal to 0 on any interval bounded by rational numbers. The limits of the definite integral can be transformed by removing k and adding it to the argument of D. As was proven previously, for rational number k, $D(x + k) = D(x)$.

$$\int_k^l D(x)dx = \int_0^{l-k} D(x + k)dx = \int_0^{l-k} D(x)dx$$

As the dirichlet function is nonnegative, following is true.

$$0 \leq \int_0^{l-k} D(x)dx \leq \int_0^{\lceil l-k \rceil} D(x)dx$$

The right-hand side integral can be split into unit-long parts.

$$\int_0^{\lceil l-k \rceil} D(x)dx = \int_0^1 D(x)dx + \int_1^2 D(x)dx + \ldots + \int_{\lceil l-k \rceil -1}^{\lceil l-k \rceil} D(x)dx$$

Finally, as every unit integral has rational bounds, it can be transformed to integral with 0 as lower bound like already shown.

$$\int_u^{u+1} D(x)dx = \int_0^1 D(x + u)dx = \int_0^1 D(x)dx = 0$$

$$0 \leq \int_k^l D(x)dx \leq \int_0^{\lceil l-k \rceil} D(x)dx = 0$$

To show the robustness of Dirichlet function, the quantitative robustness with respect to constant weight function w_1 and identity as the evaluation function can be used.

$$Q_{w_1,id}(\langle k,l \rangle) = \frac{\int_k^l 1 \cdot D(x)dx}{\int_k^l 1} = \frac{0}{l-k} = 0$$

$\square$

Since this thesis focuses on robustness of discrete input regions, I will further assume only discrete version of the robustness problem.

Definition 2.2.3 (Strict qualitative robustness of function F on input region I)**.** Function $F : P \to Q$ is strictly (qualitatively) robust on input region $I \subseteq P$ if and only if for some $p \in P$ it is robust on this region with respect to evaluation function $\overline{h_p}$ defined as follows:

$$\overline{h_p}(q) = \begin{cases} 0 & F(p) = q \\ 1 & F(p) \neq q \end{cases}$$

Lemma 2.2.5. *Function F is strictly robust on input region I if and only if for each two inputs $i, j \in I$, the function F assigns the same value to them.*

Proof. Proof of equivalence in this lemma is done by prooving corresponding implications.

For empty input region $I = \emptyset$, the statement holds trivially. Further in the proof I will always assume nonempty input region.

First the left-to-right implication. Let function F be strictly robust on input region I, $\overline{h_p}$ being the evaluation function. As shown in the proof of Lemma 2.2.3, for all instances from input region $i \in I$ holds

$$\overline{h_p}(F(i)) = 0$$

By definition of evaluation function $\overline{h_p}$ it also holds

$$F(p) = F(i)$$

As this holds for every input $i \in I$, for every two inputs $i, j \in I$:

$$F(i) = F(p) = F(j)$$

Now for the opposite implication: Let for every $i, j \in I$, $F(i) = F(j)$. Let $p \in I$. As $\overline{h_p}(F(p)) = 0$ and for every input $i \in I$ it holds that $F(i) = F(p)$, $\overline{h_p}(F(i)) = 0$ for every input from the input region I. The F is thus strictly robust on input region I. □

The strict robustness is equivalent to the robustness as defined in [6]. The t-target robustness from this article is equivalent to my definition of robustness with respect to evaluation function $h_t : Q \to \mathbb{R}$

$$h_t(q) = \begin{cases} 0 & \text{if } q \neq t \\ 1 & \text{if } q = t \end{cases}$$

Finally the term $Pr(R(u, \tau))$ is equal to quantitative robustness with respect to weight function w_1 and evaluation function h_u on input region $R(u, \tau)$.

2.2.2 Definition of input regions

As both the qualitative and quantitative robustness rely on subsets of feasible inputs, definition of these is needed.

I provide definition of two classes of input regions, input regions based on the Hamming distance and input regions with fixed indices. The input region based on the Hamming distance $R(\vec{u}, r)$ contains all input vectors that have at most r bits changed. The input region with fixed indices $R(\vec{u}, I)$ specifies set of indices I on which the input vector may differ form I. Definition are taken from [1].

Definition 2.2.4 (Input region based on the Hamming distance)**.** For an input $\vec{u} \in \mathbb{B}_{\pm 1}^{n_1}$ and an integer $r \geq 0$, let $R(\vec{u}, r) := \{\vec{x} \in \mathbb{B}_{\pm 1}^{n_1} \mid HD(\vec{x}, \vec{u}) \leq r\}$, where $HD(\vec{x}, \vec{u})$ denotes the Hamming distance between $\vec{x}$ and $\vec{u}$.

Intuitively, $R(\vec{u}, r)$ includes input vectors that differ from $\vec{u}$ on at most r positions. Examples of such input regions are:

$$\begin{aligned} R((1,1,1,1),1) = \{&(1,1,1,1), \\ &(-1,1,1,1),(1,-1,1,1),(1,1,-1,1),(1,1,1,-1)\} \end{aligned}$$

$$\begin{aligned} R((-1,1,-1),2) = \{&(-1,1,-1), \\ &(1,1,-1),(-1,-1,-1),(-1,1,1), \\ &(-1,-1,1),(1,1,1),(1,-1,-1)\} \end{aligned}$$

Lemma 2.2.6. *Let $\vec{u}$ be a vector from $\mathbb{B}^d_{\pm 1}$. Then $||R(\vec{u}, r)|| = \sum_{i=0}^{\min(r,d)} \binom{d}{i}$*

Proof. $R(\vec{u}, r)$ is union of sets

$$R(\vec{u}, r) = \bigcup_{i=0}^{r} \{\vec{x} \mid HD(\vec{x}, \vec{u}) = i\}$$

These sets are disjoint as elements of each have different number of positions changed. The size of each of these sets is equal to

$$||\{\vec{x} \mid HD(\vec{x}, \vec{u}) = i\}|| = \binom{d}{i}$$

because they consist of d positions, out of which i are choosen to be altered. Finally, as for i larger than d, $\binom{d}{i} = 0$, the statement holds. $\square$

Definition 2.2.5 (Input region based on fixed bits)**.** For an input $\vec{u} \in \mathbb{B}^{n_1}_{\pm 1}$ and set of indices $I \subseteq [n_1]$, let $R(\vec{u}, I) := \{\vec{x} \in \mathbb{B}^{n_1}_{\pm 1} \mid \forall i \in I.\, x_i = u_i\}$.

The input region based on fixed bits $R(\vec{u}, I)$ does specify the positions which are fixed to the values of the base vector $\vec{u}$. Examples of such input regions are:

$$R((1,1,-1,1), \{1,3,4\}) = \{(1,1,-1,1), (1,-1,-1,1)\}$$

$$\begin{aligned} R((1,1,1,1),\{3\}) = \{&(1,1,1,1), (-1,1,1,1), (1,-1,1,1), (-1,-1,1,1), \\ &(1,1,1,-1), (-1,1,1,-1), (1,-1,1,-1), (-1,-1,1,-1)\} \end{aligned}$$

Lemma 2.2.7. *Let $\vec{u}$ be a vector from $\mathbb{B}^d_{\pm 1}$, $I \subseteq [n_1]$. Then $||R(\vec{u}, I)|| = 2^{d - ||I||}$*

Proof. Each element of I does fix a single position in the input vector. The number of variable positions is $d - ||I||$, each being instance of $+1$ or -1. This makes the number of variants $2^{d-||I||}$. $\square$

Remark. Lemmas 2.2.6 and 2.2.7 allow for fast computation of the size of the input region.

3 Answer set programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems [7]. ASP is particularly suited for solving difficult combinatorial search problems [8]. ASP is somewhat closely related to propositional satisfability checking (SAT) in sense that the problem is represented as logic program. Difference is in computational mechanism of finding solution.

3.1 Extended logic program

Extended logic program is a finite set of facts, rules and constraints each consisting literals. A literal is an atom or its negation. An atom is the elementary construct for representing knowledge [9]. Each atom constitutes a single variable, it can be seen as a possible feature of solution.

In order to be able to distinguish between a query which fails in the sense that it does not succeed and a query which fails in the stronger sense that its negation succeeds, extended logic programs allow for the *classical negation* $\neg$ in addition to the *negation-as-failure not* [10].

Further I provide formal definition of extended logic programs according to [10, 11]. I will assume the notation of signature as defined in [12].

3.1.1 Syntax of Extended logic program

Definition 3.1.1 (Terms [12])**.** Let Σ be a signature. The set of Σ-terms of sort σ is the smallest set of expressions satisfying the following properties:

- Each variable x of sort σ is a term of sort σ, provided that $\sigma \in \Sigma^S$.
- Each constant symbol $c \in \Sigma^C$ of sort σ is a Σ-term of sort σ.
- If $f \in \Sigma^F$ is a function symbol of arity $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ and t_i is a Σ term of σ_i, for $i = 1, \ldots, n$, then $f(t_1, \ldots, t_n)$ is a term of sort σ.

Definition 3.1.2 (Atoms, literals [12])**.** Let Σ be a signature. A Σ-atom is an expression of the form

$$p(t_1, \dots, t_n)$$

where $p \in \Sigma^P$ is a predicate symbol of arity $\sigma_1 \times \cdots \times \sigma_n$ and for $i = 1, \dots, n$, t_i is a Σ-term of sort σ_i.

A Σ-literal is a formula of the form

$$\varphi \text{ or } \neg\varphi$$

where φ is a Σ-atom.

For both function and predicate symbols infix notation can be used if was estabilished (e.g. $+, -, =, \leq, \dots$). If this notation would result in ambiguity, brackets should be used.

Example 3.1.1. Let Σ_1 be a signature with sorts

$$\sigma_{\mathbb{B}} = \{True, False\}, \sigma_{\mathbb{N}} = \mathbb{N}_0$$

and with function symbols $\vee, \wedge$ of arity $\sigma_{\mathbb{B}} \times \sigma_{\mathbb{B}} \to \sigma_{\mathbb{B}}$, and function symbols $=, <$ of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{N}} \to \sigma_{\mathbb{B}}$. Let p be a predicate of arity $\sigma_{\mathbb{B}}$, and q be a predicate of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{B}}$ Also let variables X, Y, Z be of sort $\sigma_{\mathbb{N}}$ and let there be no variables of sort $\sigma_{\mathbb{B}}$.
Then expressions

$$\begin{gathered} True,\ False, \\ True \vee False,\ False \wedge True, \\ (42 = 42),\ (X = 17),\ (3 < Y) \vee (6 = 6), \\ [(5 = 1) \wedge True] \vee False \end{gathered}$$

are some of the Σ-terms of sort $\sigma_{\mathbb{B}}$.
Expressions

$$\begin{gathered} p(True),\ p(False), \\ p([(5 = 1) \wedge True] \vee False), \\ q(0, True),\ q(42, False), \\ q(123, (42 = 42)),\ q(93, True \vee (Z < 17)) \end{gathered}$$

are some of the Σ-atoms.

Definition 3.1.3 (Rule [11]). Let Σ be a signature. A Σ-rule (Σ-formula) r is an expression of the form

$$l_0 \, or \, \ldots \, or \, l_k \leftarrow l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n.$$

where each l_i is a Σ-literal. The following notation is used:

$$\text{head}(r) = \{l_0, \ldots, l_k\}$$
$$\text{body}(r) = \{l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n\}$$
$$\text{body}^+(r) = \{l_{k+1}, \ldots, l_m\}$$
$$\text{body}^-(r) = \{\text{not}\, l_{m+1}, \ldots, \text{not}\, l_n\}$$

Further, if $\text{head}(r) = \emptyset$, the rule is called a constraint and is written as

$$\leftarrow l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n.$$

If $\text{body}(r) = \emptyset$, the rule is called a fact and is written as

$$l_0 \, or \, \ldots \, or \, l_k.$$

The term not is called default negation.

Definition 3.1.4 (Logic program [11]). Logic program is a pair (Σ, Π) where Σ is a signature and Π is a collection (set) of Σ-rules.

In this thesis, I will be mostly working with logic programs that have zero or one literal in the head of each of their rules. While allowing for multiple literals in the head of a rule can allow for otherwise impossible to achieve expressions, it does introduce a large portion of nontriviality and even computational cost. General logic programs are in the complexity class Σ_2^P [13], logic programs without disjunction in head are NP-complete [14] and logic programs with neither disjunction nor default negation belong to P [11].

Example 3.1.2. An example of a logic program with signature Σ_1 from Example 3.1.1 is a pair (Σ_1, Π_1) where Π_1 is a set

$$\Pi_1 = \begin{cases} p(True). \\ q(5, True). \\ p(False) \leftarrow \text{not}\, p(True). \\ q(Y, True) \leftarrow q(X, (Y < X)). \end{cases}$$

The logic program is often denoted only by its second element Π. In that case its signature consists of symbols occuring in the program.

3.1.2 Semantics of extended logic program

For the definition of the extended logic programs to be usefull, definition of stable models is needed. Many definitions have been created [15]. Here I will show the definition using the reduct of a logic program [11], which was used for the ASP solver I am using for the implementation of verification, Clingo [16].

First of all, the logic program needs to be grounded. Grounding is a process in which we substitute each rule in the program for its equivalent rules with constant symbols only.

The semantics of function symbols is usually defined in the grounder (the program that does the grounding) thus the programs can be grounded before the start of solving of the logic program.

Definition 3.1.5 (Ground program [11])**.** Terms, literals, and rules of program Π with signature Σ are called ground if they contain no variables and no function symbols. A program is called ground if all its rules are ground. A rule r' is called a ground instance of a rule r of Π if it is obtained from r by:

- replacing r's variables by properly typed ground terms of Σ;
- replacing r's function terms by their values.

A program $gr(\Pi)$ consisting of all ground instances of all rules of Π is called the ground instantiation of Π.

Example 3.1.3. The ground instantiation of the logic program (Σ_1, Π_1) defined in the Example 3.1.2 can be found as follows:

The fact $p(True)$. does not contain any variable nor function symbol. For this, it is ground. The same holds for the fact $q(5, True)$. and the rule $p(False) \leftarrow \text{not}\, p(True)$.

To ground the rule $q(Y, True) \leftarrow q(X,(Y < X))$., this rule should be first substitued for each rule with pair (X, Y) substitued for every element from $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{N}}$, that is every element from $\mathbb{N}_0 \times \mathbb{N}_0$. Then in the second parameter of atom in the body, the expression substitued for $(Y < X)$ would be replaced with *True* or *False*, depending on the substitution for X and Y. This would lead to the ground instantiation $gr(\Sigma_1, \Pi_1)$ being infinite.

To fight this, we can allow only the instantiation of rule r in which only the substitutions of variables in the body$^+(r)$ leading to the already existing atoms are allowed. After the stable model will be defined, it will be easy to see that this constraint does not remove any stable models of program, as every atom in it has to also be in the head of some rule.

In this example, we already know that the atom $p(5, \mathit{True})$ does exist. This means we can substitute $X \equiv 5$ and $(Y < X) \equiv \mathit{True}$. Also as $(Y < X) \equiv (Y < 5) \equiv \mathit{True}$, to successfully instantiate the rule, it has to hold that $Y < 5$. There are no other constraints on the rule, thus the rule instantiates into rules:

$$\begin{array}{rcl}
Y = 0 & : & q(0, \mathit{True}) \leftarrow q(5, \mathit{True}). \\
Y = 1 & : & q(1, \mathit{True}) \leftarrow q(5, \mathit{True}). \\
Y = 2 & : & q(2, \mathit{True}) \leftarrow q(5, \mathit{True}). \\
Y = 3 & : & q(3, \mathit{True}) \leftarrow q(5, \mathit{True}). \\
Y = 4 & : & q(4, \mathit{True}) \leftarrow q(5, \mathit{True}).
\end{array}$$

But now we have introduced new atoms on predicate q. We have to also add all instances of the rule, in which these atoms are allowed. As the individual atoms only differ in the first term, the instances are seen easily. The instances added by the new rules also do not introduce new atoms on predicate q, the ground instance is thus in this case finite.

$$\begin{array}{rcl}
X = 4, Y = 0 & : & q(0, \mathit{True}) \leftarrow q(4, \mathit{True}). \\
X = 4, Y = 1 & : & q(1, \mathit{True}) \leftarrow q(4, \mathit{True}). \\
X = 4, Y = 2 & : & q(2, \mathit{True}) \leftarrow q(4, \mathit{True}). \\
X = 4, Y = 3 & : & q(3, \mathit{True}) \leftarrow q(4, \mathit{True}). \\
X = 3, Y = 0 & : & q(0, \mathit{True}) \leftarrow q(3, \mathit{True}). \\
X = 3, Y = 1 & : & q(1, \mathit{True}) \leftarrow q(3, \mathit{True}). \\
X = 3, Y = 2 & : & q(2, \mathit{True}) \leftarrow q(3, \mathit{True}). \\
X = 2, Y = 0 & : & q(0, \mathit{True}) \leftarrow q(2, \mathit{True}). \\
X = 2, Y = 1 & : & q(1, \mathit{True}) \leftarrow q(2, \mathit{True}). \\
X = 1, Y = 0 & : & q(0, \mathit{True}) \leftarrow q(1, \mathit{True}).
\end{array}$$

The full ground instantiation $gr((\Sigma_1, \Pi_1))$ is thus the program:

$$gr((\Sigma_1, \Pi_1)) = \begin{cases} p(True). \\ q(5, True). \\ p(False) \leftarrow \text{not}\, p(True). \\ q(0, True) \leftarrow q(5, True). \\ q(1, True) \leftarrow q(5, True). \\ q(2, True) \leftarrow q(5, True). \\ q(3, True) \leftarrow q(5, True). \\ q(4, True) \leftarrow q(5, True). \\ q(0, True) \leftarrow q(4, True). \\ q(1, True) \leftarrow q(4, True). \\ q(2, True) \leftarrow q(4, True). \\ q(3, True) \leftarrow q(4, True). \\ q(0, True) \leftarrow q(3, True). \\ q(1, True) \leftarrow q(3, True). \\ q(2, True) \leftarrow q(3, True). \\ q(0, True) \leftarrow q(2, True). \\ q(1, True) \leftarrow q(2, True). \\ q(0, True) \leftarrow q(1, True). \end{cases}$$

The Example 3.1.3 also illustrates the need for the logic programs to be well written. As was shown in this example, due to a single flawed rule, the size of the ground instantiation of the logic program is quadratic given the highest number in the first parameter of atoms of the predicate q.

Grounders differ in the exact algorithm of the grounding as well as in the allowed signature of the logic program. The exact full syntax and semantics of the ASP grounder GRINGO can be found in [17].

Definition 3.1.6 (Partial interpretation of a signature)**.** A partial interpretation S of a signature Σ is a set of Σ-literals in which for each Σ-atom φ there is either φ, $\neg\varphi$ or neither of them.

A partial interpretation can be used to denote the solution of a logic program. It allows for 3-valued logic — an atom either is true, is false or is unknown. The atom being unknown can specify an atom, that was not derived, while the atom being false strictly says it cannot be true.

For a partial interpretation to be a solution (answer set) of the program, it further needs to be consistent with the logic program and all its literals have to be founded in the program (the set has to be minimal).

Each rule r is seen as a statement $\text{head}(r)$ holds if all the literals from the $\text{body}^+(r)$ hold and at the same time no literal from the $\text{body}^-(r)$ hold. The facts have empty body, thus their head has to hold in every solution. On the other hand, the body of constraints can never be consistent with the solution as its head is empty thus can not be consistent with solution. Further follows a more formal definition.

Definition 3.1.7 (Consistent partial interpretation)**.** Partial interpretation S is said to be consistent with the body of rule r if and only if $\text{body}^+(r) \subseteq S$ and $\text{body}^-(r) \cap S = \emptyset$.

Partial interpretation S is said to be consistent with the head of rule r if and only if $\text{head}(r) \cap S \neq \emptyset$

Partial interpretation S is said to be consistent with a rule r if and only if S is not consistent with $\text{body}(r)$ or S is consistent with $\text{head}(r)$.

Partial interpretation S is said to be consistent with a logic program (Σ, Π) if and only if it is consistent with all of its rules.

Definition 3.1.8 (Basic logic program)**.** A logic program (Σ, Π) is called basic if there is no negative atom in body of any of its rules nor constraints, that is

$$\forall r \in \Pi.\ \text{body}^-(r) = \emptyset$$

Now for the answer sets (partial interpretations that are solutions) of a logic program a reasonable constraints on them are to satisfy (be consistent with) the logic program and to only contain information that needs to be true (each answer set has to be minimal).

Definition 3.1.9 (Answer set of a basic logic program [11])**.** Let (Σ, Π) be a basic logic program and S be a partial interpretation of Σ. S is an answer set (solution) for Π if S is minimal (in the sense of set-theoretic inclusion) among the partial interpretations consistent with Π.

For a grounded basic logic program, finding its answer set (solution) is easy. It can be built incrementally. Heads of facts have to be included in the partial interpretation for it to be an answer set. Then in each step all heads of rules, that have its bodies consistent with the previous set, are added into the set. This is done as long as any literal is added into the partial interpretation.

Lemma 3.1.1. *Incremental building of the answer set of the basic logic program. Let (Σ, Π) be a basic logic program with at most one literal in the head of every rule. Let*

$$S_0 = \emptyset,$$

$$S_{i+1} = S_i \cup \{\varphi \mid \{\varphi\} = \text{head}(r), \text{body}^+(r) \subseteq S_i\} \textit{ for each } i \geq 0.$$

Denote by S_∞ the union of all S_i for $i \geq 0$. If it holds that:

- *for every Σ-atom φ, S_∞ contains at most one of literals $\varphi, \neg\varphi$, and*
- *S_∞ is consistent with every constraint of Π,*

then S_∞ is the only answer set of program (Σ, Π). Else (Σ, Π) has no answer set.

Proof. S_∞ does contain only (some of) heads of rules from Π, each of these being Σ-literals. If S_∞ does also not contain both φ and $\neg\varphi$ for any Σ-atom φ, S_∞ is a partial interpretation of Σ.

By definition S_∞ contains the literal from the head of each rule, that has its body consistent with the S_∞. For this the S_∞ is consistent with every non-constraint rule of the logic program.

Let $rank : S_\infty \to \mathbb{N}_0$ be a function that assigns to every literal $\varphi \in S_\infty$ the lowest value n such that $\varphi \in S_n$. Let there be some answer set A of (Σ, Π) such that $S_\infty \setminus A = \Delta \neq \emptyset$. Let α be a literal from Δ with the lowest $rank$. By definition, $rank(\alpha) \geq 1$ as S_0 is empty. As the literal $\alpha \in S_{rank(\alpha)}$ and $\alpha \notin S_{rank(\alpha)-1}$, either α is in a head of some fact (thus $rank(\alpha) = 1$ and α needs to be in A for it to be consistent), or there must be some rule $r \in \Pi$ such that $\text{body}^+(r)$ is not consistent with $S_{rank(\alpha)-2}$ and is consistent with $S_{rank(\alpha)-1}$. (The rule r is what made the literal α in $S_{rank(\alpha)}$.) But A does contain all the literals from $S_{rank(\alpha)-1}$. (Literal α has the lowest rank of all literals from S_∞ not included in A.) Thus A is not consistent with r and is not an answer set of (Σ, Π). By contradiction, for every answer set B of (Σ, Π), $S_\infty \subseteq B$. As the answer set is by definition minimal, the S_∞ is the only possible answer set of (Σ, Π). □

Example 3.1.4. Let (Σ, Π) be a logic program,

$$\Pi = \begin{cases} p(a). \\ p(b) \leftarrow p(a). \\ p(c) \leftarrow p(b). \\ p(x). \\ p(y) \leftarrow p(x), p(a). \\ p(z) \leftarrow p(x), p(c). \\ p(n) \leftarrow p(a), p(k). \end{cases}$$

When building the solution, we first start with an empty set.

$$S_0 = \{\}$$

In the first step only the facts have bodies consistent with the partial solution S_0. We can add literals in their heads into the partial solution.

$$S_1 = \{p(a), p(x)\}$$

With these two literals in the partial solution, rules $p(b) \leftarrow p(a)$. and $p(y) \leftarrow p(x), p(a)$. have their bodies consistent with the partial solution S_1. For this we add literals in heads of these rules into the partial solution.

$$S_2 = \{p(a), p(x), p(b), p(y)\}$$

In the next step, another rule $p(c) \leftarrow p(b)$. has its body consistent with the partial solution S_2. We add the literal from its head into the partial solution.

$$S_3 = \{p(a), p(x), p(b), p(y), p(c)\}$$

This addition into the partial solution makes the body of yet another rule $p(z) \leftarrow p(x), p(c)$. to be consistent with the partial solution.

$$S_4 = \{p(a), p(x), p(b), p(y), p(c), p(z)\}$$

No other literals have to be added into the partial solution. S_4 is the only solution of logic program (Σ, Π).

The only literal from the heads of rules that have not been included into the partial solution is $p(n)$. This literal relies on some unknown literal $p(k)$ that is not in any head of rules of Π. Another case of literal that is not in a solution of logic program is shown in the following example.

Example 3.1.5. Let (Σ, Π) be a logic program,

$$\Pi = \begin{cases} p(a) \leftarrow p(b). \\ p(b) \leftarrow p(a). \end{cases}$$

This logic program a single solution, that is an empty set. While the partial interpretation $\{p(a), p(b)\}$ is consistent with the logic program, it is not the minimal set.

The simple way of building answer sets of a basic programs, can be extended to all logic programs. In basic programs we have assumed that no rule with negative literals does exist in the logic program. To fight the negative literals in a general extended logic program, we can first define a partial interpretation of Σ that constitutes the possible answer set, then take all rules $r \in \Pi$ that have no intersection of their $\text{body}^-(r)$ with our partial interpretation and into the reduced logic program include only the nonnegative $\text{body}^+(r)$ from them. If this reduced program yields the same answer set as the partial interpretation we have defined, it is an answer set.

Definition 3.1.10 (Reduct of a logic program [11])**.** Let (Σ, Π) be a logic program, S be a partial interpretation of Σ. Let Π^S be a set of rules such that

$$\Pi^S = \{\text{head}(r_i) \leftarrow \text{body}^+(r_i). \mid r_i \in \Pi, \text{body}^-(r_i) \cap S = \emptyset\}$$

Then Π^S is called a reduct of Π relative to the partial interpretation S.

Example 3.1.6. In the reduct can be encountered a new type of rule that was not yet discussed here. Let $\Pi = \{\leftarrow \text{not}\ \varphi.\}$, $S = \{\}$ for some atom φ. The reduct Π^S contains a single rule with both head and body empty. Based on the Definition 3.1.7, no partial interpretation can be consistent with such rule as no partial interpretation can be consistent with an empty head and every partial interpretation is consistent with an empty body. This is however a desired behaviour, the interpretation S was defined in a way it was not consistent with the rule so it is only right the reduct can not have any answer set.

Definition 3.1.11 (Answer set of general grounded logic program [11])**.** A partial interpretation S of Σ is an answer set for (Σ, Π) if S is an answer set for Π^S.

Definition 3.1.11 gives a direct way of computing answer sets of any logic programs. For every possible subset S of grounded atoms in program Π, the reduct Π^S can be constructed and evaluated for its answer set. If the found answer set is equal to the subset of grounded atoms S, it is said to be also an answer set of the program Π.

Let's illustrate the computation of an answer set of general logic program on two examples.

Example 3.1.7. Let (Σ, Π), where

$$\Pi = \{\varphi \leftarrow \varphi., \psi \leftarrow \text{not}\, \varphi.\},$$

be a logic program with two atoms φ, ψ. There are 4 subsets of set of all atoms. First the reduct relative to the subset is made, on it the calculation of its answer set (AS) is made. If the answer set of reduct is equal to the subset, it is an answer set of the whole logic program.

S	Π^S	AS of Π^S
$\{\}$	$\varphi \leftarrow \varphi.$ $\psi \leftarrow .$	$\{\psi\}$
$\{\varphi\}$	$\varphi \leftarrow \varphi.$	$\{\}$
$\{\psi\}$	$\varphi \leftarrow \varphi.$ $\psi \leftarrow .$	$\{\psi\}$
$\{\varphi, \psi\}$	$\varphi \leftarrow \varphi.$	$\{\}$

There is only a single answer set of (Σ, Π), that is $\{\psi\}$. Similiar to Example 3.1.5, in the reduct $\Pi^{\{\varphi\}}$ partial interpretation $\{\varphi\}$ is not an answer set as it is not minimal.

Example 3.1.8. Let (Σ, Π), where

$$\Pi = \{\varphi \leftarrow \text{not}\,\psi., \psi \leftarrow \text{not}\,\varphi.\},$$

be a logic program with two atoms φ, ψ. Again, there are 4 subsets of set of all atoms.

S	Π^S	AS of Π^S
$\{\}$	$\varphi \leftarrow .$ $\psi \leftarrow .$	$\{\varphi, \psi\}$
$\{\varphi\}$	$\varphi \leftarrow .$	$\{\varphi\}$
$\{\psi\}$	$\psi \leftarrow .$	$\{\psi\}$
$\{\varphi, \psi\}$		$\{\}$

This time there are two answer sets of (Σ, Π), $\{\varphi\}$ and $\{\psi\}$.

3.2 Clingo

Clingo is an integrated ASP system, consisting of a grounder Gringo and solver Clasp [18]. In the following section I will show basics of the Clingo language, Gringo as translation from Clingo to Aspif language and solving with Clasp.

3.2.1 Clingo language

Clingo language [19] is used for transcribing ASP programs in the Clingo system. The Clingo language does allow for facts, rules and constraints just like the logic programs defined in this thesis.

Example 3.2.1. Transcription of a logic program from the Example 3.1.4 to Clingo language. On the left side is a set of rules from this example, on the right is equivalent program in the Clingo language.

$$\Pi = \begin{cases} p(a). \\ p(b) \leftarrow p(a). \\ p(c) \leftarrow p(b). \\ p(x). \\ p(y) \leftarrow p(x), p(a). \\ p(z) \leftarrow p(x), p(c). \\ p(n) \leftarrow p(a), p(k). \end{cases}$$

```
p(a).
p(b) :- p(a).
p(c) :- p(b).
p(x).
p(y) :- p(x), p(a).
p(z) :- p(x), p(c).
p(n) :- p(a), p(k).
```

3.2.2 Basic syntax of Clingo language

Unless specified otherwise, source of information in this section is [19].

Terms in the Clingo language are of multiple sorts. The simplest sort of terms are constants and strings. Constants match the regular expression `_*[a-z][A-Za-z0-9_']*`. They start with underscores followed by lowercase letter and a sequence of characters. Similiarly, strings match the regular expression `"([^\"↩]|\[\"n])*"`. They allow for any reasonable text.

Another sort of Clingo language signature are integers. Constant integers can be captured by the regular expression `-?(0|[1-9][0-9]*)`. As can be seen, zero character can only be at the start of zero number. Number zero (`0`) and negative zero (`-0`) are the same constants. The range of integers is platform dependent, they are either 32-bit or 64-bit.

Grounder Gringo also built-in support for arithmetic functions over integers. Following symbols are used: `+` (addition), `-` (substraction), `*` (multiplication), `/` (integer division, rounding down), `\` (modulo), `**` (exponentiation), `|...|` (absolute value), `&` (bitwise AND), `?` (bitwise OR), `^` (bitwise XOR), and `~` (bitwise complement). Arithmetic functions use infix notation.

Gringo also allows for comparison predicates over all terms. These are predicates that are evaluated during grounding. The comparison predicates are `=` (equal), `!=` (not equal), `<` (less than), `<=` (less than or equal), `>` (greater than) and `>=` (greater than or equal). Integers are comared in the usual way, constants and strings are ordered lexicographically. All integers are smaller than constants which are smaller than strings. Additionaly two special constants `#sup` and `#inf` do exist, constituting to the greatest and lowest term.

Another sorts of terms are functions and tuples. Such terms take form of $f(t_1, \ldots, t_n)$ for the functions and $(t_1, \ldots, t_n)$ for the tuples where f is a function symbol and $t_1, \ldots, t_n$ are terms. The tuple has to have at least a single term long, in that case it is written as $(t_1,)$ to differentiate it from brackets. Function symbols are constrained in the same way as names of constants. Function with no terms as arguments is by the grounder seen as a constant with the function symbol as a name. Functions and tuples as terms do not have any interpretation. There is no mapping from function terms to some other terms as the name might suggest.

Finally, the last type of terms are variables. Variables can be matched with a regular expression `_*[A-Z][A-Za-z0-9_']*`. Variables are used to generalize rules of the logic program, a ground (variable-free) program is constructed during the process of grounding. A special variable `_` is called an annonymous variable or wildcard. At the start of the grounding process, a fresh (not used elsewhere) variable term is substitued for each annonymous variable. It stands for a placeholder, meaning any term can take its place.

An atom is syntatically consistent with a constant or a function. The predicate symbol starts with any number of underscores `_` followed by a lowercase letter and any number of alphanumeric characters, underscores and `'`. Atom using predicate symbol with arity of lenght zero can be written without the parentheses. One predicate symbol can be used with different arities.

A literal is an atom or its classical negation. In the Clingo language, classical negation is indicated by minus symbol `-` before the literal.

Finally, each ASP rule r of form

$$l_0 \; or \; \ldots \; or \; l_k \leftarrow l_{k+1}, \ldots, l_m, \text{not}\, l_{m+1}, \ldots, \text{not}\, l_n.$$

is transcribed into following expression of Clingo language:

$$l_0\texttt{; ...; } l_k \texttt{ :- } l_{k+1}\texttt{, ..., } l_m\texttt{, not } l_{m+1}\texttt{, ..., not } l_n\texttt{.}$$

Specially, if head(r) is nonempty and body(r) is empty, a shorthand transcription can be used:

$$l_0\texttt{; ...; } l_k\texttt{.}$$

In the basic notation of rules, semicolons ; and commas , can be freely interchanged. While this is true, I will try to stick to semicolons in heads and commas in bodies of rules for better readability. Where this is not possible (see Conditional literals), I will always point it out.

Example 3.2.2. Transcription of terms and atoms of Example 3.1.1.

In the example, signature Σ_1 had sorts

$$\sigma_{\mathbb{B}} = \{True, False\}, \sigma_{\mathbb{N}} = \mathbb{N}_0.$$

On these sorts, function symbols $\vee, \wedge$ of arity $\sigma_{\mathbb{B}} \times \sigma_{\mathbb{B}} \rightarrow \sigma_{\mathbb{B}}$, and function symbols $=, <$ of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{N}} \rightarrow \sigma_{\mathbb{B}}$ were defined. Also, there were two predicate symbols, p of arity $\sigma_{\mathbb{B}}$, and q of arity $\sigma_{\mathbb{N}} \times \sigma_{\mathbb{B}}$. Finally, there were 3 variables X, Y, Z of sort $\sigma_{\mathbb{N}}$.

Let the interpretation of these function symbols be the usual one, that is $\vee, \wedge$ are logical *OR* and *AND* and $=, <$ are *equal* and *less than*. In Clingo language, there is no such thing as boolean values. However, we can still use integers 0 and 1 in their place. Function symbols $\vee$ and $\wedge$ can then be modelled by using bit and and bit or arithmetic functions of the Clingo language.

Function symbols $=$ and $<$ are a bit more problematic. In the Clingo language as defined so far there are no user defined function[1]. When working with rules with these function symbols, we can however use comparison predicates to build multiple rules, each with its own assignment to the term value.

In the following, on the left side are Σ-terms of sort $\sigma_{\mathbb{B}}$, on the right side are their transcriptions into the Clingo language according to the above text. When the expression would need to be rewritten into multiple rules, different cases are inside curly brackets split by comma, for each case the substituted term is before the square brackets and added comparison predicates are inside the square brackets.

1. In fact there are External functions which use Python or Lua API to compute values on grounding. You can read about them in [19]. I will not be showing them here as I do not use them in my implementation.

True, False,	`1, 0,`
$\mathit{True} \vee \mathit{False}$, $\mathit{False} \wedge \mathit{True}$,	`1 ? 0, 0 & 1,`
$(42 = 42)$,	`{1 [42 = 42],`
	` 0 [42 != 42]},`
$(X = 17)$,	`{1 [X = 17],`
	` 0 [X != 17]},`
$(3 < Y) \vee (6 = 6)$,	`{1 ? 1 [3 < Y, 6 = 6],`
	` 1 ? 0 [3 < Y, 6 != 6],`
	` 0 ? 1 [3 >= Y, 6 = 6],`
	` 0 ? 0 [3 >= Y, 6 != 6]},`
$[(5 = 1) \wedge \mathit{True}] \vee \mathit{False}$	`{(1 & 1) ? 0 [5 = 1],`
	` (0 & 1) ? 0 [5 != 1]}`

Usage of these and other similiar Σ-terms in atoms is described in the following transcription. On the left side are some Σ-facts, on the right side is their transcription into the Clingo language.

$p(\mathit{True})$.	`p(1).`
$p(\mathit{False})$.	`p(0).`
$p([(5 = 1) \wedge \mathit{True}] \vee \mathit{False})$.	`p((1 & 1) ? 0) :- 5 = 1.`
	`p((0 & 1) ? 0) :- 5 != 1.`
$q(0, \mathit{True})$.	`q(0, 1).`
$q(123, (42 = 42))$.	`q(123, 1) :- 42 = 42.`
	`q(123, 0) :- 42 != 42.`

When transcripting rules of Example 3.2.2, we have made some facts into rules by introducing new comparison predicates into the rule. In the section about grounding, we will however see, that these predicates will be removed during the grounding process.

3.2.3 Extensions of ASP in Clingo language

Unless specified otherwise, source of information in this section is [19].

Clingo language allows for many extensions of the Answer set programming. In this section, I will describe extensions, which will be used in the implemention part of this thesis.

Intervals and Pooling

When defining rules, it is often desired to define rules on a range of values. For this type of rule, Clingo language allows for the use of intervals and pools. Intervals and pools can be used in place of terms in both heads and bodies of rules.

For two whole numbers i, j, an interval $i\,..\,j$ means, that during grounding a rule containing this interval expands to multiple rules, one for each k s.t. $i \leq k \leq j$ in its place. If there are multiple such intervals, they expand one by one, latter ones expanding in all rules already partially expanded by previous intervals.

Example 3.2.3. Consider the following program in Clingo language [19]:

```
grid(1..3, 1..3).
```

At the start of the grounding process, this program expands into the program:

```
grid(1, 1). grid(1, 2). grid(1, 3).
grid(2, 1). grid(2, 2). grid(2, 3).
grid(3, 1). grid(3, 2). grid(3, 3).
```

A pool is an expression in the form $(t_1;\ t_2;\ \ldots;\ t_n)$. Similiarly to the intervals during the grounding of a rule containing this pool, the rule expands to all rules containing $t_1, t_2, \ldots, t_n$ in place of this pool. If there are multiple pools they expand consecutively.

Example 3.2.4. Consider the following program in Clingo language:

```
grid((a; 3; #inf), (1; 2)).
```

At the start of the grounding process, this program expands into the program:

```
grid(a, 1).    grid(a, 2).
grid(3, 1).    grid(3, 2).
grid(#inf, 1). grid(#inf, 2).
```

Example 3.2.5. Consider the following program in Clingo language:

```
stairs(A, 1..A) :- A = 1..3.
```

When grounding this program, the grounder has to first expand the interval in the body. The head interval contains a variable `A` with value not yet known. After the first expansion, the following program emerges.

```
stairs(A, 1..A) :- A = 1.
stairs(A, 1..A) :- A = 2.
stairs(A, 1..A) :- A = 3.
```

Now there is only a single possible value of `A` in each expanded rule. `A` can be unified with this value and the head interval can be expanded. For better readability, I have removed comparison predicates from the body.

```
stairs(1, 1).
stairs(2, 1). stairs(2, 2).
stairs(3, 1). stairs(3, 2). stairs(3, 3).
```

Conditional literals

A conditional literal is an expression of the form:

$$l_0 \ : \ l_1, \ \ldots, \ l_n$$

where each of $l_0, l_1, \ldots, l_n$ are literals. This expression has a meaning of an inner implication. The conditional literal is consistent with a signature if and only if its head l_0 is consistent with this signature or its body $l_1, \ldots, l_n$ is not consistent with this signature. It can be seen as an expression *put l_0 in its place if all literals $l_1, \ldots, l_n$ are consistent with the partial solution*.

In the notation of conditional literals, use of commas `,` is compulsory. This is different from the rules, where commas and semicolons can be interchanged freely. When the conditional literal is not the last literal of a rule body, a semicolon must be used as an end of this conditional literal. Conditional literals can be quite nontrivial, especially when used in head of a rule.

Example 3.2.6. Consider following rules in Clingo language:

```
a :- b : c; d.
a :- b : c, d.
```

The first rule has a body composed of a conditional literal `b : c` and a literal `d`. For this rule an equivalent program without conditional literals would be following:

```
x :- b.
x :- not c.
a :- x, d.
```

In this logic program the conditional literal was replaced with a fresh literal `x`. This literal is true if either the head of conditional literal is consistent or the body is not consistent.

The only difference between first and second rule is the comma or semicolon between literals `c` and `d`. Second rule has a body consisting only of conditional literal `b : c, d`. An equivalent program to this rule would be following:

```
x :- b.
x :- not c.
x :- not d.
a :- x.
```

Example 3.2.7. Consider following rule in Clingo language:

```
a : b.
```

The rule consists of a single conditional literal in the head. If the literal `b` is not consistent with the partial solution, this rule acts as a rule with both head and body empty, thus invalidating any such solution. Rather than simply rewriting as a rule with `a` and `b` in the head and body respectively, the rule is equivalent to following program:

```
:- not b.
a :- b.
```

Aggregates

Aggregates are expressions of form

$$s_1 \prec_1 \alpha \; \{ \; t_1 : L_1 ; \; \dots ; \; t_n : L_n \; \} \prec_2 s_2,$$

where s_1, s_2 are terms, $\prec_1, \prec_2$ are comparison predicate symols, α is an aggregate symbol, $t_1, \dots, t_n$ are terms and $L_1, \dots, L_n$ are sets of literals.

The full expression can be split into multiple parts. First, there is set of (conditional) terms[2]

$$T \equiv \{ \; t_1 : L_1 ; \; \dots ; \; t_n : L_n \; \}.$$

2. To distinguish between a comparison predicate symbol of Clingo language and a equality in the normal means, in this subsection I will use $\equiv$ rather than $=$ for the second.

This set can be computed as a union of subsets

$$T \equiv \{t_1 : L_1\} \cup \ldots \cup \{t_n : L_n\}.$$

If any of these subsets contains variables, it can be grounded similiarly to rule. Grounded expressions can then be evaluated into sets of terms. The full expression is thus evaluated into set of terms containing all terms whose sets of literals are consistent with the partial solution.

Set T is evaluated in a set-like manner meaning if multiple its terms are the same, only a single one of them is included in the final set. When multiple occurences of the same term is needed, special form of terms can be used. This form is tuple-like, only without outer brackets.

Should some term t_i be included in every such set regardless of consistency of any atoms (that is $L_i = \emptyset$), both the colon and part with literals $:L_i$ may be omitted.

Example 3.2.8. Let S be a partial solution s.t. literals `p(1)`, `p(4)`, `p(5)` and `p(7)` are consistent with S.

The expression

$$T \equiv \texttt{\{ 1; 3; 4; a; x \}}$$

does not contain any variables nor literals, thus it is evaluated onto itself.

The expression

$$T \equiv \texttt{\{ 1:p(1); 3:p(3); 4:p(4), p(7) \}}$$

does contain literals. When evaluated, it is first looked at the literals if they are consistent with the partial solution. For the first conditional term, `p(1)` is consistent with S, thus `1` is included in the set. The second conditional term depends on literal `p(3)` which is not consistent with S, `3` is not included in the set. Finally, the last conditional term depends on literals `p(4)` and `p(7)`. Both of these literals are consistent with S, `4` is included in the set.

$$T \hookrightarrow \texttt{\{ 1; 4 \}}$$

The expression

$$T \equiv \texttt{\{ Y-X:p(X), p(Y), X <= Y \}}$$

does contain both literals and variables. First we can start by grounding the expression. We are looking for pairs of literals `p(X), p(Y)` that comply to the comparison predicate `X <= Y`.

$T \hookrightarrow$ `{ 0:p(1), p(1); 3:p(1), p(4); 4:p(1), p(5); 6:p(1), p(7); 0:p(4), p(4); 1:p(4), p(5); 3:p(4), p(7); 0:p(5), p(5); 2:p(5), p(7); 0:p(7), p(7) }`

Now as we only took tuples of literals satisfying both the rules from the expression and being consistent with the partial solution, to evaluate T we can directly take terms in the set.

$T \hookrightarrow$ `{ 0; 1; 2; 3; 4; 6 }`

In the evaluation of the last expression, we have lost the multiplicity of each term as it was evaluated as a set. To counteract that, we can add the lower parameter of each pair of literals to the terms.

$T \equiv$ `{ Y-X,X:p(X), p(Y), X <= Y }`
$T \hookrightarrow$ `{ 0,1; 3,1; 4,1; 6,1; 0,4; 1,4; 3,4; 0,5; 2,5; 0,7 }`.

An aggregate acts as a function over set of conditional terms, aggregate symbol α specifies the type of function. Currently allowed aggregate symbols are `#count`, `#sum`, `#sum+`, `#min` and `#max`. These stand for count of different terms, sum over terms, sum over positive terms, minimal term and maximal term respectively. Max and min aggregates do follow the same ordering as comparison predicates. If a term is composed of multiple parts, `#sum` and `#sum+` do their summation over the first part of term. If this part is not integer, the whole term is skipped. Summation over tuples or functions is not implemented. If the aggregate symbol α is omitted, it defaults to `#count`.

Aggregate returns a value in the form of a term. This value can then be compared with another term or assigned to variable. The same comparison predicate symbols can be used in place of $<_1, <_2$ as in comparison predicates. Both $s_1 \ <_1$ and $<_2 \ s_2$ can be omitted. If only $<_1$ or $<_2$ is omitted, comparison predicates defaults to `<=`.

Example 3.2.9. In this example we will work with expressions from Example 3.2.8. Let S be a partial solution s.t. literals `p(1)`, `p(4)`, `p(5)` and `p(7)` are consistent with S.

The rule

```
a(X) :- X = #count{ 1; 3; 4; a; x }.
```

adds a single atom `q(5)` into the solution as all elements of the aggregate set are terms. Here only a single left-hand comparison predicate is used. The right-hand predicate is omitted.

The rule

```
a :- 2 #sum{ 1:p(1); 3:p(3); 4:p(4), p(7) } <= 7
```

adds a single atom `q` into the solution. As already shown, the inner expression evaluates into

```
{ 1; 4 },
```

sum of this set is equal to 5, which satisfies comparison `2 <= 5 <= 7`. The left-hand side comparison predicate symbol is omitted, thus by default is <=.

Both rules

```
a(X) :- X = #max{ Y-X:p(X), p(Y), X <= Y }.
```

and

```
a(X) :- X = #max{ Y-X,X:p(X), p(Y), X <= Y }.
```

add a single atom `a(6)` into the solution. If tuple without brackets is used in the definition of term, the `#max` aggregate does take only the first element into account.

If both terms are needed, one can use the following rule:

```
a(X, Y) :- (X, Y) = #max{ (Y-X,X):p(X), p(Y), X <= Y }.
```

This time each term of the aggregate set is a single tuple. As written in Basic syntax, terms are compared in lexicographic manner.

A special form of aggregates is also allowed in the head of a rule. This type of aggregates, called head aggregates, are of form:

$$s_1 \ <_1 \ \alpha \ \{ \ t_1 : l_1 : L_1; \ \ldots; \ t_n : l_n : L_n \ \} \ <_2 \ s_2,$$

where s_1, s_2 are terms, $<_1, <_2$ are comparison predicate symols, α is an aggregate symbol, $t_1, \ldots, t_n$ are terms, $l_1, \ldots, l_n$ are literals and $L_1, \ldots, L_n$ are sets of literals.

When evaluating the set, term t_i is included in the set if both literal l_i and all literals from L_i are consistent with the solution. Additionaly, if the aggregate forms a head of rule r and body of r is consistent with the solution, the aggregate can introduce any number of literals l_i for such all literals L_i are consistent with the solution. This is especially useful for generating an input space of logic program.

For all $s_1, s_2, <_1, <_2, \alpha, L_i$ the default values and rules for omitting are still the same as when using the aggregates in the body. Specially, the expression

$$s_1 \ <_1 \ \texttt{\{} \ l_1\texttt{:}L_1\texttt{;} \ \texttt{...;} \ l_n\texttt{:}L_n \ \texttt{\}} \ <_2 \ s_2,$$

is equivalent to

$$s_1 \ <_1 \ \texttt{\#count\{} \ t_1\texttt{:}l_1\texttt{:}L_1\texttt{;} \ \texttt{...;} \ t_n\texttt{:}l_n\texttt{:}L_n \ \texttt{\}} \ <_2 \ s_2,$$

where all t_i are distinct.

Example 3.2.10. Let the logic program consist of a single fact

```
{a; b; c; d; e}.
```

This fact is a syntactic shortcut of fact[3]

```
#count{ 1:a:∅; 2:b:∅; 3:c:∅; 4:d:∅; 5:e:∅ }.
```

As an empty set of literals is consistent with any solution, any of atoms `a`, `b`, `c`, `d`, `e` can be included into the solution by this rule. As there is no constraint on the result of this aggregate, any subset of this set of atoms can be included into the solution.

The statement in the Example 3.2.10 allows us to easily define the input space of logic program.

Show statements

Often only a part of a solution is needed, while most of the solution can be hidden. For this, Clingo does support show statement. Such statement takes one of following forms:

3. The following expression is not a valid rule in clingo language as it does not contain ∅. I choose to include it into the notation here to improve readability.

```
#show p/n.
#show t:L_1, ..., L_n.
#show.
```

In the first form, atoms with predicate symbol p with exactly n parameters are shown on the output. If at least one such atom is included in the solution, only atoms specified by `#show` statements are shown. A predicate symbol and its classical negation is taken as two distinct symbols.

The second form allows for showing terms if some literals are consistent with the solution. For a term t and literals $L_1, \dots, L_n$, term t is shown on the output if all literals $L_1, \dots, L_n$ are consistent with the solution. This form does not hide any atoms from the solution.

Finally, the third form specifies, that no atom other than these specified by show statements in the first form should be shown on the output.

3.2.4 Gringo

Gringo [16] is software that grounds logic program in Clingo language and translates it into the Aspif format that is readable by logic program solver Clasp. Gringo first resolves every rule with variables into (possibly multiple) variable-free rules and then changes format of the program into Clasp-readable Aspif. Gringo thus can introduce new atoms that were not obvious from the Clingo program. Full specification of Aspif language is written in [18].

Aspif format

Aspif (ASP intermediate format) language consists of statements, each on its own line. First line of file is a header of form

```
asp v_m v_n v_r t_1 ... t_k
```

where v_m, v_n and v_r are versions of major, minor and revision numbers respectively and each t_i is a tag. Then follow lines with rules and statements translated from program. Last line of Aspif format file is a single 0.

For the translation from Clingo to Aspif language, Gringo introduces mapping M of literals onto positive numbers. Each literal is assigned a positive number, which identifies it in the whole Aspif program. To assert the constraint that no solution contains both an atom and its classical negation, for each such tuple of literals $\varphi, \neg\varphi$, Gringo introduces new rule to forbid such solutions:

$$\texttt{:-}\ \varphi\texttt{,}\ \neg\varphi\texttt{.}$$

In this thesis only rules and show statements are relevant.

Rule statements

Rule in Aspif has form of

$$\texttt{1}\ \ H\ \ B$$

in which head H has form of

$$h\ \ m\ \ a_1\ \ \ldots\ \ a_m$$

where $h \in \{0, 1\}$, $m \geq 0$, $\forall i \in \{1, \ldots, m\} . a_i \in \mathbb{N}^+$. Parameter h determines wherther head of this rule is disjunction (0) or choice (1), m determines number of literals and a_i are literals mapped to positive integers.

Body B is called normal if it has form of

$$\texttt{0}\ \ n\ \ l_1\ \ \ldots\ \ l_n$$

where n is the number of literals in statement and for each literal φ corresponding to $l_i, l_i = M(\varphi)$ if φ is positive in the sense of the default negation, otherwise $l_i = -M(\varphi)$. Literals of the normal body are in conjunction meaning all its positive literals and none of its negative literals have to be consistent with the answer set in order for the head of rule to be used.

The other type of body B is called weight body. Its form is

$$\texttt{1}\ \ b\ \ n\ \ l_1\ \ w_1\ \ \ldots\ \ l_n\ \ w_n$$

Parameter $b \geq 0$ determines lower bound, n the length of rule body, for each literal φ corresponding to l_i, $l_i = M(\varphi)$ if φ is positive in the sense of the default negation, otherwise $l_i = -M(\varphi)$ and $w_i \geq 1$ is the weight assigned to this literal. In weight body, the head of the rule is used if the sum of weights of expressions consistent with the answer set is greater or equal to the lower bound.

Example 3.2.11. Consider the following program in Clingo language:

```
1  p(a).
2  p(b) :- p(a).
3  p(c) :- p(a), p(b).

5  q(1..3).

7  r(1) :- not r(2).
8  r(2) :- not r(1).

10 s(true) :- q(1) : q(2), q(3).

12 t(A) :- A = #count { X: q(X) }.
```

When grounding the program, first a set of ground rules will be generated. In this case, non ground rules are on lines 5 and 12. Expansion of the rule on line 5 is going to be easy, it is going to expand into 3 facts, with values 1, 2 or 3 as the parameter.

```
5  q(1). q(2). q(3).
```

The rule on line number 12 will be expanded in two steps. First, the set of conditional terms will be expanded.

```
12 t(A) :- A = #count{ 1: q(1); 2: q(2); 3: q(3) }.
```

Now that both the minimal and maximal number of terms are known, the variable `A` can be grounded.

```
12 t(0) :- 0 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
13 t(1) :- 1 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
14 t(2) :- 2 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
15 t(3) :- 3 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
```

After the grounding process, the full program will have the form of:

```
1  p(a).
2  p(b) :- p(a).
3  p(c) :- p(a), p(b).

5  q(1). q(2). q(3).

7  r(1) :- not r(2).
8  r(2) :- not r(1).

10 s(true) :- q(1) : q(2), q(3).
```

```
12 t(0) :- 0 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
13 t(1) :- 1 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
14 t(2) :- 2 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
15 t(3) :- 3 = #count{ 1: q(1); 2: q(2); 3: q(3) }.
```

Finally, Gringo will process these grounded rules into statements[4]. On the first line of the resulting Aspif file informations about the version will be written.

```
1 1 0 0
```

Then, the fact and rules on lines 1–3 will be translated. Literals `p(a)`, `p(b)` and `p(c)` will be mapped to numbers `1`, `2` and `3`.

```
2 1 0 1 1 0 0
3 1 0 1 2 0 1 1
4 1 0 1 3 0 2 1 2
```

The first statement has a disjunctive head containing a single literal with assigned number `1`, that is the literal `p(a)`. It also has normal body containing no literal. As it does not contain any literal, the statement is a fact.
The second statement also has a disjunctive head containing a single literal `p(b)`. Its body is normal, containing a single literal `p(a)`, thus this rule ensures literal `p(b)` if `p(a)` is consistent.
The third statement also has a disjunctive head containing a single literal `p(c)`. Its body is normal, containing 2 literals `p(a)` and `p(b)`.

Line 5 contains only three facts. Each of them is going to be translated into a single rule, similiar to the fact from line 1. Literals `q(1)`, `q(2)` and `q(3)` will be assigned numbers `4`, `5`, `6`.

```
5 1 0 1 4 0 0
6 1 0 1 5 0 0
7 1 0 1 6 0 0
```

Rules on lines 7–8 contain default negation. According to the definition, negated values of numbers assigned to corresponding literals will be used in statement bodies. Literals `r(1)` and `r(2)` will be assigned numbers `7` and `8`.

4. In fact, Gringo will do some optimizations already in the time of translation into statements. For instance, it does see that literal `p(a)` has to always be included in the answer set, thus also the body of rule on line 2 does always hold and line 2 would be substitued for a fact. The mapping of literals to numbers is also not the same as Gringo would use.

```
1 0 1 7 0 1 -8
1 0 1 8 0 1 -7
```

The rule on line 10 does use conditional literal. To translate this into statements, Gringo uses trick similiar to the one used in the Example 3.2.6. For each conditional literal $H : B$ it first creates a rule with a new literal φ in the head and the body B as the body of the rule. Then it adds another literal ψ that corresponds to the value of the conditional literal and another two rules asserting that the literal ψ is consistent with solution if H is consistent or φ is not consistent with the solution.

The rule expands into the following rules in the Clingo language:

```
φ :- q(2), q(3).
ψ :- q(1).
ψ :- not φ.
s(true) :- ψ.
```

These rules are then directly translated into Aspif statements. Literals `q(1)`, `q(2)` and `q(3)` are already assigned numbers `4`, `5` and `6`. Literals φ, ψ and `s(true)` are assigned numbers `9`, `10` and `11`.

```
1 0 1 9 0 2 5 6
1 0 1 10 0 1 4
1 0 1 10 0 1 -9
1 0 1 11 0 1 10
```

The last set of rules do contain aggregates. These four rules only differ in the number of literals being true in the solution. To be transcribe these rules into statements in which only the lower bound is defined (where only the operator <= is allowed), Gringo splits each equality into two inequalities. Any expression in form of (A and B are head and rest of body):

```
A :- K = α{...}, B.
```

is first rewritten as three rules using only operator <=[5]:

```
φ :- K <= α{...}, B.
ψ :- K + 1 <= α{...}, B.
A :- φ, not ψ.
```

5. In fact, Gringo treats the lowest and highest values differently. As there is no way to undershoot or overshoot these values, it does not create the first or second rule respectively in these cases.

where φ and ψ are fresh literals. These expressions are equivalent as the value of an aggregate is always a whole number. The default negation in the last rule negates the `<=` operator of the second rule making its meaning close to `>`.

In our example, there are 4 such rules, thus we will be using 8 fresh literals. `t(0)`, `t(1)`, `t(2)` and `t(3)` are assigned numbers `12`, `13`, `14` and `15`. Fresh literals are assigned numbers `16`, `17`, `18`, `19`, `20`, `21`, `22`, `23`. `q(1)`, `q(2)` and `q(3)` are already assigned numbers `4`, `5` and `6`.

```
1 0 1 16 1 0 3 4 1 5 1 6 1
1 0 1 17 1 1 3 4 1 5 1 6 1
1 0 1 12 0 2 16 -17
1 0 1 18 1 1 3 4 1 5 1 6 1
1 0 1 19 2 1 3 4 1 5 1 6 1
1 0 1 13 0 2 18 -19
1 0 1 20 2 1 3 4 1 5 1 6 1
1 0 1 21 3 1 3 4 1 5 1 6 1
1 0 1 14 0 2 20 -21
1 0 1 22 3 1 3 4 1 5 1 6 1
1 0 1 23 4 1 3 4 1 5 1 6 1
1 0 1 15 0 2 22 -23
```

The last line of the Aspif program will be a single zero. Full program[6] will take form of:

```
1 0 0
1 0 1 1 0 0
1 0 1 2 0 1 1
1 0 1 3 0 2 1 2
1 0 1 4 0 0
1 0 1 5 0 0
1 0 1 6 0 0
1 0 1 7 0 1 -8
1 0 1 8 0 1 -7
1 0 1 9 0 2 5 6
1 0 1 10 0 1 4
1 0 1 10 0 1 -9
1 0 1 11 0 1 10
1 0 1 16 1 0 3 4 1 5 1 6 1
1 0 1 17 1 1 3 4 1 5 1 6 1
1 0 1 12 0 2 16 -17
1 0 1 18 1 1 3 4 1 5 1 6 1
```

6. As there are no show statements in the former logic program, the Aspif program would contain a show statement for each literal of the grounded program.

```
1 0 1 19 2 1 3 4 1 5 1 6 1
1 0 1 13 0 2 18 -19
1 0 1 20 2 1 3 4 1 5 1 6 1
1 0 1 21 3 1 3 4 1 5 1 6 1
1 0 1 14 0 2 20 -21
1 0 1 22 3 1 3 4 1 5 1 6 1
1 0 1 23 4 1 3 4 1 5 1 6 1
1 0 1 15 0 2 22 -23
0
```

It is very important to choose the right representation of a problem in answer set programming. As shown in the example above, some representations may lead to very large size of a grounded logic program. When using variables in the head of a rule, one has to be aware of the number of literals and rules added through the grounding process.

Show statements

Show statement is for specification of output, they result from `#show` directive. Each show statement is of form:

$$4 \;\; m \;\; s \;\; n \;\; l_1 \; \dots \; l_n$$

where m is length of string s, s is string with name, n is number length of condition and l_i are literals. The show statement prints the string s if all literals l_i are cosistent with the answer set.

Example 3.2.12. In the logic program from Example 3.2.11 in Clingo format, there are no show statements. Thus for every literal l with of the grounded logic program mapped to number $M(l)$, a single show statement of form:

$$4 \;\; ||L|| \;\; L \;\; 1 \;\; M(l)$$

would be added into the logic program in Aspif format.

To get the Aspif representation mentioned in this Example, a single show statement `#show.` could be used.

If show statement `#show q/1.` was added to the former logic program, the following set of show statements would be included in its Aspif representation:

```
4 4 q(1) 1 4
4 4 q(2) 1 5
4 4 q(3) 1 6
```

3.2.5 Clasp

Clasp is the solver of the Clingo framework. It takes logic program in the Aspif format to search for answer sets of a grounded logic program. The solver approaches the inference using the unit propagation of nogoods [20].

For the inference of answer sets of some logic program, Clasp uses backpropagation over a tree of partial solutions. Each time the it sees a partial solution that is not an answer set of the problem, it derives new constraints on the solution which it propagates up the tree of partial solutions. This allows Clasp to prune large branches of partial solutions.

For model (answer sets) counting, Clasp uses a method called model enumeration. To determine that a set is an answer set, it needs to evaluate the exact set. This means, that the time complexity of an enumeration is always dependent on the number of answer sets.

4 ASP encoding of BNN robustness

In this section I show possible encodings of binarised neural networks and of the robustness problem. Specifically, I create an encoding of a binarised neural network using negation as failure, input regions and adversarial inputs. Finally, I create a Python program able to encode weights and biases of a binarised neural network into a part of Clingo program.

4.1 Analysis of BNN

In the Definition 2.1.4 I have introduced a binarised perceptron $p^{\mathbb{B}}$ as a linear model composed from an inner potential ξ, parametrised by real bias b and vector of binarised values $\vec{w}$, and a heavyside step function H. Further I have shown that this model has the exact same expressive power as the binarised perceptron with batch normalization.

Then in the Section 3.2 I have shown the Clingo language. This language allows for computations over the set of whole numbers. It also allows for the use of aggregate expressions, which can be leveraged for summations.

The last large roadblock in the encoding of binarised perceptron into Clingo language is the real bias, as Clingo does only allow for the use of whole numbers.

Lemma 4.1.1. *For every binarised perceptron with bias from real numbers there is an equivalent binarised perceptron with bias from whole numbers and vice versa.*

Proof. Idea behind this construction comes from [1].

Let $p^{\mathbb{B}}$ be a binarised perceptron without batch normalisation.

$$p^{\mathbb{B}}(\vec{x}) = H \circ \xi(\vec{x}) = \begin{cases} 1 & b + \sum_{i=1}^{k} w_i \cdot x_i \geq 0 \\ -1 & b + \sum_{i=1}^{k} w_i \cdot x_i < 0 \end{cases}$$

Both each w_i and x_i are whole numbers (± 1-binarised values), thus also the sum $\sum_{i=1}^{k} w_i \cdot x_i$ is a whole number. As the formula for binarised perceptron only compares a sum of a real number and a whole number to a whole number, in both cases integer part of the bias $\lfloor b \rfloor$ can be used in place of the bias.

$$p^{\mathbb{B}}(\vec{x}) = H \circ \xi(\vec{x}) = \begin{cases} 1 & \lfloor b \rfloor + \sum_{i=1}^{k} w_i \cdot x_i \geq 0 \\ -1 & \lfloor b \rfloor + \sum_{i=1}^{k} w_i \cdot x_i < 0 \end{cases}$$

Such perceptron uses only whole numbers. Whole numbers are subset of real numbers, thus also the implication in the other direction is true. □

Remark. The proof of Lemma 4.1.1 gives a direct way to encode a binarised perceptron without batch normalisation into a binarised perceptron with bias from whole numbers. Using also the Lemma 2.1.2, a binarised perceptron with bias from whole numbers can be easily constructed even from binarised perceptron with batch normalisation.

Real bias is however included not only in perceptrons of inner layers, but also in the last Argmax layer. A transcription of this type of layer into whole numbers is needed.

Lemma 4.1.2. *Every Argmax layer with real-valued bias parameters can be transcribed using multiple whole numbers in place of bias parameters.*

Proof. The Argmax layer $t^{am} : \mathbb{B}^m \to \mathbb{B}^n$ consists of inner potential ξ and argmax encoding using the one-hot vector.

$$t^{am}(\vec{x}) = (y_1, \ldots, y_n)$$

$$y_k = \begin{cases} 1 & k = \arg\max_{i=1}^{n}(\xi_i(\vec{x})) \\ 0 & \text{otherwise} \end{cases}$$

$$\xi_i(\vec{x}) = b_i + \sum_{j=1}^{m} w_{i,j} \cdot x_j$$

The condition of an output position y_k having value 1 can be rewritten into another condition using the maximality of this position.

Position y_k on the output has value of 1 if and only if for every other output position y_l one of following holds:

- $\xi_k(\vec{x}) > \xi_l(\vec{x})$

- $\xi_k(\vec{x}) = \xi_l(\vec{x}) \land k < l$

By substitution for inner potential and splitting bias b into its integer part $\lfloor b \rfloor$ and fractional part $\{b\}$, the first condition splits. The condition can be true either if the integer part $\lfloor \xi_k(\vec{x}) \rfloor$ is bigger than the integer part $\lfloor \xi_l(\vec{x}) \rfloor$ or if they are equal and the fractional part $\{\xi_k(\vec{x})\} = \{b_k\}$ is bigger than $\{\xi_l(\vec{x})\} = \{b_l\}$:

- $\lfloor b_k \rfloor + \sum_{j=1}^{m} w_{k,j} \cdot x_j > \lfloor b_l \rfloor + \sum_{j=1}^{m} w_{l,j} \cdot x_j$
- $\lfloor b_k \rfloor + \sum_{j=1}^{m} w_{k,j} \cdot x_j = \lfloor b_l \rfloor + \sum_{j=1}^{m} w_{l,j} \cdot x_j \land \{b_k\} > \{b_l\}$
- $\lfloor b_k \rfloor + \sum_{j=1}^{m} w_{k,j} \cdot x_j = \lfloor b_l \rfloor + \sum_{j=1}^{m} w_{l,j} \cdot x_j \land \{b_k\} = \{b_l\} \land k < l$

Conditions in the last paragraph do provide for an encoding using whole numbers. Integer parts of biases b_k and b_l are already whole numbers. The output positions are enumerated, thus k and l are already whole numbers. Finally, on fractional parts $\{b_k\}$ and $\{b_l\}$ an ordering can be created. The Argmax layer can thus be transcribed using only whole numbers. □

Remark. Instead of using both the ordering of the fractional part of bias $\{b_k\}$ and the position k, an ordering that prioritizes high fractional part of bias and with lower priority low position k can be used. I will be using this ordering in the encoding of BNN.

4.2 ASP encoding of BNN

In the encoding of binarised neural network in Clingo language I use predicate symbols with meaning described in Table 4.1. This table contains atoms with their meaning categorised into ones that encode the binarised neural network, ones that specify the input region and ones that are computed while grounding. Both atoms that define the binarised neural network and that specify the input region are given to the logic program as facts by the Python encoder.

Symbol		Semantics
Encoding of BNN		
`layer`(L, N)	L	Layer number
		Input layer has number 0
		Output layer has the highest number
	N	Number of perceptrons in layer L
`weight`(L, M, N, W)	L	Layer
	M	Position of perceptron of layer $L - 1$
	N	Position of perceptron in layer L
	W	Weight of input M to perceptron N in layer L
`bias`(L, N, B)	L	Layer
	N	Position of perceptron in layer L
	B	Bias of perceptron N in layer L
`outpre`(N, P)	N	Position of output
	P	N has precedence P
		lower precedence mean higher priority
Encoding of input space		
`input`(N)		Specification of the base input
	N	Input on the position N of input layer has value 1
`hammdist`(R)		The input space is based on the hamming distance
	R	Maximal hamming distance from base input is R
`inpfix`(N)		The input space is based on fixed bits
	N	Position N of input is fixed to the value of base input
Computation		
`output_layer`(L)	L	Layer number L is the output layer
`potential`(L, N, P)	L	Layer number
	N	Position of perceptron in layer L
	P	Perceptron N of layer L has inner potential P
`on`(L, N)	L	Layer number
	N	Perceptron on position N of layer L outputs 1
`output`(N)	N	Output on position N of output layer has value 1

Table 4.1: Semantics of encoding BNN into Clingo-readable file

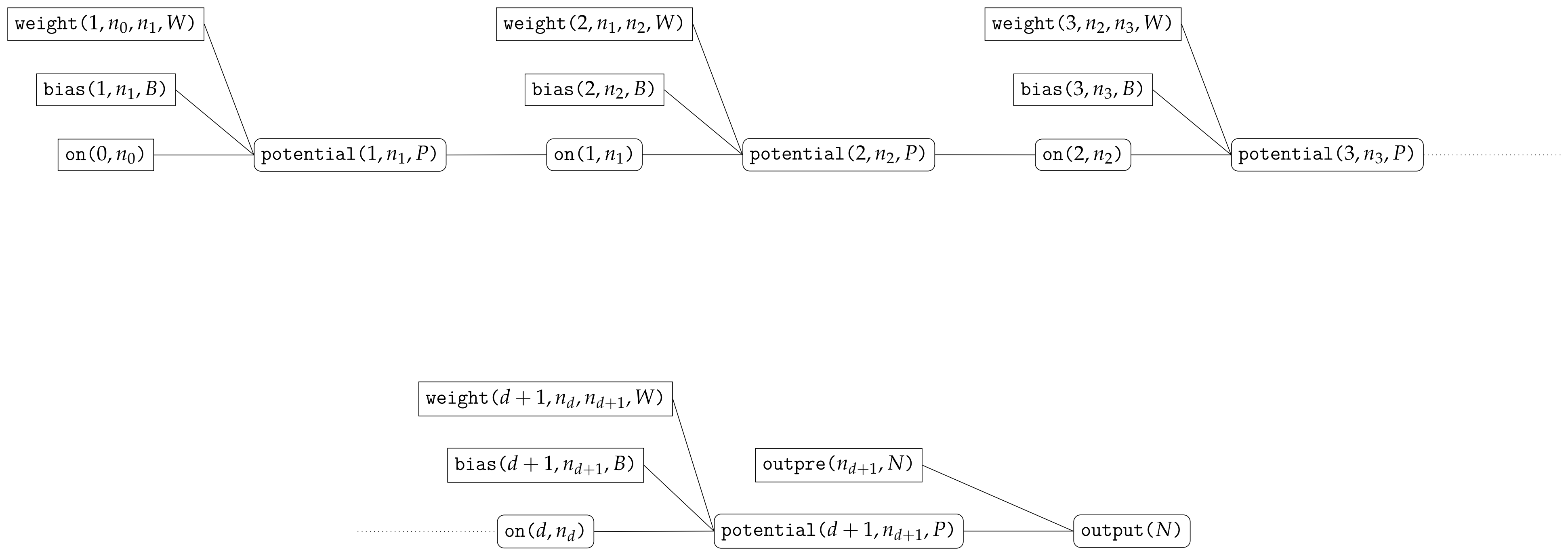

Figure 4.1: Schema of a multi-layer perceptron encoding

The encoding of BNN into Clingo language will be a composition of layers. Output of each layer is dependent only on the output values, weights and biases of the previous layer. Schema of inference for this architecture is shown in Figure 4.1. The schema contains two types of nodes, rectangular nodes do contain data that can be seen as an input of the inference and nodes with rounded corners that are computed in the runtime of the inference. As can be seen from the schema, the inference is linear, meaning the value of each layer is dependent only on the input data or data computed in the previous layer.

4.2.1 Encoding of a perceptron

As shown in Lemma 4.1.1, each binarised perceptron can be encoded using only bias with value from whole numbers. The encoding in Clingo language is straightforward.

```
potential(L, N, S+B) :-
    S = #sum{
         W,M :     on(L-1, M), weight(L, M, N, W);
        -W,M : not on(L-1, M), weight(L, M, N, W) },
    bias(L, N, B).

on(L,N) :- potential(L, N, P), P >= 0.
```

This implementation is a direct encoding of a binarised perceptron. It follows the formula of perceptron as described in the Lemma 4.1.1. The implementation is however weak as it relies on the use of intermediate symbol `potential`. While grounding, this would require to build a literal for every possible value of inner potential of each perceptron in the network and consequently in a large ground program as shown in the Example 3.2.11. To fight that, literals of symbol `on` may be built directly.

```
on(L, N) :-
    -B <= #sum{
         W,M :     on(L-1, M), weight(L, M, N, W);
        -W,M : not on(L-1, M), weight(L, M, N, W) },
    bias(L, N, B).
```

The last implementation still has a single flaw. It uses both the positive and negative variant of the literal `on(L-1, M)` in sense of the default negation. We can however make a transformation of the BNN such that it eliminates use of the negative variant.

Starting with the expression from 4.1.1, the ±1-binarised input vector $\vec{x}$ can be substituted by a $\{1,0\}$-binarised input vector $\vec{x}_b$:

$$x_{b,i} = \begin{cases} 1 & x_i = 1 \\ 0 & x_i = -1 \end{cases}$$

$$x_i = 2 \cdot x_{b,i} - 1$$

$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot (2 \cdot x_{b,i} - 1)$$

This expression can be further transformed by splitting the sum and eliminating the multiplication by 2.

$$\xi(\vec{x}) = b + \sum_{i=1}^{k} w_i \cdot 2 \cdot x_{b,i} - \sum_{i=1}^{k} w_i \cdot 1$$

$$\xi(\vec{x}) = b - \sum_{i=1}^{k} w_i + 2\sum_{i=1}^{k} w_i \cdot x_{b,i}$$

$$p^{\mathbb{B}}(\vec{x}) = H \circ \xi(\vec{x}) = \begin{cases} 1 & b - \sum_{i=1}^{k} w_i + 2\sum_{i=1}^{k} w_i \cdot x_{b,i} \geq 0 \\ -1 & b - \sum_{i=1}^{k} w_i + 2\sum_{i=1}^{k} w_i \cdot x_{b,i} < 0 \end{cases}$$

Finally, both cases can be divided by 2. The expression $\frac{b-\sum_{i=1}^{k} w_i}{2}$ in the final form of equality is independent of the input vector. It can be seen as the new bias, thus integer part of this expression can take its place similiarly to Lemma 4.1.1.

$$p^{\mathbb{B}}(\vec{x}) = H \circ \xi(\vec{x}) = \begin{cases} 1 & \left\lfloor \frac{b-\sum_{i=1}^{k} w_i}{2} \right\rfloor + \sum_{i=1}^{k} w_i \cdot x_{b,i} \geq 0 \\ \\ -1 & \left\lfloor \frac{b-\sum_{i=1}^{k} w_i}{2} \right\rfloor + \sum_{i=1}^{k} w_i \cdot x_{b,i} < 0 \end{cases}$$

With weights and biases adjusted in the way described above, the perceptron can be encoded without the use of default negation:

```
on(L, N) :-
    -B <= #sum{ W,M : on(L-1, M), weight(L, M, N, W) },
    bias(L, N, B).
```

4.2.2 Encoding of an Argmax layer

As shown in Lemma 4.1.2 and remark that follows it, Argmax layer can be encoded using the vector of weights, biases and the precedence of individual outputs. First, literals of predicate symbol `potential` can be constructed similiarly to the perceptron, on these `max` aggregate can be used.

```
potential(D+1, N, S+B) :-
    S = #sum{
         W,M :     on(D, M), weight(D, M, N, W);
        -W,M : not on(D, M), weight(D, M, N, W) },
    bias(D, N, B), output_layer(D+1).

output(N) :-
    (Sum, Precedence, Node) = #max{
        (S, -P, N) : potential(D+1, N, S), outpre(N, P)
   },
    output_layer(D+1).
```

While this may work, in the grounding process it will expand into a large ruleset. The ruleset will contain a rule with `potential` in head for every possible value of sum of inputs and bias, for each of these multiple rules to implement the `max` aggregate will be created.

The `potential` can however not be directly substituted for a comparison in the body of `output` like in the implementation of perceptron as the value of inner potential is needed for the comparison of different output values.

To overcome this problem, it can be observed from the other side. Instead of asking wether the particular output position has the largest inner potential, all output position with inner potential less than some other can be forbidden from being the final output. An `output` literal can then be introduced into the solution using a rule with count aggregate in its head.

```
potential(D+1, N, S+B) :-
    S = #sum{
         W,M :     on(D, M), weight(D+1, M, N, W);
        -W,M : not on(D, M), weight(D+1, M, N, W) },
    bias(D+1, N, B), output_layer(D+1).

1 { output(1..N) } 1 :- output_layer(D+1), layer(D+1, N).

:- output(N),
    potential(D+1, N, S), potential(D+1, M, T),
    output_layer(D+1),
    N != M, S < T.
:- output(N),
    potential(D+1, N, S), potential(D+1, M, T),
    outpre(N, P), outpre(M, Q),
    output_layer(D+1),
    N != M, S = T, P > Q.
```

Negated literals with predicate symbol `on` can be eliminated similiarly to the elimination in the encoding of perceptron. The precedence must be however calculated from the new bias after elimination.

```
potential(D+1, N, S+B) :-
    S = #sum{ W,M : on(D, M), weight(D+1, M, N, W) },
    bias(D+1, N, B), output_layer(D+1).

1 { output(1..N) } 1 :- output_layer(D+1), layer(D+1, N).

:- output(N),
    potential(D+1, N, S), potential(D+1, M, T),
    output_layer(D+1),
    N != M, S < T.
:- output(N),
    potential(D+1, N, S), potential(D+1, M, T),
    outpre(N, P), outpre(M, Q),
    output_layer(D+1),
    N != M, S = T, P > Q.
```

Literals with predicate symbol `potential` can now be also eliminated by using `sum` aggregate directly in the comparison of inner potential values.

```
1 { output(1..N) } 1 :- output_layer(D+1), layer(D+1, N).

:- output(N),
    S = #sum{ W,O : on(D, O), weight(D+1, O, N, W) },
    T = #sum{ W,O : on(D, O), weight(D+1, O, M, W) },
    bias(D+1, N, B), bias(D+1, M, C),
    output_layer(D+1),
    N != M, S + B < T + C.
:- output(N),
    S = #sum{ W,O : on(D, O), weight(D+1, O, N, W) },
    T = #sum{ W,O : on(D, O), weight(D+1, O, M, W) },
    bias(D+1, N, B), bias(D+1, M, C),
    outpre(N, P), outpre(M, Q),
    output_layer(D+1),
    N != M, S + B = T + C, P > Q.
```

While grounding, the value of an aggregate may be only compared to an integer value. It may not be compared with value of another aggregate directrly. This results in a grounding of this logic program being more or less the same as of the last logic program using potential (in fact, the grounded program is even bigger than before). To avert this, both aggregates have to be merged into a single one.

$$S + B < T + C$$

$$S - T < C - B$$

The expression is easily transformed for variables representing values of aggregates to be on one side. As both aggregates are `sum` aggregates, values of aggregate with variable `T` may be multiplied by -1. As the computation of aggregate expression uses representation as sets, additional value must be added to tuples for summation to distinguish between values of weights for node `N` and `M`. The same transformation may be done also with the second constraint of the logic program.

```
1 { output(1..N) } 1 :- output_layer(D+1), layer(D+1, N).

:- output(N),
    C - B < #sum{  W,O, 1 : on(D, O), weight(D+1, O, N, W
   ) ;
                  -W,O,-1 : on(D, O), weight(D+1, O, M, W
   ) },
    bias(D+1, N, B), bias(D+1, M, C),
    output_layer(D+1),
    N != M.
:- output(N),
    C - B = #sum{  W,O, 1 : on(D, O), weight(D+1, O, N, W
   ) ;
                  -W,O,-1 : on(D, O), weight(D+1, O, M, W
   ) },
    bias(D+1, N, B), bias(D+1, M, C),
    outpre(N, P), outpre(M, Q),
    output_layer(D+1),
    N != M, P > Q.
```

Finally, as described in Example 3.2.11, when an equality symbol is used in an aggregate, while grounding it is transformed into two inequalities. To further simplify (and shorten) the grounded logic program, symbol <= may be used instead of = in the second constraint. If $C - B < S - T$, then the output is not the right one by the first constraint already, thus it is equivalent to the previous logic program.

```
1 { output(1..N) } 1 :- output_layer(D+1), layer(D+1, N).

:- output(N),
    C - B < #sum{  W,O, 1 : on(D, O), weight(D+1, O, N, W
   ) ;
                  -W,O,-1 : on(D, O), weight(D+1, O, M, W
   ) },
    bias(D+1, N, B), bias(D+1, M, C),
    output_layer(D+1),
    N != M.
:- output(N),
    C - B <= #sum{  W,O, 1 : on(D, O), weight(D+1, O, N,
   W) ;
                   -W,O,-1 : on(D, O), weight(D+1, O, M,
   W) },
    bias(D+1, N, B), bias(D+1, M, C),
    outpre(N, P), outpre(M, Q),
    output_layer(D+1),
    N != M, P > Q.
```

4.2.3 Encoding of input regions

Similiarly to the final encoding of the argrmax layer, I will encode input region by first allowing for any input vector using an head aggregate with unspecified boundaries over literals `on(0, 1..N)`, on top of this I will then build constraints defined by the desired input region.

```
{ on(0, 1..N) } :- layer(0, N).
```

Input region based on the Hamming distance

To constrain input such that only inputs with hamming distance at most r are allowed, a simple aggregate rule can be added into the set of rules. The rule denies any partial solution with input that contains more than r possitions differing from the base input to be an answer set.

```
:- #count{ N :      on(0, N), not input(N);
```

```
           N : not on(0, N),      input(N) } > R,
    hammdist(R).
```

Input region based on the fixed bits

Constraining the input on some fixed bits is trivial. To make such constraint, for each fixed position, one can forbid either the base input position having value 1 and real input position not having 1 and vice versa.

```
:-     on(0, N), not input(N), inpfix(N).
:- not on(0, N),     input(N), inpfix(N).
```

The second constraint can be substitued for a rule with `on(0,N)` in its head and not in its body.

```
:- on(0, N), not input(N), inpfix(N).
on(0, N) :- input(N), inpfix(N).
```

Finally, instead of allowing for any input to have either value at the start, one can encode the possibility for another value only for the input positions that are not fixed. This may allow for further optimizations in the grounding process.

```
on(0, N) :- input(N), inpfix(N).
{ on(0, K) } :- not inpfix(K), layer(0, N), K = 1..N.
```

4.2.4 Encoding of robustness

As already discussed in section 3.2.5, for the inference on a logic program, Clasp uses model enumeration. This means that it needs to enter every leaf node of a tree of partial solutions that is a model (answer set) of this logic program. On the other hand, if part of this tree does not contain any model, it may prune this whole branch.

The robustness as defined in the Definition 2.2.1 does not depend on inputs that yield output values for which the evaluation function h_p does yield 0. By the previous observation it is good to remove all such inputs from the models of the logic program.

Clingo itself can not directly compute the quantitative robustness of the binarised neural network. It can however still count the number of its models. By forbidding the desired output value in the logic program, only the partial solution that do not output this desired value (thus break the robustness) are left as models of the logic program. When using the evaluation function h that assigns 0 to forbidden output values and 1 to all other values and weight function $w(x) = 1$, the quantitative robustness can be easily computed from the number of found models and the size of the input region. The size of the input region can in turn be found out using Lemmas 2.2.6 and 2.2.7.

$$Q(I) = \frac{\sum_{i \in I} w(i) \cdot \overline{h_p}(F(i))}{\sum_{i \in I} w(i)} = \frac{\sum_{i \in I} 1 \cdot \overline{h_p}(F(i))}{\sum_{i \in I} 1}$$

As only the models of such logic program yield $\overline{h_p}(F(i)) = 1$, the upper part can be substitued for the number of models. The lower part corresponds to the size of the input region.

$$Q(I) = \frac{\#models}{||input\_region||}$$

Forbidden outputs can be specified using constraints on literals with predicate symbol `output`. In case of an evaluation function $\overline{h_p}$ from Definition 2.2.3 that assigns 0 to output value k, the constraint would be:

```
:- output(k).
```

as the value k does lead to the desired output and is thus forbidden. On the other hand, to encode t-target robustness (see end of Section 2.2.1) following constraint could be used:

```
:- not output(t).
```

as only the output value t is not desired.

4.3 Encoding robustness of BNN into logic program

For transcription of the binarised neural network and the robustness problem, I use Python script. This script takes on input a binarised neural network in form of a directory of csv files, a base input in form of text file and possibly multiple other parameters (see Section 4.3.3). It then encodes this model into logic program using specified parameters. Finally, it executes the logic program using Clingo, counting all models with output that differs from the desired one.

4.3.1 Structure of saved BNN

The script allows for evaluation of binarised neural networks in the form of binarised multi-layer perceptron with batch normalization.

Binarised neural network is represented as a directory containing a subdirectory named `blk`X for every inner layer X of the network and a subdirectory named `out_blk` for the output Argmax layer. Each inner block subdirectory then contain files `bn_bias.csv`, `bn_mean.csv`, `bn_var.csv`, `bn_weight.csv`, `lin_bias.csv` and `lin_weight.csv`. Each of these files contain vector or matrix corresponding to the parameter of the layer. The last Argmax layer subdirectory contains only files `lin_bias.csv` and `lin_weight.csv`.

Further follows specification of individual files.

`bn_bias.csv`

File contains a vector specified using a single line of comma separated values. This vector corresponds to the biases $\vec{\gamma}$ of a binarised single-layer perceptron with batch normalization, i-th entry corresponds to the bias of perceptron on i-th position of this layer.

`bn_mean.csv`

File contains a vector specified using a single line of comma separated values. This vector corresponds to the means $\vec{\mu}$ of a binarised single-layer perceptron with batch normalization, i-th entry corresponds to the mean of perceptron on i-th position of this layer.

`bn_var.csv`

File contains a vector specified using a single line of comma separated values. This vector corresponds to the variances $\vec{\sigma}^2$ of a binarised single-layer perceptron with batch normalization. To get the standard deviations $\vec{\sigma}$, piecewise square root must be applied to this vector. i-th entry corresponds to the variance of perceptron on i-th position of this layer.

`bn_weight.csv`

File contains a vector specified using a single line of comma separated values. This vector corresponds to the weights $\vec{\alpha}$ of a binarised single-layer perceptron with batch normalization, i-th entry corresponds to the weight of perceptron on i-th position of this layer.

`lin_bias.csv`

File contains a vector specified using a single line of comma separated values. This vector corresponds to the biases $\vec{b}$ of a binarised single-layer perceptron with batch normalization, i-th entry corresponds to bias the perceptron on i-th position of this layer.

`lin_weight.csv`

File contains a matrix specified using a table of values with comma as separator of columns. This matrix corresponds to the weights $\mathbf{w}$ of a binarised single-layer perceptron with batch normalization. Values of this matrix are ± 1-binarised. Vector constituted by the i-th row of the matrix $\mathbf{w}$ corresponds to the vector of weights from previous layer to the perceptron on i-th position of this layer. Entry on i-th row and j-th column corresponds to the weight from j-th position of the previous layer to i-th position of this layer.

4.3.2 Transformation of BNN

After loading, the binarised neural network with batch normalization (Definitions 2.1.5 and 2.1.7) is transformed into the whole number-valued form using Python package Numpy [21].

Further I show the whole transformation with representation using matrices and vectors contrary to vectors and scalar values that were used so far. When using vectors I assume they represented by a column of values. In the implementation, they are transposed when loading from files. To distinguish between them, I will use $\cdot$ for dot product, $\star$ for pointwise product and $\div$ for pointwise division.

Transformation of inner layer

Starting with binarised single-layer with batch normalization Definition 2.1.5:

$$t^{\mathbb{B}}(\vec{x}) \equiv \vec{\alpha} \star \left(((\vec{b} + \mathbf{w} \cdot \vec{x}) - \vec{\mu}) \div \vec{\sigma} \right) + \vec{\gamma} \geq 0$$

Using Lemma 2.1.2, the expression can be altered to use only a single bias and a matrix of weights.

$$b'_i = \begin{cases} b_i - \mu_i + \frac{\sigma_i}{\alpha_i} \cdot \gamma_i & \frac{\alpha_i}{\sigma_i} > 0 \\ \gamma_i & \frac{\alpha_i}{\sigma_i} = 0 \\ -b_i + \mu_i - \frac{\sigma_i}{\alpha_i} \cdot \gamma_i & \frac{\alpha_i}{\sigma_i} < 0 \end{cases}$$

$$\vec{w_i}' = \begin{cases} \vec{w_i} & \frac{\alpha_i}{\sigma_i} > 0 \\ \vec{0} & \frac{\alpha_i}{\sigma_i} = 0 \\ -\vec{w_i} & \frac{\alpha_i}{\sigma_i} < 0 \end{cases}$$

$$t^{\mathbb{B}}(\vec{x}) \equiv \vec{b}' + \mathbf{w}' \cdot \vec{x} \geq 0$$

In the encoding of weight, there is now a possibility for some rows to be vectors of zeros. That is not a problem as they can still be encoded into the Clingo language. In aggregate expressions any whole numbers may be used.

To further simplify the encoding into logic program, the ± 1-binarised input vector $\vec{x}$ may be mapped to vector $\vec{x}_b$ of values $\{1, 0\}$ (Section 4.2.1).

$$t^{\mathbb{B}}(\vec{x}) \equiv \vec{b}' + \mathbf{w}' \cdot (2\vec{x}_b - \vec{1}) \geq 0$$

$$\vec{b}'' = \frac{\vec{b}' - \mathbf{w}' \cdot \vec{1}}{2}$$

$$t^{\mathbb{B}}(\vec{x}) \equiv \vec{b}'' + \mathbf{w}' \cdot \vec{x}_b \geq 0$$

Finally to prepare the perceptron for the transcription to logic program, pointwise flooring function may be used on the bias $\vec{b}''$. If the use of $\{1, 0\}$-binarised perceptron is not desired, $\vec{b}'$ and $\vec{x}$ can be used in place of $\vec{b}''$ and $\vec{x}_b$ in this step.

$$t^{\mathbb{B}}(\vec{x}) \equiv \left\lfloor \vec{b}'' \right\rfloor + \mathbf{w}' \cdot \vec{x}_b \geq 0$$

Transformation of Argmax layer

The transformation of Argmax layer is arguably simpler than the one of inner layers. It starts directly with inner potential and outputs vector with 1 only at the position of largest value.

$$t^{am}(\vec{x}) = \arg\max(\vec{b} + \mathbf{w} \cdot \vec{x})$$

Again as shown in Sections 4.2.1 and 4.2.2, the input vector $\vec{x}$ may be mapped to vector $\vec{x}_b$ of values $\{1, 0\}$ to simplify the encoding of logic program.

$$t^{am}(\vec{x}) = \arg\max(\vec{b} + \mathbf{w} \cdot (2\vec{x}_b - \vec{1}))$$

$$\vec{b}' = \frac{\vec{b} - \mathbf{w} \cdot \vec{1}}{2}$$

$$t^{am}(\vec{x}) = \arg\max(\vec{b}' + \mathbf{w} \cdot \vec{x}_b)$$

As shown in Lemma 4.1.2, the bias may be split to its integer part and fractional part and ordering made on fractional part. This ordering should have the position with highest fractional part as highest priority. The algorithm for the sorting needs to be stable as in the event of the fractional part being the same value, lower position has higher priority.

$$\vec{b}' = \left\lfloor \vec{b}' \right\rfloor + \{\vec{b}'\}$$

$$ord = \arg\operatorname{sort}(-\{\vec{b}'\})$$

4.3.3 Parameters of the Evaluator

5 Evaluation

6 Conclusion

Appendices

Code of BNN verificator

The full code of the verificator implemented in this thesis together with the examples can be found either in the Thesis archive in the IS MU or in the Github repository https://github.com/Ardnij123/BNN_verification

Bibliography

1. ZHANG, Yedi; ZHAO, Zhe; CHEN, Guangke; SONG, Fu; CHEN, Taolue. BDD4BNN: a BDD-based quantitative analysis framework for binarized neural networks. In: *International Conference on Computer Aided Verification*. Springer, 2021, pp. 175–200.
2. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; SCHAUB, Torsten. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*. 2019, vol. 19, no. 1, pp. 27–82.
3. HUBARA, Itay; COURBARIAUX, Matthieu; SOUDRY, Daniel; EL-YANIV, Ran; BENGIO, Yoshua. Binarized Neural Networks. *ArXiv*. 2016, vol. abs/1602.02505.
4. BISHOP, Christopher M. Neural networks for pattern recognition. In: 1995. Available also from: `https://api.semanticscholar.org/CorpusID:60563397`.
5. HONG, Don; WANG, Jianzhong; GARDNER, Robert. *Real Analysis with an Introduction to Wavelets and Applications*. Burlington: Academic Press, 2005. ISBN 978-0-12-354861-0. Available from DOI: `https://doi.org/10.1016/B978-012354861-0/50003-8`.
6. ZHANG, Yedi; ZHAO, Zhe; CHEN, Guangke; SONG, Fu; CHEN, Taolue. Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach. *ACM Trans. Softw. Eng. Methodol.* 2023, vol. 32, no. 3. ISSN 1049-331X. Available from DOI: `10.1145/3563212`.
7. LIFSCHITZ, V. What is answer set programming? In: *Proc. 23rd AAAI Conf. on Artificial Intelligence, 2008*. 2008, pp. 1594–1597.
8. ANGER, Christian; KONCZAK, Kathrin; LINKE, Thomas; SCHAUB, Torsten. A Glimpse of Answer Set Programming. *Künstliche Intell.* 2005, vol. 19, no. 1, p. 12.
9. DELGRANDE, Jim. *Answer Set Programming* [online]. [N.d.]. [visited on 2024-04-27]. Available from: `https://www2.cs.sfu.ca/CourseCentral/721/jim/ASP1.Intro.pdf`.

10. GELFOND, Michael; LIFSCHITZ, Vladimir. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*. 1991, vol. 9, pp. 365–385. Available from DOI: `10.1007/BF03037169`.

11. HARMELEN, Frank; LIFSCHITZ, Vladimir; PORTER, Bruce. The Handbook of Knowledge Representation. *Elsevier Science San Diego, USA*. 2007, p. 1034.

12. ZARBA, Calogero G. *Many-Sorted Logic* [online]. 2006. [visited on 2024-10-26]. Available from: `https://web.archive.org/web/20070929131504/http://react.cs.uni-sb.de/~zarba/snow/ch01.pdf`.

13. EITER, Thomas; GOTTLOB, Georg; MANNILA, Heikki. Disjunctive datalog. *ACM Trans. Database Syst.* 1997, vol. 22, no. 3, pp. 364–418. ISSN 0362-5915. Available from DOI: `10.1145/261124.261126`.

14. SCHLIPF, J.S. The Expressive Powers of the Logic Programming Semantics. *Journal of Computer and System Sciences*. 1995, vol. 51, no. 1, pp. 64–86. ISSN 0022-0000. Available from DOI: `https://doi.org/10.1006/jcss.1995.1053`.

15. LIFSCHITZ, Vladimir. Twelve Definitions of a Stable Model. In: GARCIA DE LA BANDA, Maria; PONTELLI, Enrico (eds.). *Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 37–51. ISBN 978-3-540-89982-2.

16. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; SCHAUB, Torsten. Multi-shot ASP solving with clingo. *CoRR*. 2017, vol. abs/1705.09811.

17. GEBSER, MARTIN; HARRISON, AMELIA; KAMINSKI, ROLAND; LIFSCHITZ, VLADIMIR; SCHAUB, TORSTEN. Abstract gringo. *Theory and Practice of Logic Programming*. 2015, vol. 15, no. 4–5, pp. 449–463. ISSN 1475-3081. Available from DOI: `10.1017/s1471068415000150`.

18. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; OSTROWSKI, Max; SCHAUB, Torsten; WANKO, Philipp. *Theory Solving Made Easy with Clingo 5 (Extended Version)*. 2016.

19. GEBSER, Martin; KAMINSKI, Roland; KAUFMANN, Benjamin; LINDAUER, Marius; OSTROWSKI, Max; ROMERO, Javier; SCHAUB, Torsten; THIELE, S; WANKO, P. *Potassco guide version 2.2. 0*. 2019.

20. GEBSER, Martin; KAUFMANN, Benjamin; SCHAUB, Torsten. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 2012, vol. 187, pp. 52–89.

21. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON, Pearu; GÉRARD-MARCHANT, Pierre; SHEPPARD, Kevin; REDDY, Tyler; WECKESSER, Warren; ABBASI, Hameer; GOHLKE, Christoph; OLIPHANT, Travis E. Array programming with NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: 10.1038/s41586-020-2649-2.