FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

OF THE COMENIUS UNIVERSITY IN BRATISLAVA

OPTIMIZATION OF AN ABDUCTIVE REASONER FOR DESCRIPTION LOGICS

Master thesis

2019 **Katarína Fabianová**

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

OF THE COMENIUS UNIVERSITY IN BRATISLAVA

OPTIMIZATION OF AN ABDUCTIVE REASONER FOR DESCRIPTION LOGICS

Master thesis

Study program:	Applied informatics
Field of study:	2511 Applied informatics
School department:	Department of Applied Informatics
Adviser:	Mgr. Júlia Pukancová, PhD.
Consultant:	RNDr. Martin Homola, PhD.

Bratislava 2019 **Katarína Fabianová**

27565914

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Katarína Fabianová
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	aplikovaná informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Optimization of an abductive reasoner for description logics
Optimalizácia abduktívneho usudzovania nad deskripčnými logikami

Anotácia: Abduktívne usudzovanie je stále nová a neštandardná inferenčná technika v deskripčných logikách. V súčasnosti sú známe viaceré návrhy a implementácie abduktívnych inferenčných strojov, takéto prístupy sú ale založené na prehľadávaní obrovského priestoru hypotetických vysvetlení. Toto otvára priestor pre využitie heuristík, či iných optimalizačných metód, za účelom zvýšenia ich efektivity.

Cieľ: Navrhnúť a implementovať optimalizačné techniky pre existujúci abduktívny inferenčný nástroj pracujúci nad deskripčnou logikou s cieľom zlepšiť efektivitu výpočtu.

Literatúra:
1. Elsenbroich, C., Kutz, O. and Sattler, U., 2006. A case for abductive reasoning over ontologies. In OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA. Vol. 216 of CEUR-WS, 2006.
2. Pukancová, J. and Homola, M., Tableau-based ABox abduction for the ALCHO description logic. In: 30th International Workshop on Description Logics Montpellier, France. Vol. 1879 of CEUR-WS, 2017.
3. Halland, K. and Britz, K. ABox abduction in ALC using a DL tableau. In: South African Institute for Computer Scientists and Information Technologists Conference, Pretoria, South Africa. ACM, 2012.
4. Reiter, R. A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57-95, 1987.

Vedúci:	Mgr. Júlia Pukancová, PhD.
Konzultant:	doc. RNDr. Martin Homola, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	13.10.2017

Dátum schválenia: 13.10.2017 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.... študent

.... vedúci práce

Declaration of honour

I hereby declare that I wrote this master thesis by myself only with the help of the referenced literature and with the help of my adviser and consultant.

Bratislava Katarína Fabianová

Acknowledgements

I would like to thank to my adviser Mgr. Júlia Pukancová, PhD. and to my consultant RNDr. Martin Homola, PhD. for their amazing help, valuable advices and for their guidance that helped me write this thesis.

Abstract

Logical abduction is a form of logical reasoning where we don't know the possible cause but we know theory and effect. In this thesis we propose two ways of solving ABox Abduction problem. The first way uses an existing algorithm, the Minimal Hitting Set algorithm. This algorithm is sound and complete but it has an exponential time complexity. We propose the MergeXPlain algorithm as the second way. This algorithm already exists but it was not used to solve this problem and we want to adjust it to our problem and find out if this algorithm is the solution we look for. The advantage of this algorithm is its time complexity. We want to implement both approaches and compare their execution times and results.

Key words: Abduction, Description logics, Minimal Hitting Set, MergeXPlain

Abstrakt

Logická abdukcia je typ logického odvodzovania, kde nepoznáme možnú príčinu, ale poznáme teóriu a dôsledok. V tejto práci navrhujeme dve riešenia pre problém ABox abdukcie. Prvý riešenie je použitie už existujúceho algoritmu, Minimal Hitting Set algoritmus. Tento algoritmus je zdravý a úplný, ale má exponencionálnu zložitosť. Preto navrhujeme druhé riešenie, ktorým je MergeXPlain algoritmus. Tento algoritmus tiež už existuje, ale ešte nebol použitý na náš problém a chceme ho upraviť pre riešenie nášho problému. Chceme zistiť, či tento algoritmus je riešenie, ktoré hľadáme. Výhodou tohto algoritmu je jeho časová zložitosť. Chceme implementovať oba spôsoby a porovnať ich výsledne časy behu a ich výsledky.

Kľúčové slová: Abdukcia, Deskripčné logiky, Minimal Hittting Set, MergeXPlain

Contents

List of Figures

Introduction

Description logics are a family of knowledge representation and we use their language in our problem. Our problem is ABox abduction where our input is ontology and observation. They both use the language of description logics.

Logical abduction is a form of logical reasoning depending of knowing input theory and effect. Abduction knows theory and effect and is looking for the cause. In our work we want to optimize ABox abduction task. ABox abduction task is using ABox assertion as an observation which is effect in general abduction. We need to have a given theory which in ABox abduction is called an ontology. Ontology is our knowledge base where our input data are described in a form of description logics.

We want to optimize ABox abduction problem. Until now there was only Reiter's Minimal Hitting Set algorithm and its variations and optimizations. We want to compare our implementation of this algorithm with another algorithm that we propose. The other algorithm is the MergeXPlain algorithm from Shchekotykhin. This algorithm already exists but it was not used to compute minimal explanations for ABox abduction problem. We want to adjust this algorithm to match the solution of our problem. We want to compare the results from both implementations and discover if the MergeXPlain algorithm is suitable as solution for this problem. We assume that if the MergeXPlain is able to solve this problem we could significantly eliminate execution time which takes to solve this problem with Minimal Hitting Set algorithm. The Minimal Hitting Set algorithm is sound and complete, so it will find all minimal explanations but it has an exponentional time complexity which makes this algorithm unusable for bigger inputs. On the other hand the MergeXPlain algorithm should be much more quicker but we do not know if the MergeXPlain algorithm is sound and complete yet.

At first we will introduce description logics and abduction. Then we will present the theory of the Minimal Hitting Set algorithm and the MergeXPlain algorithm. After theory we will describe our approach with theses two algorithms and then we evaluate results from experiments.

1 Description Logics

Description logics (DLs) are a family of knowledge representation formalism. Each description logic has different expressivity. Every expressivity is expressed with a unique set of constructors. We will work with $\mathcal{ALC}$, $\mathcal{EL}$ and $\mathcal{EL}$++ DL. Each description logic has its own syntax and semantics. In this chapter we will introduce syntax and semantics (Rudolph, 2011) for each DL that we will be working with.

1.1 $\mathcal{ALC}$ DL

$\mathcal{ALC}$ DL is a DL which is more expressive than $\mathcal{EL}$ and $\mathcal{EL}$++ DL. $\mathcal{ALC}$ is less expressive than many other DLs. $\mathcal{ALC}$ stands for Attributive (Concept) Language with Complements. It means that not only complement of atomic concept is allowed but also a complement of complex concept is allowed.

1.1.1 $\mathcal{ALC}$ Syntax

$\mathcal{ALC}$ description logic consists of three mutually disjoint sets. These sets (Definition 1.1.1) represent whole vocabulary that is used by $\mathcal{ALC}$ DL.

Definition 1.1.1 ***DL vocabulary***

DL vocabulary contains all symbols we use in concepts and knowledge base. It consists of three mutually disjoint sets: N_I, N_C and N_R.

Set of individuals: $N_I = \{a, b, c...\}$

Set of concepts: $N_C = \{A, B, C...\}$

Set of roles: $N_R = \{R_1, R_2, R_3, ...\}$

Example 1.1.1 ***DL vocabulary***

$$N_I = \{jack, john, jane\}$$

$$N_C = \{Person, Mother, Father\}$$

$$N_R = \{hasChild, likes, owns\}$$

$\mathcal{ALC}$ DL deals with individuals and concepts. An individual is a concrete instance of a concept. Concept is a class that defines some entity. Concept can be atomic or complex. Atomic concept is not constructed with any constructor. On the contrary complex concept is created from constructors and other concepts.

Definition 1.1.2 ***Complex concept***

Concepts are recursively constructed as the smallest set of expressions of the forms:

$$C, D ::= A|\neg C|C \sqcap D|C \sqcup D|\exists R.C|\forall R.C$$

where $A \in N_C$, $R \in N_R$, and C, D are concepts.

Practical usage of Definition 1.1.2 is shown in Example 1.1.2.

Example 1.1.2 ***Complex concept***

$$\neg Mother$$

$$Mother \sqcup Father$$

$$\exists hasChild.Person$$

$$\forall likes.Food$$

Complex concept uses following constructors: $\neg$, $\sqcup$, $\sqcap$, $\exists$ and $\forall$. Constructor $\neg$ is negation, constructor $\sqcup$ is or and constructor $\sqcap$ is and. Constructors $\exists$ is existential restriction and $\forall$ is called value restriction.

There are two concepts that are always in ontology. $\top$ (top) stays for everything. Each concept belongs under $\top$ which means that each concept is on left side of subsumption if on right side is only $\top$. Second concept is $\bot$ (bottom) and it stays for nothing which means that each concept is on the right side of subsumption if there is only $\bot$ on the left side. Formally these two concepts can be written as we see in Definition 1.1.3.

Definition 1.1.3 ***Top and Bottom***

$$\top \equiv A \sqcup \neg A$$

$$\bot \equiv A \sqcap \neg A$$

$\mathcal{ALC}$ description logic uses axioms in order to model some situation. Ontology (Fitz-Gerald and Wiggins, 2010) is used to formally describe these axioms. The purpose of ontology is to describe relationships between entities in a formal language. Every ontology has its own knowledge base. Knowledge base is a set of TBox axioms and ABox axioms. Ontology is described by knowledge base. Knowledge base is defined in Definition 1.1.4.

Definition 1.1.4 ***Knowledge base***

Knowledge base ($\mathcal{KB}$) is an ordered pair of TBox $\mathcal{T}$ and ABox $\mathcal{A}$.

An example for knowledge base is shown in Example 1.1.3.

Example 1.1.3 ***Knowledge base***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \\ jack : Academician \\ jane : Scientist \\ john : Professor \end{array} \right\}$$

There are two types of axioms, the first is TBox (Definition 1.1.5) and the second one is ABox (Definition 1.1.6). In TBox the subsumption symbol ($\sqsubseteq$) is used. Let's explain this symbol on an example $Mother \sqsubseteq Parent$. $Mother$ is always a $Parent$ but $Parent$ does not always have to be a $Mother$. TBox represents axioms that model ontology. Each axiom explains the relationship between entities in this axiom.

Definition 1.1.5 ***TBox***

A TBox $\mathcal{T}$ is a finite set of GCI axioms ϕ of the form:

$$\phi ::= C \sqsubseteq D$$

where C, D are any concepts.

Example 1.1.4 ***TBox***

$$\mathcal{T} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \end{array} \right\}$$

ABox on the other side does not model ontology but creates a database of facts. It contains set of assertion axioms that can be called facts. Fact is a direct assertion of an individual to a concept.

Definition 1.1.6 ***ABox***

An ABox $\mathcal{A}$ is a finite set of assertion axioms ϕ of the form:

$$\phi ::= a : C | a, b : R$$

where $a, b \in N_I$, $R \in N_R$ and C is any concept.

Example 1.1.5 ***ABox***

$$\mathcal{A} = \left\{ \begin{array}{c} jack : Academician \\ jane : Scientist \\ john : Professor \end{array} \right\}$$

To have a better understanding of $\mathcal{ALC}$ DL we can translate sentences into $\mathcal{ALC}$ description logic (Example 1.1.6).

Example 1.1.6 ***ABox***

Everybody who is sick, is not happy	$Sick \sqsubseteq \neg Happy$
Cat and dogs are animals.	$Cat \sqcup Dog \sqsubseteq Animal$
Every person owns a house.	$Person \sqsubseteq \exists owns.House$

1.1.2 $\mathcal{ALC}$ Semantics

An interpretation is a pair of a domain and an interpretation function. The domain is a set of values that represents concepts in the interpretation (Definition 1.1.7). The result of interpretation function is different for individuals, concepts and roles. If we use the interpretation function on an individual the result is an element from the domain. If we use it on the concepts the result is a set of elements from the domain. If we use it on the roles the result is a set of pairs of elements from the domain.

Definition 1.1.7 ***Interpretation***

An interpretation of a given knowledge base $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ *is a pair* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *which contains a domain* $\Delta^{\mathcal{I}}$ *and an interpretation function* $\cdot^{\mathcal{I}}$*. Domain can not be empty.*

The interpretation function is following:

$$a^{\mathcal{I}} \in \Delta^{\mathcal{I}} \; \forall a \in N_I$$

$$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \; \forall A \in N_C$$

$$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \; \forall R \in N_R$$

The interpretation of complex concepts is recursively defined:

$$\neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$

$$C \sqcap D^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$$

$$C \sqcup D^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

$$\exists R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \exists y \in \Delta^{\mathcal{I}} : \langle x, y \rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$$

$$\forall R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \forall y \in \Delta^{\mathcal{I}} : \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$$

Example 1.1.7 ***Interpretation***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \\ jack : Academician \end{array} \right\}$$

$$\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$$

$$\Delta^{\mathcal{I}} = \{P, S, A, AP\}$$

$$Professor^{\mathcal{I}} = \{P\}$$

$$Scientist^{\mathcal{I}} = \{S\}$$

$$Academician^{\mathcal{I}} = \{A\}$$

$$AssocProfessor^{\mathcal{I}} = \{AP\}$$

$$jack^{\mathcal{I}} = A$$

The interpretation satisfies (Definition 1.1.8) an axiom according to its type. We know three types of axioms. The first is an axiom from TBox, the second one is an assertion axiom to a concept and the third one is an assertion to a role.

Definition 1.1.8 ***Satisfaction*** $\models$

Given an axiom ϕ, an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ satisfies $\phi(\mathcal{I} \models \phi)$ depending on its type:

$$\mathcal{I} \models C \sqsubseteq D \text{ iff } C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$$

$$\mathcal{I} \models a : C \text{ iff } a^{\mathcal{I}} \in C^{\mathcal{I}}$$

$$\mathcal{I} \models a, b : R \text{ iff } \langle a^{\mathcal{I}}, \in b^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$$

Finding a model (an interpretation satisfying $\mathcal{KB}$, Definition 1.1.9) is crucial for consistency checking. If we find at least one model the knowledge base $\mathcal{KB}$ is consistent. We can have more models for one knowledge base. In Example 1.1.8 there are two showed models but there can be more models.

Definition 1.1.9 ***Model***

An interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ is a model of a DL knowledge base $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ iff $\mathcal{I}$ satisfies every axiom in TBox $\mathcal{T}$ and ABox $\mathcal{A}$.

Example 1.1.8 ***Model***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \\ jack : Academician \end{array} \right\}$$

$$\mathcal{I}_1 = \{\Delta^{\mathcal{I}} = \{P, S, A, AP\},$$
$$Professor^{\mathcal{I}} = \{P\}, Scientist^{\mathcal{I}} = \{S\}, Academician^{\mathcal{I}} = \{A\}, AssocProfessor^{\mathcal{I}} = \{AP\}$$
$$jack^{\mathcal{I}} = A\}$$

$$\mathcal{I}_2 = \{\Delta^{\mathcal{I}} = \{A\},$$
$$Professor^{\mathcal{I}} = \{A\}, Scientist^{\mathcal{I}} = \{A\}, Academician^{\mathcal{I}} = \{A\}, AssocProfessor^{\mathcal{I}} = \{A\}$$
$$jack^{\mathcal{I}} = A\}$$

Definition 1.1.10 ***Consistency***

A knowledge base $\mathcal{KB}$ is consistent iff $\mathcal{KB}$ has at least one model $\mathcal{I}$.

We are familiar with more decision problems, satisfiability, subsumption ($\sqsubseteq$), equivalence ($\equiv$) and disjointness. Satisfiability means that concept is satisfiable in regard to $\mathcal{KB}$ if we can find such a model of a knowledge base for which holds that interpretation of that concept is not empty. Subsumption between two concepts must hold that interpretation of left-sided concept is a proper subset of interpretation of right-sided concept in each possible model of a knowledge base. Equivalence is similar to subsumption but the difference is that the interpretations of both concepts must be equal in each possible model of a knowledge base. Disjointness means that intersection of the interpretations of both concepts must be an empty set in each possible model of a knowledge base.

Definition 1.1.11 ***Decision problems***

Given a DL $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$, and two concepts C,D, we say that:

- *C is satisfiable w.r.t. $\mathcal{KB}$ iff there is such a model $\mathcal{I}$ of $\mathcal{KB}$ for which holds that $C^{\mathcal{I}} \neq \emptyset$;*
- *C is subsumed by D w.r.t. $\mathcal{KB}$ (denoted $\mathcal{KB} \models C \sqsubseteq D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{KB}$;*
- *C and D are equivalent w.r.t. $\mathcal{KB}$ (denoted $\mathcal{KB} \models C \equiv D$) iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{KB}$;*
- *C and D are disjoint w.r.t. $\mathcal{KB}$ iff $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ in every model $\mathcal{I}$ of $\mathcal{KB}$.*

If $\mathcal{KB} = \emptyset$ then we say that decision problems (satisfiability, subsumption, equivalence and disjointness) of concepts are defined in general by definition and we use notation such as we omit $\mathcal{KB} \models$ from the notation.

1.2 DL Tableau Algorithm

Following definitions (Baader et al., 2003) describe rules and terms that are used by this algorithm. DL Tableau algorithm proves satisfiability of a concept or checks consistency of a given knowledge base. By proving satisfiability the input for the algorithm is concept C and TBox $\mathcal{T}$. By checking consistency the input for the algorithm is a knowledge base $\mathcal{KB}$. In both versions of this algorithm holds that each concept must be in NNF (negation normal form, Definition 1.2.1). Negation normal form is such a form that each negation is pushed in front of atomic concept inside a complex concept.

Definition 1.2.1 ***NNF***

A concept C is in NNF (negation normal form) if and only if the complement constructor $\neg$ only occurs in front of atomic concept symbols inside C.

Example 1.2.1 ***NNF***

$$nnf(\neg(C \sqcap D)) = \neg C \sqcup \neg D$$

$$nnf(\neg(C \sqcup D)) = \neg C \sqcap \neg D$$

$$nnf(\neg\exists R.C) = \forall R.\neg C$$

$$nnf(\neg\forall R.C) = \exists R.\neg C$$

DL Tableau algorithm creates a CTree (Completion tree, Definition 1.2.2). CTree proves whether model exists or not.

Definition 1.2.2 ***Completion tree***

A completion tree (CTree) is a triple $T = (V, E, \mathcal{L})$ where (V, E) is a tree and $\mathcal{L}$ is a labeling function which means that: $\mathcal{L}(x)$ is a set of concepts $\forall x \in V$ and $\mathcal{L}(\langle x, y\rangle)$ is a set of roles $\forall \langle x, y\rangle \in E$.

At first the algorithm creates node and initializes it with the input concept. Then the algorithm applies tableau rules for given DL. If the algorithm applies each rule until any rule can be applied and it finds no clash then result is that concept C is satisfiable w.r.t. $\mathcal{T}$. Children of a node are called successors or R-successors (Definition 1.2.3).

Definition 1.2.3 ***Successor, R-successor***

Given a CTree T = (V,E, $\mathcal{L}$) and $x, y \in V$ we say that:

- *y is a successor of x iff $\langle x, y\rangle \in E$*
- *y is an R-successor of x iff $\langle x, y\rangle \in E$ and $R \in \mathcal{L}(\langle x, y\rangle)$.*

A clash containing Ctree is such a tree that one branch of this tree contains a concept and simultaneously its negation too. Clash is defined in Definition 1.2.4.

Definition 1.2.4 ***Clash***

There is a clash in a CTree $T = (V, E, \mathcal{L})$ if and only iff for some $x \in V$ and for some concept C both $C \in \mathcal{L}(x)$ and $\neg C \in \mathcal{L}(x)$.

Example 1.2.2 ***Clash***

$$\mathcal{L}(s_0) = \{C, D, \neg D\}$$

Definition 1.2.5 ***Clash-free CTree***

A CTree $T = (V, E, \mathcal{L})$ is clash-free iff none of the nodes in V contains a clash.

There can be a situation that some concept can lead to infinite looping and algorithm would never stop. This situation is called blocking (Definition 1.2.6). An example for that is: $Person \sqsubseteq \exists hasParent.Person$. That is why the algorithm uses Blocking rule.

Definition 1.2.6 ***Blocking***

Given a CTree $T = (V, E, \mathcal{L})$ a node $x \in V$ is blocked if it has an ancestor y such that: either $\mathcal{L}(x) \subseteq \mathcal{L}(y)$ or y is blocked.

DL Tableau algorithm proves concept satisfiability in regard of TBox (Definition 1.2.7). Input for DL Tableau algorithm is concept C and TBox $\mathcal{T}$. Output is a boolean value which is true if concept C is satisfiable w.r.t. $\mathcal{T}$, false otherwise.

Definition 1.2.7 ***Algorithm – Concept satisfiability***

Input: *concept C and $\mathcal{T}$ in NNF*

Output: *answers whether concept C is satisfiable w.r.t. $\mathcal{T}$ or not*

Steps:

1. *Initialize a new CTree $T := (\{s_0\}, \emptyset, \{s_0 \rightarrow \{C\}\})$;*

2. *Apply tableau rules for TBoxes while at least one rule is applicable;*

3. *Answer "C is satisfiable w.r.t. $\mathcal{T}$" if T is clash-free. Otherwise answer "C is unsatisfiable w.r.t. $\mathcal{T}$".*

Definition 1.2.8 ***$\mathcal{ALC}$ tableau rules for TBoxes***

- $\sqcap - rule$: *if $C_1 \sqcap C_2 \in \mathcal{L}(x)$ and $x \in V$ and $\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$ and x is not blocked then $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$*

- $\sqcup - rule$: *if $C_1 \sqcup C_2 \in \mathcal{L}(x)$ and $x \in V$ and $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ and x is not blocked then either $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1\}$ or $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_2\}$*

- $\forall - rule$: *if $\forall R.C \in \mathcal{L}(x)$ and $x, y \in V$ and y is R-successor of x and $C \notin \mathcal{L}(y)$ and x is not blocked then $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$*

- $\exists - rule$: *if $\exists R.C \in \mathcal{L}(x)$ and $x \in V$ with no R-successor y and $C \in \mathcal{L}(y)$ and x is not blocked then $\mathcal{V} = \mathcal{V} \cup \{z\}, \mathcal{L}(z) = \{C\}$ and $\mathcal{L}(\langle x, z\rangle) = \{R\}$*

- $\mathcal{T} - rule$: *if $C_1 \sqsubseteq C_2 \in \mathcal{T}$ and $x \in V$ and nnf$(\neg C_1 \sqcup C_2) \notin \mathcal{L}(x)$ and x is not blocked then $\mathcal{L}(x) = \mathcal{L}(x) \cup \{nnf(\neg C_1 \sqcup C_2)\}$*

Knowledge base is a pair of a TBox and an Abox and yet only a TBox was mentioned in connection with the DL Tableau algorithm. If we also have a given ABox the algorithm must be modified. The algorithm checks no longer concept satisfiability w.r.t. $\mathcal{T}$ but checks consistency of a given $\mathcal{KB}$. The same condition holds for the result as in proving concept satisfiability. If there is a found clash the knowledge base is not consistent otherwise the knowledge base is consistent (Definition 1.2.9). Difference between the first version of this algorithm is that here are used named nodes. Name of the node is an instance of a concept that comes from ABox.

Definition 1.2.9 ***Algorithm – Consistency checking***

Input: $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ *in NNF*

Output: *answers whether* $\mathcal{KB}$ *is consistent or not*

Steps:

1. *Initialize a CTree T as follows:*
   - $V := \{a \mid \textit{individual } a \textit{ occurs in } \mathcal{A}\}$;
   - $E := \{\langle a, b\rangle \mid a, b : R \in \mathcal{A} \textit{ for some role } R\}$;
   - $\mathcal{L}(a) := \{nnf(E) \mid a : E \in \mathcal{A}\}$ *for all* $a \in V$;

     $\mathcal{L}(\langle a, b\rangle) := \{R \mid a, b : R \in \mathcal{A}\}$ *for all* $\langle a, b\rangle \in E$;
2. *Apply tableau rules for TBoxes while at least one rule is applicable;*
3. *Answer "*$\mathcal{KB}$ *is consistent" if T is clash-free. Otherwise answer "*$\mathcal{KB}$ *is inconsistent".*

Let's introduce an Example 1.2.3 where we know a knowledge base $\mathcal{KB}$. DL Tableau algorithm checks if knowledge base is consistent or not.

Example 1.2.3 ***(DL Tableau algorithm)***

$$\mathcal{KB} = \left\{ \begin{array}{c} Student \sqsubseteq \exists attends.MathCourse \\ MathCourse \sqsubseteq Course \\ jack : Student \end{array} \right\}$$

$$\begin{tikzcd} \text{jack} \arrow[rrr, bend left, "\text{attends}"] & & & m_1 \end{tikzcd}$$

$\mathcal{L}(jack) \ =$

$\{Student, \neg Student \sqcup$

$\exists attends.MathCourse,$

$\exists attends.MathCourse, \neg Logic \sqcup$

$Course, \neg MathCourse\}$

$\mathcal{L}(m_1) \quad =$

$\{MathCourse, \neg Student \sqcup$

$\exists attends.MathCourse, \neg Student,$

$\neg MathCourse \ \sqcup$

$Course, Course\}$

The algorithm starts with the ABox assertion axioms. It selects the axiom $jack$: $Student$. At first it creates a node $jack$ with its label $\mathcal{L}(jack)$. From the assertion axiom $jack$: $Student$ algorithm knows that jack is a Student and adds the concept $Student$ to the label $\mathcal{L}(jack)$. Then the algorithm chooses another axiom from the knowledge base. There are no more assertion axioms, so it starts with the axioms from TBox. It takes $Student \sqsubseteq \exists attends.MathCourse$. It must convert it into the negation normal form which is $\neg Student \sqcup \exists attends.MathCourse$. The algorithm applies $\sqcup - rule$. According to this rule it must choose $\exists attends.MathCourse$, otherwise there would be a clash. Now it has to apply $\exists - rule$ so a new node l_1 will be created with its label $\mathcal{L}(m_1)$. Concept $MathCourse$ will be added into this label. Let's go back to the $jack$ node. The algorithm has to finish adding all axioms of TBox form. It will add the second axiom from the knowledge base in negation normal form as $\neg MathCourse \sqcup Course$. Now it does not matter which one algorithm chooses if $Course$ or $\neg MathCourse$ because any of these two does not cause a clash. Now the algorithm goes back to the m_1 node and continues to add axioms from the knowledge base similarly as in the node $jack$. It starts with axiom $\neg Student \sqcup \exists attends.MathCourse$, it will choose the first one $\neg Student$. Then continues with axiom $\neg MathCourse \sqcup Course$. It can not choose $\neg MathCourse$ because there would be immediately a clash. So it chooses $Course$.

The created tree contains two nodes ($jack$ and m_1) and one edge between them. The edge represents that $jack$ attends some MathCourse m_1. There is not any instance of a concept $MathCourse$ so the node m_1 does not represent a concrete instance of the concept $MathCourse$ but represents some instance. There is no clash in both nodes so the answer of the DL Tableau algorithm is that the given knowledge base $\mathcal{KB}$ is consistent.

2 Abduction

Generally in logic we are familiar with three ways of thinking. Deduction, induction and abduction. The most known and natural for humans is probably deduction. All three ways are dealing with the following parts: theory, data and effect. In the description logic we can translate the theory as a knowledge base, data as the explanations and the effect as an observation.

By deduction a knowledge base and the explanations is known, the observation is missing and the goal is to deduce the missing observation. By induction is known an observation and an explanation but the knowledge base is unknown. By abduction is known a knowledge base and an observation but the explanation is a subject of searching. All definitions are from article by Pukancová and Homola (2017).

2.1 ABox Abduction

In the description logics abduction (Definition 2.1.1) is used when we are not familiar with the explanation $\mathcal{E}$ but we know a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$. It is important to know that we are looking for minimal explanations. A minimal explanation is such an explanation that it does not exist any other explanation that would be a subset of that minimal explanation.

Definition 2.1.1 ***(Abduction)***

Given a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$, an abductive explanation is such an explanation $\mathcal{E}$ that satisfies $\mathcal{KB} \cup \mathcal{E} \models \mathcal{O}$.

Definition 2.1.2 ***(Correct explanation)***

$$\mathcal{E} \text{ is consistent if } \mathcal{E} \cup \mathcal{KB} \not\models \bot;$$

$$\mathcal{E} \text{ is relevant if } \mathcal{E} \not\models \mathcal{O};$$

$$\mathcal{E} \text{ is explanatory if } \mathcal{KB} \not\models \mathcal{O}$$

Definition 2.1.3 ***(Minimal explanation)***

Minimal explanation is such an explanation that it does not exist any other explanation that would be a subset of this explanation.

For better understanding of what ABox abduction is let's introduce a few examples. The first Example 2.1.1 is easy, the searched explanation is obvious but it will demonstrate the problem. The second Example 2.1.2 is not so obvious that is why we have to use an algorithm to compute the solution. For computing the solution we use the **Minimal Hitting Set algorithm**.

Example 2.1.1 ***(ABox Abduction - Emotion)***

$$\mathcal{KB} = \left\{ \ Sick \sqsubseteq \neg Happy \ \right\}$$

$$\mathcal{O} = \left\{ \ mary : \neg Happy \ \right\}$$

In Example 2.1.1 we search for the explanations. In this easy assignment it is obvious that what we look for is that Mary is sick. If Mary is not happy she must be sick. Formally written solution to this abduction problem is the following:

$$\mathcal{E} = \left\{ \ mary : Sick \ \right\}$$

Example 2.1.2 ***(ABox Abduction - Academy)***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ \\ AssocProfessor \sqsubseteq Professor \end{array} \right\}$$

$$\mathcal{O} = \left\{ \ jack : Academician \ \right\}$$

In Example 2.1.2 we search minimal explanations when we know that *jack* is an Academician. If *jack* is an Academician he must be a Professor or a Scientist. If he is a Professor he must be also an AssocProfessor. This is an oversimplified explanation how we can retrieve correct explanations. For the better introduction into this algorithm we will explain it step by step. We know the observation so we know that jack is an Academician, that is a fact. In our first axiom in TBox it is written that if somebody is an Academician he is also a Professor or a Scientist. He can be both but at least one of them but that we are not able to determine. That is why both explanations are correct. The second axiom claims that if somebody is a Professor he must be also an AssocProfessor. In this part we do not explain the whole algorithm yet because it will be explained in our next chapter Minimal Hitting Set algorithm but we need to have at least an idea of how it works. So as we already know the correct explanations are following:

$$\mathcal{E}_1 = \left\{ \ jack : Professor \ \right\}$$

$$\mathcal{E}_3 = \left\{ \ jack : Scientist \ \right\}$$

$$\mathcal{E}_3 = \left\{ \ jack : AssocProfessor \ \right\}$$

2.2 Minimal Hitting Set Algorithm

In this part we will explain the algorithm: Minimal hitting set algorithm. This algorithm is invented by Raymond Reiter (1987). At first we will declare terms that we will be using. The first term is a hitting set. Let's have collection of sets C. Hitting set (Definition 2.2.1) is such a set that has non-empty intersection with each set of collection C.

Definition 2.2.1 ***Hitting set***

A hitting set for a collection of sets C is a set $H \subseteq \cup_{S \in C}$ such that $H \cap S$ is not an empty set for each $S \in C$.

HS-tree (Definition 2.2.2) is such a tree that contains nodes with three possible values. The first value is check mark, the second is cross mark and the third is a set. Each edge has its own label that determines a path from the parent to node. Labeled edges determines path from the root to node n. HS-tree must be the smallest tree for the collection of sets C.

Definition 2.2.2 ***HS-tree***

Let C be a collection of sets. An HS-tree T for C is a smallest edge-labeled and node-labeled tree with the following properties:

- *The root is labeled by $\checkmark$ if C is empty. Otherwise the root is labeled by an arbitrary set of C.*
- *For each node n of T, let $H(n)$ be the set of edge labels on the path in T from the root to a node n. The label for n is any set $\sigma \in C$ such that $\Sigma \cap H(n) = \emptyset$, if such a set Σ exists. Otherwise, the label for n is $\checkmark$. If n is labeled by the set Σ, then for each $\sigma \in \Sigma$, n has a successor n_σ joined to n by an edge of labeled by σ.*

The Minimal Hitting Set algorithm (Definition 2.2.3) generates a HS-tree. It is a breadth-first search algorithm which means that algorithm creates all nodes in one breadth level and then it goes to the next level if possible. The following definition describes rules of how the algorithm behaves.

Definition 2.2.3 ***Minimal Hitting Set algorithm***

- *Generate the pruned HS-tree breadth-first, generating all nodes at any fixed level in the tree before descending to generate the nodes at the next level.*
- *Reusing node labels: If node n has already been labeled by a set $S \in C$ and if n' is a new node such that $H(n') \cap S = \emptyset$, then label n' by S.*
- *Tree prunning:*
  - *If node n is labeled by $\checkmark$ and node n' is such that $H(n) \subseteq H(n')$, then close the node n'. A label is not computed for n' nor are any successor nodes generated.*
  - *If a node n has been generated and node n' is such that $H(n') = H(n)$, then close node n'.*
  - *If nodes n and n' have been labeled by sets S and S' of C, respectively and if S' is a proper subset of S, then for each $\alpha \in S - S'$ mark as redundant the edge from node n labeled by α. A redundant edge, together with the subtree beneath it, may be removed from the HS-tree while preserving the property that the resulting pruned HS-tree will yield all minimal hitting sets for C.*

Reiter's algorithm is used to compute minimal hitting sets. Input is a set of sets. The goal of Reiter's algorithm is to compute minimal hitting sets from this input. It means that the resulting hitting sets will have a non-empty intersection with each set from the input.

Let's show an Example 2.2.1 with F as the input set and HS as the result.

Example 2.2.1 ***(Minimal Hitting Set)***

$$F = \{\{a,b\},\{b,c\},\{a,c\},\{b,d\},\{b\}\}$$
$$HS = \{\{a,b\},\{b,c\}\}$$

As we can observe in this example the first result set has a non-empty intersection with each set of F and the second result set also has an intersection with each set of F. So we can determine that these resulting sets are definitely minimal hitting sets. They are both minimal because there is no other set that is smaller and at the same time has non-empty intersection with each set of F.

For better and easier understanding of this algorithm we can visualize it and explain (Figure 1).

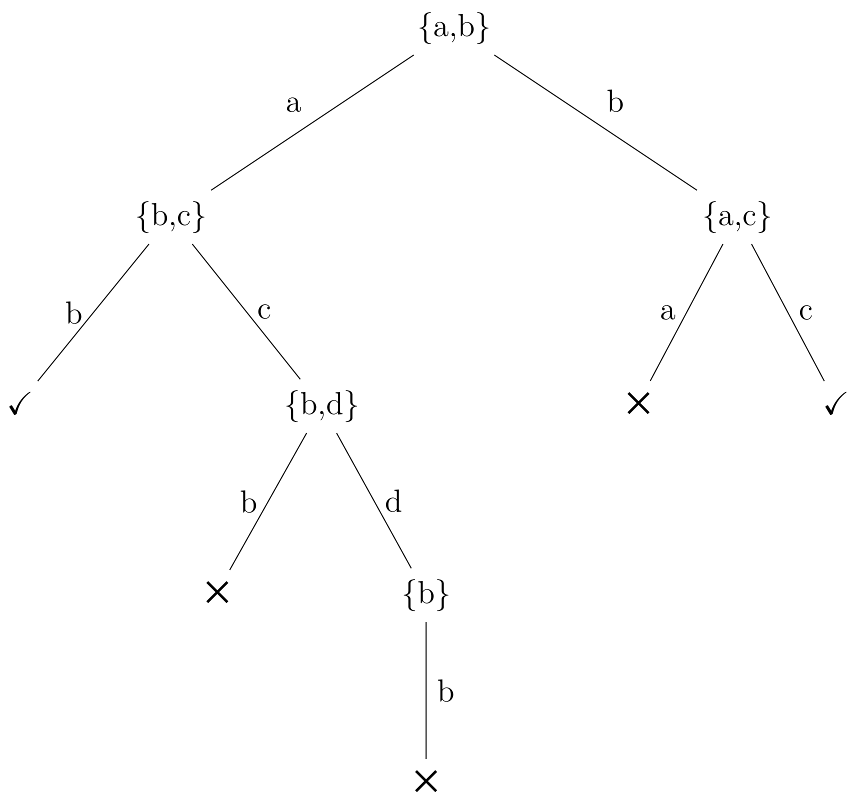

Figure 1: Minimal Hitting Set

An aim is to create a tree where nodes can have three types of a value. A node can be a set from F, check mark or cross mark. Each edge is labeled by one element of some set from F. At first we have to create a root. The root node is the first set $\{a, b\}$ from F. It will have exactly two children because the size of the node label is two.

Now we have to decide what will be the child's node. Let's check the left child. A labeled path from root to the left node is $\{a\}$. If $\{a\}$ has an intersection with each set from F we can add check mark as the node. But that is not our case so we add such a set that has no intersection with the set $\{a\}$, as the node. That set is $\{b, c\}$. Let's continue with the right child of the root. A labeled path from this node to the root is $\{b\}$. That does not have non-empty intersection with each set of F so we add set $\{a, c\}$ as node. We continue breadth-first with $\{b, c\}$ node. Left child has the path labeled by $\{a, b\}$ which has an intersection with each set of F and it fulfills the condition to be a possible minimal hitting set. Now we have to check if this possible minimal hitting set is really minimal. It is definitely a hitting set but we look only for the minimal hitting sets. If a checked node already exists and it is a proper subset of this possible hitting set we have to add cross mark as the node because it is not a minimal hitting set, otherwise we add check mark as the node. Similarly we continue in this algorithm and we get two minimal hitting sets: $\{a, b\}$ and $\{b, c\}$.

2.3 Optimizations

Optimizations by Greiner et al. (1989) and Wotawa (2001) were added to Reiter's original algorithm a few years later.

Greiner claims that Reiter's way of creating a tree handles some situations incorrectly. The base algorithm is correct but prunning may lead to lose minimal hitting sets. Proposed way of preventing this lost is to use a directed acyclic graph instead of a tree. We did not use this optimization because in our case this situation can not happened. They provided a different approach for prunning. But the new prunning rule is applicable only then if input set F contains a strict superset of some other set from F. Our input does not contain such a superset.

The next optimization by Wotawa disclaims the proposed way by Greiner. He has returned to using a tree. He tries to save time and make the algorithm quicker. His idea is to create deterministic tree where nodes are sorted from left to right where left means the smallest size of an edge label and right means the biggest size of an edge label in one breadth level. In our case this optimization is also useless because in our tree the size of the edge label is always equal.

2.4 MergeXPlain Algorithm

The main goal of MergeXPlain algorithm is to find minimal conflicts. That is a complex task that can be done by this algorithm. This algorithm is proposed by Shchekotykhin et al. (2015). The main algorithm is MergeXPlain (Algorithm 2) that finds minimal conflicts. MXP uses an algorithm that is called QuickXPlain (Algorithm 1). The QuickXPlain algorithm uses divide and conquere strategy and it returns one minimal conflict if such conflict exists.

To characterize a conflict we have to first characterize what is a diagnosable system (Definition 2.4.1) and diagnosis (Definition 2.4.2) from Reiter (1987).

Definition 2.4.1 ***Diagnosable System***

A diagnosable system is a pair (SD, COMPS) where SD is a system description (a set of logical sentences) and COMPS represents the system's components (a finite set of constants).

A diagnosis problem is when observations (OBS) are inconsistent with the diagnosable system.

Definition 2.4.2 ***Diagnosis***

Given a diagnosis problem (SD, COMPS, OBS), a diagnosis is a minimal set $\Delta \subseteq COMPS$ *such that* $SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg\{AB(c)|c \in COMPS\backslash\Delta\}$ *is consistent.*

A minimal conflict seems to be the same as minimal diagnosis. Finding all minimal conflicts corresponds to finding all minimal hitting sets (Reiter, 1987). Conflict CS is minimal if exists no proper subset of CS such as this subset is a conflict. Conflict is defined in Definition 2.4.3.

Definition 2.4.3 ***Conflict***

A conflict CS for (SD, COMPS, OBS) is a set $\{c_1, ..., c_k\} \subseteq COMPS$ *such that* $SD \cup OBS \cup \{\neg AB(c_i)|c_i \in CS\}$ *is incosistent.*

2.4.1 QXP Algorithm

The QuickXPlain (QXP) algorithm is designed for computation of explanations. In its current state it has the ability to return one conflict which means one possible explanation. It is a recursive algorithm showed in Algorithm 1.

At first the QuickXPlain algorithm checks for consistency of $\mathcal{B}$ and $\mathcal{C}$. There is no conflict if they are consistent so algorithm returns "no conflict". Otherwise the next check in the algorithm is if $\mathcal{C}$ is empty. If it is empty there can not be any conflict. Then algorithm calls method *GETCONFLICT* with parameters $\mathcal{B}$, $\mathcal{B}$ and $\mathcal{C}$. The *GETCONFLICT* method checks if the second parameter D is not empty and if $\mathcal{B}$ is not consistent. Otherwise if $\mathcal{C}$ has size 1 then the result is $\mathbb{C}$ because $\mathcal{C}$ can not be divided into two sets anymore. It is already a conflict. If algorithm continues then $\mathcal{C}$ is randomly divided into two sets $\mathcal{C}_1$ and $\mathcal{C}_2$. Then the method *GETCONFLICT* is recursively called, first with parameters $\mathcal{B} \cup \mathcal{C}_1$, $\mathcal{C}_1$ and $\mathcal{C}_\in$, second with $\mathcal{B} \cup D_2$, D_2

and $\mathcal{C}_1$. After return from recursion the results D_1 and D_2 are joined and returned as result.

Algorithm 1 QXP($\mathcal{B}$,$\mathcal{C}$)

```
Input: $\mathcal{B}$: Background theory, $\mathcal{C}$: the set of possibly faulty constraints
Output: A minimal conflict $CS \subseteq C$
 1: if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then
 2:     return "no conflict"
 3: else if $\mathcal{C} = \emptyset$ then
 4:     return $\emptyset$
 5: end if
 6: return $GETCONFLICT(\mathcal{B}, \mathcal{B}, \mathcal{C})$
 7: function GetConflict($\mathcal{B}, D, \mathcal{C}$)
 8:     if $D \neq \emptyset \wedge \neg isConsistent(\mathcal{B})$ then
 9:         return $\emptyset$
10:     end if
11:     if $|\mathcal{C}| = 1$ then
12:         return $\mathcal{C}$
13:     end if
14:     Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
15:     $D_2 \leftarrow$ GetConflict($\mathcal{B} \cup \mathcal{C}_1, \mathcal{C}_1, \mathcal{C}_2$)
16:     $D_1 \leftarrow$ GetConflict($\mathcal{B} \cup D_2, D_2, \mathcal{C}_1$)
17:     return $D_1 \cup D_2$
18: end function
```

2.4.2 MXP Algorithm

The MergeXPlain (MXP) algorithm uses QXP algorithm. The MXP calls QXP repeatedly for computation of an conflict. Difference between QuickXPlain and MergeXPlain algorithm (Algorithm 2) is that the MergeXPlain algorithm can return multiple conflicts at a time and the QuickXPlain algorithm only one in such a case that at least one conflict exists.

Algorithm 2 MXP($\mathcal{B}$,$\mathcal{C}$)

Input: $\mathcal{B}$: Background theory, $\mathcal{C}$: the set of possibly faulty constraints

Output: Γ, a set of minimal conflicts

```
 1: if $\neg isConsistent(\mathcal{B})$ then
 2:     return "no solution"
 3: else if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then
 4:     return $\emptyset$
 5: end if
 6: $\langle \_, \Gamma \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}$)
 7: return $\Gamma$
 8: function FindConflicts($\mathcal{B}$,$\mathcal{C}$) returns a tuple $\langle \mathcal{C}', \Gamma \rangle$
 9:     if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then
10:         return $\langle \mathcal{C}, \emptyset \rangle$
11:     else if $|\mathcal{C}| = 1$ then
12:         return $\langle \emptyset, \{\mathcal{C}\} \rangle$
13:     end if
14:     Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
15:     $\langle \mathcal{C}'_1, \Gamma_1 \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}_1$)
16:     $\langle \mathcal{C}'_2, \Gamma_2 \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}_2$)
17:     $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$
18:     while $\neg isConsistent(\mathcal{C}'_1 \cup \mathcal{C}'_2 \cup \mathcal{B})$ do
19:         $X \leftarrow$ GetConflict($\mathcal{B} \cup \mathcal{C}'_2$,$\mathcal{C}'_2$,$\mathcal{C}'_1$)
20:         $CS \leftarrow X \cup$ GetConflict($\mathcal{B} \cup X$,$X$,$\mathcal{C}'_2$)
21:         $\mathcal{C}'_1 \leftarrow \mathcal{C}'_1 \backslash \{a\}$ where $a \in X$
22:         $\Gamma \leftarrow \Gamma \cup \{CS\}$
23:     end while
24:     return $\langle \mathcal{C}'_1 \cup \mathcal{C}'_2, \Gamma \rangle$
25: end function
```

At the beginning of the MergeXPlain algorithm, the algorithm checks for consistency of $\mathcal{B}$. If $\mathcal{B}$ is already inconsistent it has no point to continue and algorithm returns no solution. If $\mathcal{B}$ and $\mathcal{C}$ are consistent together, no conflict can be found and algorithm returns empty set. Otherwise algorithm calls a method *FINDCONFLICTS* with parameters $\mathcal{B}$ and $\mathcal{C}$. This method checks consistency of $\mathcal{B}$ and $\mathcal{C}$ joined together. If they are consistent method returns a pair of $\mathcal{C}$ and empty set. If $\mathcal{C}$ has size one, method returns a pair of empty set and $\{\mathcal{C}\}$. This condition is here because if $\mathcal{C}$ has size one it can not be divided into two disjoint non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$. If $\mathcal{C}$ has size zero then the first condition is fulfilled because it is consistent with the set $\mathcal{B}$. But if the set $\mathcal{C}$ has size bigger than one then algorithm continues and splits the set $\mathcal{C}$ into two disjoint non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$.

The method recursively calls itself, first with the set $\mathcal{C}_1$ as the second parameter and secondly with the set $\mathcal{C}_2$ as the second parameter. This is repeated until the method is emerged from both recursions and found explanations are joined together into variable Γ. The pair which is returned by function *FINDCONFLICTS* returns remaining elements of $\mathcal{C}$ as the first argument and set of explanations as the second argument. The method did not check the whole search space. It can happened that some part of explanation is in the set $\mathcal{C}_1$ and other part is in the set $\mathcal{C}_2$. To find at least some explanations there is a while loop that should find more explanations.

The following while loop is running until the set $\mathcal{C}'_1$ is either empty or consistent with the union of the sets $\mathcal{B}$ and $\mathcal{C}'_2$. In the while loop from the set $\mathcal{C}'_1$ are removed elements that is why this condition can be fulfilled. In the while loop the method calls the above mentioned method *GETCONFLICT* which returned one conflict if exists. This conflict is stored into variable X. Then the *GETCONFLICT* method is called again but with X as parameter and the result is stored into variable CS. It is also joined with the previous result X. Then comes the most important part of this loop. From the set $\mathcal{C}'_1$ is removed α where α belongs to the set X. This can be interpreted differently. It could mean that always only one element is removed from the set $\mathcal{C}'_1$. Or it could mean that more elements are removed from this set. In the line below Γ represents all found explanations in this run of the method. After jumping from while loop the methods returns a pair of remaining literals ($\mathcal{C}_1' \cup \mathcal{C}_2'$) and found explanations.

This algorithm was not used on solving ABox abduction problem yet. Until now this algorithm was designed to detect conflicts. This conflict detection uses divide and conquer strategy as is described in Algorithms 2 and 1.

3 Our Approach

In this chapter we would like to introduce two ways of abductive reasoning which we have implemented. The first one uses Minimal Hitting Set (MHS) and is based one the Minimal Hitting Set algorithm from Reiter (1987). But it is modified to include ABox abduction according to previous work from Pukancová and Homola (2016). Our approach based on the Minimal Hitting Set algorithm is described in the Section 3.1. The second approach is the MergeXPlain (MXP) which is based on the MergeXPlain from Shchekotykhin et al. (2015). This approach is described in the Section 3.2.

3.1 MHS-Based Approach

Reiter's algorithm considers minimal hitting sets but we have to take into account the conditions for abductive solutions defined in the Algorithm 3. So our input data are a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$. The result from this algorithm is the set of all the minimal explanations.

3.1.1 MHS Algorithm

The MHS algorithm (defined in the Algorithm 3) requires following parameters: a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$. At the beginning the algorithm starts with finding a negation model $\neg M$. The model M is acquired from the union of the knowledge base $\mathcal{KB}$ and the negation of the observation $\neg\mathcal{O}$ (line 1). How to create a model, respectively a negation model is explained in the Section 3.1.2. After retrieving the negation model $\neg M$ the algorithm checks if the negation model $\neg M$ is null or not (line 2). If we can find no negation model that means that we can not continue in the algorithm and we can not return any explanations. We can not continue in the algorithm because we need some non-empty negation model which would be a value for the root r.

Algorithm 3 MHS($\mathcal{KB}$,O)

```
Require: knowledge base $\mathcal{KB}$, observation $O$
Ensure: set $\mathcal{S_E}$ of all explanations of $\mathcal{P} = (\mathcal{KB}, O)$ of the class Abd
 1: $M \leftarrow$ a model $M$ of $\mathcal{KB} \cup \{\neg O\}$
 2: if $M = $ null then
 3:     return "nothing to explain"
 4: end if
 5: create new HS-tree $T = (V, E, L)$ with root $r$
 6: label $r$ by $L(r) \leftarrow$ Abd($M$)
 7: for each $\sigma \in L(r)$ create a successor $n_\sigma$ of $r$ and label the resp. edge by $\sigma$
 8: $\mathcal{S_E} \leftarrow \{\}$
 9: while there is next node $n$ in $T$ w.r.t. BFS do
10:     if $n$ can be pruned then
11:         prune $n$
12:     else if there is a model $M$ of $\mathcal{KB} \cup \{\neg O\} \cup H(n)$ then
13:         label $n$ by $L(n) \leftarrow$ Abd($M$)
14:     else
15:         $\mathcal{S_E} \leftarrow \mathcal{S_E} \cup \{H(n)\}$
16:     end if
17:     for each $\sigma \in L(n)$ create a successor $n_\sigma$ of $n$ and label the resp. edge by $\sigma$
18: end while
19: return $\mathcal{S_E}$
```

If we can continue in the algorithm we proceed with line 5. The algorithm creates a new HS-tree T initialized with a root r. The node r is labeled by the negation model $\neg M$. Then we continue with line 6 and 7 where we create successors for the root r. Each successor has labeled its edge from the root r to the successor's node with one element of the negation model $\neg M$. Each successor's node is not labeled yet, it will be labeled in the next step. The next step is to label the successor's nodes by breadth first search which means that we will label each successor from left to right. This node labeling is done in the while loop which starts in line 9 and is explained in the Section 3.1.3.

After we decide how the successor's nodes will be labeled we continue with line 10. We can prune node n if there is no point to keep this path anymore. Pruning rules are defined in the Section 3.1.4. If the node n is labeled by a negation model $\neg M$ then we repeat above mentioned steps like by the root r. That means that we create its successors and labeled their edges and nodes. This node is added to the HS-tree T. Otherwise the path from the root r to the node n creates a minimal explanation and we will add this minimal explanation to the set of all minimal explanations $\mathcal{S}_{\mathcal{E}}$ (line 15). Until we still have some node in our HS-tree T we repeat this process and acquire minimal explanations. If there is no node in the HS-tree T the algorithm breaks from the while loop and returns the set of all the found minimal explanations $\mathcal{S}_{\mathcal{E}}$ in line 19.

This algorithm has an exponential time complexity but it is sound and complete and it always terminates. It also means that this algorithm will find all minimal explanations eventually.

3.1.2 Model Creation

The model creation requires following parameters: the knowledge base $\mathcal{KB}$, the negation of the observation $\neg\mathcal{O}$ and the path $H(n)$ which represents path from the root r to node n. A new model is a set of ABox assertions. Each assertion has an ABox form $a : A$ or $a : \neg A$. The individual we take is from the observation $\mathcal{O}$. Each concept is from the knowledge base $\mathcal{KB}$. First we create a new set S for the model M. Secondly we take the negation of the observation $\neg\mathcal{O}$ and add it to the set S. Then we add ABox assertions from the path $H(n)$. Then we have to add the remaining ABox assertions to the set S. The set S has to contain all concepts from knowledge base $\mathcal{KB}$ in the ABox assertions. Which means that we have to iterate through all concepts and evaluate if each concept is already in some assertion of the S or not. Let's show it with concept C and an individual a. The concept C is in some assertion of the set S if $a : C \in S$ or $a : \neg C \in S$. If we evaluate that the concept C is in any assertion of the set S we have to add either $a : C$ or $a : \neg C$. We decide which assertion we use with the help of a reasoner. We ask the reasoner if after adding assertion $a : C$ is the set S still consistent. If yes then we add assertion $a : C$ to the set S otherwise we add assertion $a : \neg C$. We continue with this approach with all remaining concepts. We add one optimization and that is that we remove the negation of the observation $\neg\mathcal{O}$ after the set S is completed. We do this because in the MHS algorithm we need to get a negation of the model M. The set S is the model M. But we need to return a negation of the model M which means that we negate each assertion of the set S. Then we create a set $\neg S$ that is shaped from the negated assertions from the set S. That is our negation model $\neg M$. And now back to the explanation why we removed the negation of the observation $\neg\mathcal{O}$ from the set S. It is because the set $\neg S$ would contain the actual observation $\mathcal{O}$ and that would not lead to relevant explanation.

3.1.3 Node Labeling

In the MHS algorithm nodes can have three types of label. The first type is a new negation of the model that is acquired from the union of the knowledge base $\mathcal{KB}$, the negation of the observation $\neg\mathcal{O}$ and the parameter $H(n)$. The parameter $H(n)$ is a path from the root r to node n. The second type of label is a check mark (✓). That means that the path $H(n)$ is a possible explanation. It does not have to be a minimal explanation. The last type of label is a cross mark (✕) which represents such a path $H(n)$ that is inconsistent or not relevant.

3.1.4 Pruning

The pruning rules are simple. We can not have a path $H(n)$ from the root r to a node n if there is a clash which leads to inconsistency. Or if there is already such a path $H(n')$ that $H(n') \subseteq H(n)$. The path $H(n)$ could be an explanation but it could not be a minimal explanations because it already exists an explanation that is a subset of this path $H(n)$. Such a path that fulfills at least one of this two conditions could not lead to a possible minimal explanations and that is why we have to prune this path.

3.2 MXP-Based Approach

We have used MergeXPlain as is described by Shchekotykhin et al. (2015). At first we need to create input data for MXP from given knowledge base $\mathcal{KB}$ and observation $\mathcal{O}$. The MXP algorithm takes parameters $\mathcal{B}$ as background theory and $\mathcal{C}$ as the set of possibly faulty constraints. The parameter $\mathcal{B}$ is created by our input data which are the knowledge base $\mathcal{KB}$ and the negation of the observation $\mathcal{O}$. We create a set of possibly faulty constraints as parameter $\mathcal{C}$. This set $\mathcal{C}$ is created from the knowledge base $\mathcal{KB}$. At first we need to acquire all concepts from the knowledge base $\mathcal{KB}$ and all individuals from the observation $\mathcal{O}$. Then we pair each individual witch each concept and its negation which results in a set of ABox assertions. We exclude one pair that is equal to the observation $\mathcal{O}$. For better understanding let's show it on the Example 3.2.1.

Example 3.2.1 ***Transformation of input data in MXP***

Let's have knowledge base that contains following concepts: $N_C = \{A, B, C\}$

And observation: $\mathcal{O} = \{d : B\}$

Then we have one individual from observation: $N_I = \{d\}$

So the parameter $\mathcal{B} = \{A, B, C, d : \neg B\}$

So set $\mathcal{C} = \{d : A, d : \neg A, d : \neg B, d : C, d : \neg C\}$

After transformation we can begin with the algorithm. The MXP algorithm checks for consistency of given input which is knowledge base joined with other constraints. For consistency checking we use reasoner. Reasoner decides if the given input is consistent or not. We have implemented the algorithm exactly as is shown in Algorithm 2. There is only one thing that is not exactly specified and that is a line 21 in the method *FINDCONFLICTS*. We need to reduce either the set $\mathcal{C}'_1$ or the set $\mathcal{C}'_2$. If we would not do that we would be infinitely looping in the while loop. So now we have to decide which one will be removed. Our approach is working so that always first element from the set X is removed from the set $\mathcal{C}'_1$. This can work in such a case that no conflict is overlapping with another. But if there are conflicts that are overlapping each other, this way will not find all conflicts, it will find only its subset.

MergeXPlain algorithm is sound because it always terminates. But it is not complete because it can not always find all conflicts. This problem happens when they are explanations that are overlapping each other. Let's show the contradiction on following Example 3.2.2.

Example 3.2.2 ***MergeXPlain contradiction example***
Let $\mathcal{K} = \{A \sqcap B \sqsubseteq D, A \sqcap C \sqsubseteq D\}$ *and let* $O = \{a : D\}$.

Let us ignore negated ABox expressions and start with $Abd = \{a : A, a : B, a : C\}$. *There are two minimal explanations of* $\mathcal{P} = (\mathcal{K}, O)$*:* $\{a : A, a : B\}$, *and* $\{a : A, a : C\}$. *Calling MXP(*$\mathcal{K} \cup \{\neg O\}, Abd$*), it passes the initial tests and calls* FINDCONFLICTS*(*$\mathcal{K} \cup \{\neg O\}, Abd$*).*

FINDCONFLICTS *needs to decide how to split* $\mathcal{C} = Abd$ *into* $\mathcal{C}_1$ *and* $\mathcal{C}_2$. *Let us assume the split was* $\mathcal{C}_1 = \{a : A\}$ *and* $\mathcal{C}_2 = \{a : B, a : C\}$. *Since both* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are now conflict-free w.r.t.* $\mathcal{K} \cup \{\neg O\}$, *the two consecutive recursive calls return* $\langle \mathcal{C}'_1, \emptyset \rangle$ *and* $\langle \mathcal{C}'_2, \emptyset \rangle$ *where* $\mathcal{C}'_1 = \{A(a)\}$ *and* $\mathcal{C}'_2 = \{a : B, a : C\}$.

In the while loop, GETCONFLICT*($\mathcal{K} \cup \{\neg O\} \cup \{a : B, a : C\}$, $\{a : B, a : C\}$, $\{a : A\}$) returns $X = \{a : A\}$ while* GETCONFLICT*($\mathcal{K} \cup \{\neg O\} \cup \{a : A\}$, $\{a : A\}$, $\{a : B, a : C\}$) returns $a : B$, and hence the first confclit $\gamma = \{b : A, a : B\}$ is found and added into Γ.*

However, consecutively $a : A$ is removed from C'_1 leaving it empty, and thus the other conflict is not found and $\Gamma = \{\{b : A, a : B\}\}$ is returned.

The MXP algorithm returns a set of conflicts. But not all conflicts are minimal explanations. That is why we have to filter these conflicts. Given result set of conflicts often contains conflicts that are not consistent because they contain a clash. We iterate over each conflict and first we check if the conflict is consistent. If it is not consistent we do not consider it as an explanation. If it is consistent it is a possible minimal explanation. Then we have to check if given conflict is consistent with the knowledge base. If that is true we say that the conflict is a minimal explanation and add it to the resulting set. After iteration the resulting set contains a set of conflicts that represent minimal explanations.

3.3 Implementation

Our work is implemented in Java 1.8. Both algorithms, MHS and MergeXPlain, are implemented in one project. Our work can be found on github: `https://github.com/katuskaa/MasterThesis`.

The project is using maven for dependency management. We needed to use library for OWL API and all the three reasoners. We use it also for generating a jar file from our project. Our program does not have a graphical user interface. It can run in some IDE for Java for example (IntelliJ IDEA) or our program can create a jar file that can be launched in a terminal. In both ways our program expects some arguments.

The useful argument is help (-h or –help) which shows what arguments can be used and if they are optional or not. This command also shows what values can be used with the arguments. Necessary arguments (-f "relativePath/ontology.owl") are the path to file which contains ontology (*.owl format). Another mandatory argument is an observation (-o "a:A") that can be given in the Manchester syntax. Our algorithms work with single observation but that can be also complex, for example (-o "d:(A and B) or C"). Other arguments are optional. Following two arguments change reasoner and algorithm. To define which reasoner should be used, use an argument (-r "pellet"). If this argument is not used, default reasoner is used and that is HermiT. To define which algorithm is used, use an argument (-s "mergexplain"). Default for this algorithm is MHS method. The last two arguments are also optional and works only with MHS algorithm. To define how many depths of HS-tree should be generated, use an argument (-d n), where n specifies a depth of the HS-tree. To define a timeout for MHS method, use an argument (-t s), where s specifies a number of seconds. The timeout parameter defines how long should program maximally run. After achieving this timeout, algorithm is terminated and returns the minimal explanations found until timeout was reached.

In both approaches we need to use reasoner. In our implementation we use three reasoners from which you can always choose one that will be used in the computation. These reasoner that we use are, HermiT, JFact and Pellet. We use them only for consistency checking which means that in both algorithms (MHS, MXP) we need to ask the reasoner if our ontology joined with constraints is consistent in current state. The ontology is changing in the flow of algorithm, so we ask many times if ontology is consistent or other words if the ontology is clash free. With these reasoners we communicate through OWL API which is a common api for all reasoners. Each reasoner has its own implementation of this api.

Our program has common part that use both approaches or better to say that this part is evaluated before the approach is called. Let's call it preprocessing of the input data. First we have to map our arguments to a configuration structure where we remember which reasoner and which method should be used. We can remember depth and timeout if given too. Second we have to parse the observation into our common structure. Then we have to load the given ontology into selected reasoner through OWL API. We only change the instance of a reasoner, then we use the same implementation for all reasoners thanks to the OWL API.

After preprocessing selected approach is called. We use one interface that has a method *solve*. Both our approaches implement this interface and after selecting the method, correct instance is created and then we only call the method *solve* that is implemented in both approaches.

We implemented time measurement for the timeout. This timer does not count time from start to end or to timeout. It counts that time that the computer spends on doing this task (our program).

4 Evaluation

A preliminary experimental evaluation was conducted with implementations of MHS and MXP, both paired with three DL reasoners – Pellet, HermiT, and JFact. Both algorithms are implemented in Java and they communicate with the reasoners through OWL API.

The source code of both implementations is available online. [1]

The evaluation is split into two experiments. Experiment 1 is focused on computing explanations of size one. In this case the MHS algorithm can be made more effective by bounding the HS-tree depth, and on the other hand MXP is complete in this case. Experiment 2 was conducted without any constraints on the size of explanations, but a timeout needed to be set. Both experiments were focused on comparing execution times between the two approaches and the three reasoners. Each time was computed as an average value from ten runs with ten different observations.

4.1 Dataset and Methodology

Three ontologies were chosen. The Family ontology [2], is our own ontology of family relations. It is smaller, but it is particularly useful in this use case as it generates a number of explanations of size higher than one. The second ontology, LUBM (Lehigh University Benchmark Guo et al. (2005)), is a standard benchmark. The Beer ontology[3] was chosen. Both LUBM and Beer were chosen because of their larger size compared to the Family ontology, but on the other hand as in the case of many real world ontologies their axiomatic structure is less complex which implies that most if not all explanations

[1] https://github.com/katuskaa/MasterThesis
[2] http://dai.fmph.uniba.sk/~pukancova/aaa/ont/family2.owl
[3] https://www.cs.umd.edu/projects/plus/SHOE/onts/beer1.0.html

are of size one. In the Table 1 the structure of these three ontologies is described.

Table 1: Parameters of the ontologies

0.8 **Ontology**	**Concepts**	**Roles**	**Individuals**	**Axioms**
0.8 Family ontology	10	1	0	28
0.8 Beer ontology	58	9	0	165
0.8 LUBM	43	25	0	243

All experiments were done on a virtual machine with 8 cores (16 threads) of the processor Intel®Xeon®CPU E5-2695 v4, 32 GB RAM, 2.10GHz, running Ubuntu 18.04.1, while the maximum Java heap size was set to 4GB. Execution times were measured in Java using `ThreadMXBean` from the `java.lang.management` package. We used *user* time, that is the actual time without system overhead.

4.2 Experiment 1

The Experiment 1 compares execution times between the MHS and the MXP algorithm. The MHS algorithm was launched with the depth parameter and the depth was set to 1. The MXP algorithm was launched without any additional parameters.

The first tested ontology was Family ontology. The resulting plot we can see in the Figure 2. In all three cases the MHS algorithm was quicker because it was stopped after reaching depth 1 of generated HS-tree. The MXP algorithm was a little bit slower. In this graph time is measured in seconds so it is not a big difference between MHS and MXP algorithm. Especially in the case of JFact. The JFact reasoner finished with similar results in both approaches and with the MXP approach was the quickest reasoner. The HermiT reasoner was the slowest reasoner in all three cases and the Pellet reasoner wins with the MHS approach.

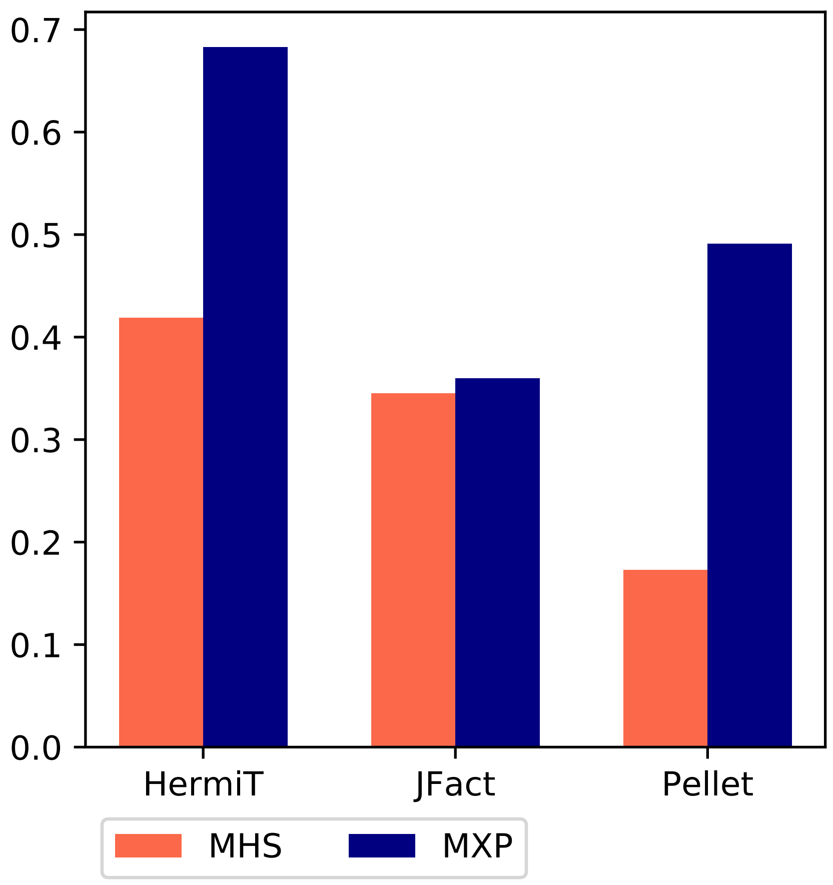

Figure 2: Family ontology - Experiment 1

The second tested ontology was the Beer ontology. The result from the experiment on this ontology we can see in the Figure 3. The Beer ontology is significantly bigger than the Family ontology. We can see that the JFact reasoner was definitely quicker with the MXP approach. On the other hand the Pellet reasoner was quicker with the MHS algorithm. The HermiT reasoner is also quicker with the MHS algorithm and it seems to have better performance for larger ontologies. The HermiT reasoner achieved worse results with the Family ontology comparing to the other reasoners and the Family ontology is smaller than the Beer ontology.

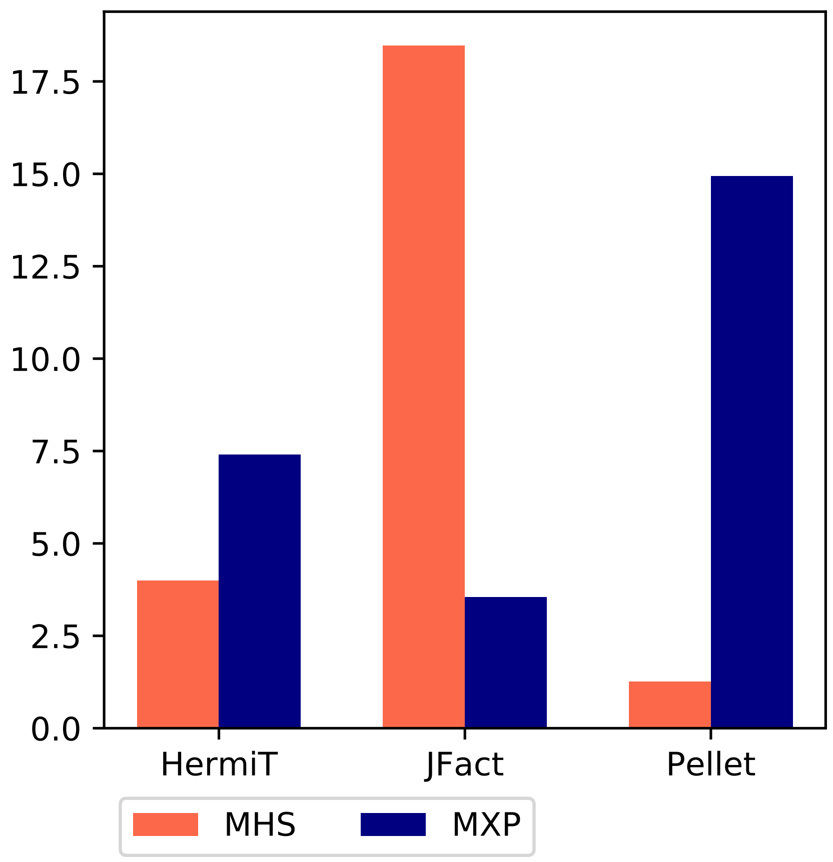

Figure 3: Beer ontology - Experiment 1

The last ontology that has been tested is the LUBM ontology. That is our largest ontology. The result from the experiment on this ontology is in the Figure 4. As we can see the JFact reasoner is again extremely slow with the MHS algorithm comparing to MXP algorithm. The Pellet reasoner acts similarly as it did in two previous cases, the MHS algorithm is quicker than the MXP algorithm. The HermiT reasoner is with both approaches in the middle, the MHS algorithm was quicker with the Pellet reasoner, the MXP algorithm was quicker with the JFact reasoner.

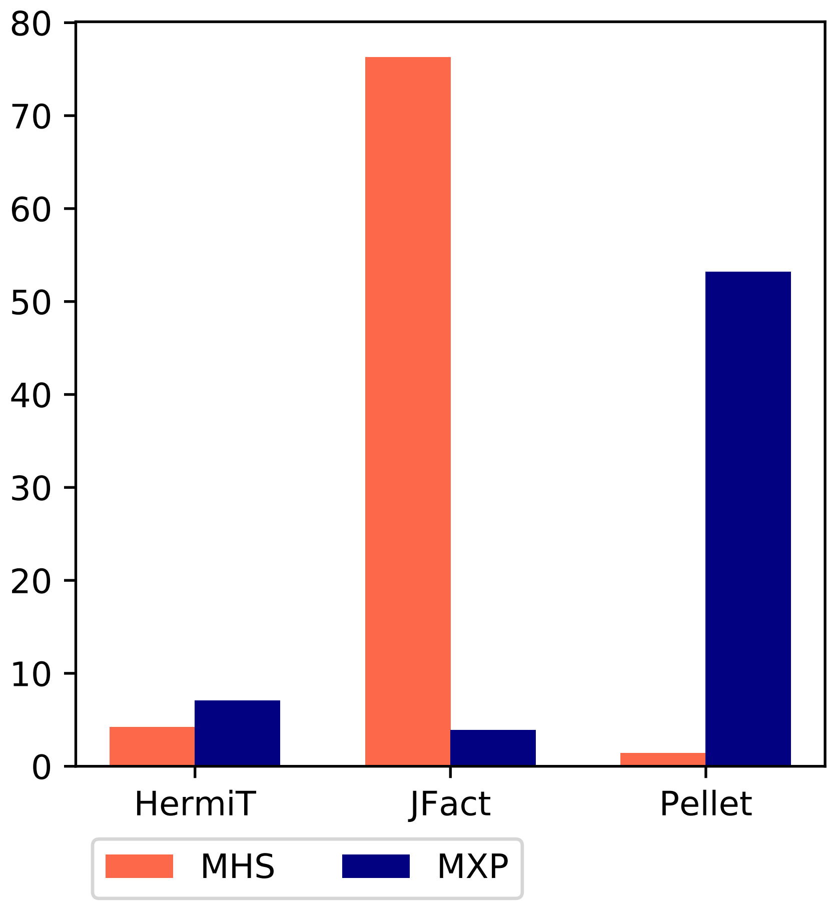

Figure 4: LUBM ontology - Experiment 1

4.3 Experiment 2

The Experiment 2 compares execution times between the MHS and the MXP algorithm. The MHS algorithm was launched with the timeout parameter and the timeout was set to 43200 seconds which is 12 hours. The MXP algorithm finished without any timeout. In this experiment we want to compare execution times and number of found explanations expressed in percents. We assume that the MHS algorithm will find all possible explanations and we take its count as maximum or in other words as 100% of all minimal explanations. Then we calculate how much percent of minimal explantions the MXP algorithm was able to find. In this experiment all resulting plots show the percentage of found minimal explanations at the top of the bar in the histogram. Under each plot the legend is displayed. Each legend describes if the used algorithm is the MHS or the MXP algorithm and in the case of the MHS algorithm the legend shows the depth of the HS-tree. Each depth is calculated as an average of all runs from all observations.

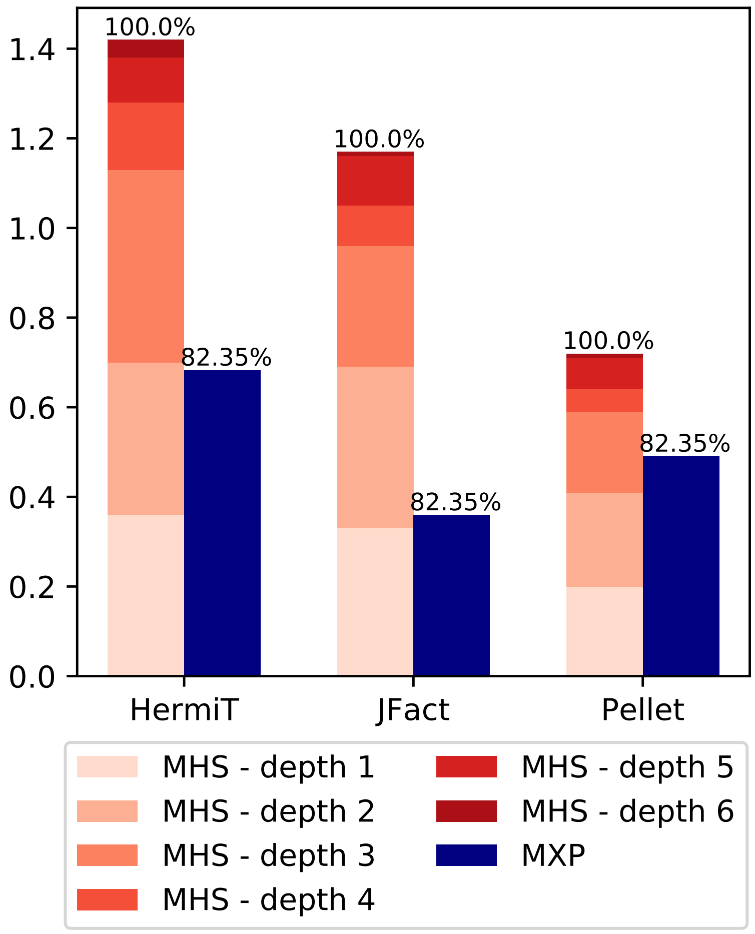

Figure 5: Family ontology - Experiment 2

The Family ontology is the first ontology that has been tested in our experiment. The resulting plot we can see in the Figure 5. This ontology is small so both algorithms finished their goal without timeout. In all three cases the MXP approach found exactly the same number of minimal explanations because the percentage is the same. This ontology is interesting for its structure. There is a multiple hierarchy in concepts and that is why a simple observation can have minimal explanations with cardinality bigger than one. The MXP algorithm is not able to find all minimal explanations which cardinality is bigger than one. So the resulting percentage is not 100%.

This experiment is measured in seconds but in the plot in the Figure 5 is seen that the MXP algorithm is with any of tested reasoners quicker than the MHS algorithm. The MHS algorithm has found all minimal explanations but it was definitely slower. The HermiT reasoner is the slowest and the Pellet reasoner is the quickest. Interesting is that we can not say that if one reasoner is the quickest with the MHS approach it will be also the quickest with the MXP approach. Great example is the JFact reasoner with the Family ontology. The JFact reasoner is the quickest reasoner with the MXP approach but is the second best reasoner with the MHS approach.

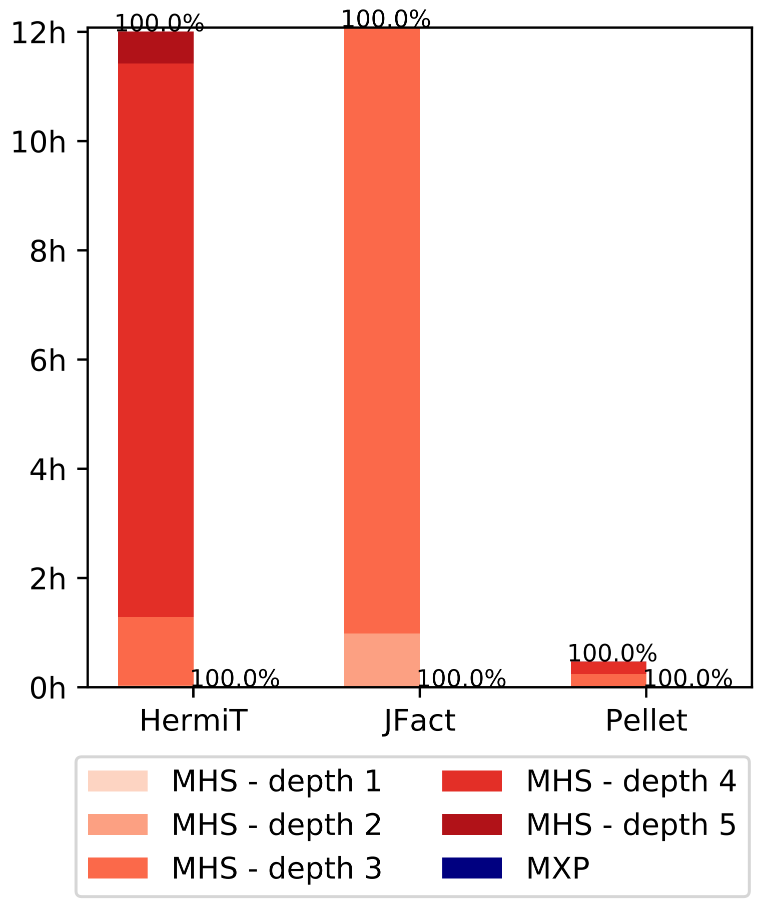

Figure 6: Beer ontology - Experiment 2

The Beer ontology is used as the second ontology in Experiment 2. We can see its result on Figure 6. The MXP approach takes only seconds and it is not even seen in the plot in the Figure 6. This plot is measured in hours because the MHS algorithm took a long time. But the MXP algorithm was running only a few seconds. For the MHS approach a 12 hours timeout has to be set. And we can see that HermiT and JFact have managed to run this 12 hours but Pellet did not. The reason why Pellet took shorter time is not that it was that quick. It is because the Pellet reasoner threw an exception (OutOfMemoryException). That is not problem with our algorithm, that is a problem with the reasoner. The Pellet reasoner does it unfortunately with bigger ontologies. We assume that minimal explanations that have been found before timeout are our 100% of all minimal explanations. And we compare this result with the MXP approach and its result. The Beer ontology is bigger than the Family ontology and it has different structure. The Beer ontology is flat, the maximum depth of a concept hierarchy is two. So that means that the explanations of simple observation can not have bigger size than one. The MXP algorithm has to find all minimal explanations in such a case. We have confirmed that the MXP algorithm has found all minimal explanations that were found by the MHS approach. We can observe that each reasoner finished in different depth before timeout. The HermiT reasoner managed to get into the third depth of the HS-tree, the JFact reasoner was only in the second and the Pellet reasoner seems to be the quickest because it was in the second depth before the other reasoners. Unfortunately the Pellet reasoner threw an OutOfMemoryException and could not finish.

The third ontology is the LUBM ontology in our experiment. Its result we can see in the Figure 7. The LUBM ontology is bigger than the Beer ontology but its structure is similar to the Beer ontology. Also the resulting plot is similar to the plot with the Beer ontology. Again the MXP approach took only seconds so that is why it is not even seen in the plot. The plot is measured in hours. The Pellet reasoner threw an OutOfMemoryException so it finished earlier. The HermiT and Pellet reasoner were terminated because of the timeout. Both reasoners did not finish naturally but we assume that they have found 100% of all explanations. The MXP approach found also 100% comparing to the MHS algorithm. But the LUBM ontology has similar structure as the Beer ontology. Its structure is flat so we can not find minimal explanations with cardinality bigger than one.

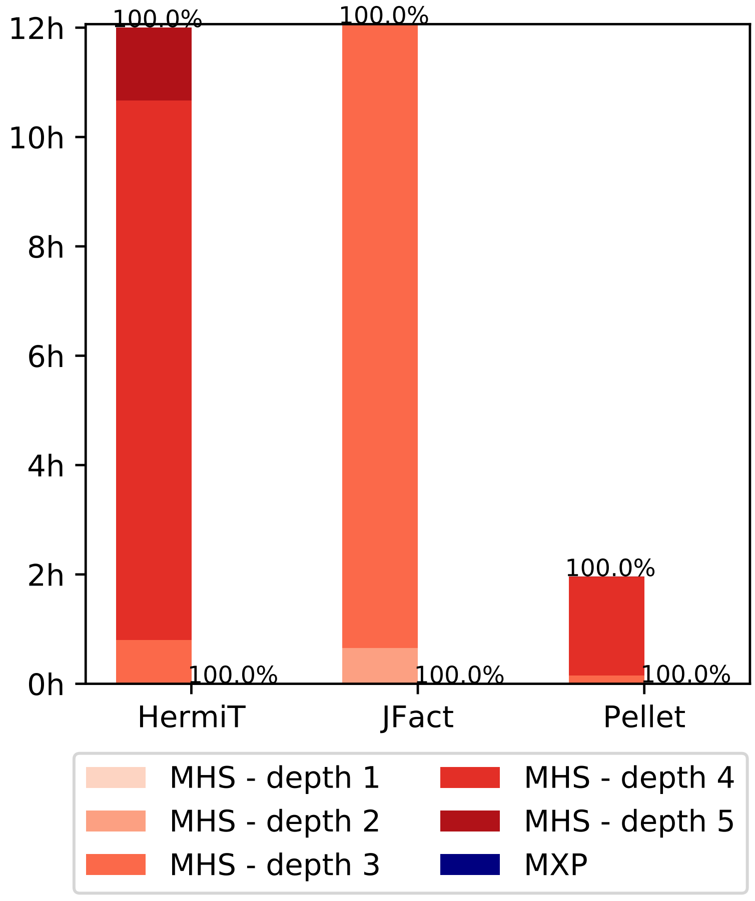

Figure 7: LUBM ontology - Experiment 2

4.4 Conclusion from Experiments

The Experiment 1 finished in all cases without timeout and without any exceptions. In general we can say that the MXP algorithm found all minimal explanations with cardinality equals to one. We know that the MHS algorithm is sound and complete so we assume that all found minimal explanations are really all and no other minimal explanation exists. That is why we say that the MXP algorithm has found all minimal explanations comparing to the MHS algorithm. By comparing result from the MXP and the MHS approach we can see that in this case is more useful to use the MHS algorithm because it is quicker, except JFact reasoner. But that could be because of the implementation of this reasoner. Interesting about JFact is that with the MHS approach is the slowest reasoner. But with the MXP approach it is not the slowest reasoner. So we can not say that the JFact reasoner is not suitable for our solution.

In the Experiment 2 a timeout needed to be set. The timeout was set to 12 hours and for bigger ontologies like the Beer and the LUBM ontology that was still not enough. They did not manage to finish and they were terminated because of the timeout. The Family ontology is a small ontology so there was not necessary to use a timeout. This ontology finished in seconds in our experiment. This experiment was about to find all possible minimal explanations, not just minimal explanations with size one. The goal of this experiment is to compare how much percents of all possible minimal explanations could be found by the MXP algorithm. The experiment did not find 100% of minimal explanations only with the Family ontology. That is because its structure is more hierarchical comparing to the other two ontologies. The experiment found all minimal explanations with these two ontologies. The execution times with both approaches are very different. With the Family ontology we can see that there is no such big difference, it is just a few seconds because the ontology is small. But with the Beer and the LUBM ontology we can see that the MHS algorithm took hours and it still was not enough and the MXP algorithm took only seconds and it found all minimal explanations.

Both experiments proved that the MXP algorithm is quick and able to find all minimal explanations of size one. It is more suitable for ontologies with flat hierarchy.

But that depends on the situation. If we prefer to find only minimal explanations with the cardinality equals to one, then we can definitely use the MXP approach. Or if we have ontologies like the Beer or the LUBM ontology that are big but flat then we can also use the MXP algorithm to find minimal explanations. But with this situation we can also use the MHS approach with defined depth parameter.

Conclusion

We wanted to find out if the MergeXPlain algorithm is the solution for the ABox abduction problem. From evaluation part we know that the MXP algorithm will find all minimal explanations of size one. But we can not guarantee that the MXP will find all minimal explanations. So the answer for our question is not so simple. We can not say that the MHS algorithm takes too long and can be replaced with the MXP algorithm. We can say that for some situations where we have ontologies with flat hierarchy we can use the MXP algorithm or we can use the MHS algorithm with depth parameter set to one.

The goal of our thesis was to implement two approaches and compare their execution times and results. We managed to implement both approaches to work correctly. In the process of implementation we have found out that the MXP algorithm is not able to find all minimal explanations. This happens if there are some minimal explanations which are overlapping each other. Then the MXP algorithm will not find them all, it will find only a subset of them. That is happening because the MXP algorithm is not able to search the whole space of possible explanations. But if we have minimal explanations that do not overlap each other, the MXP algorithm should find them. The evaluation showed that the MXP algorithm can be used if we look only for minimal explanations with size one. The Experiment 1 showed that if we look only for minimal explanations with cardinality equals to one we want to use the MHS algorithm because it is a little bit quicker. The Experiment 2 showed in the case with the Family ontology that even if the MXP won't find all explanations, it found 83% which is not so bad.

In our future work we think about combining these two approaches. The slow part of the MHS algorithm is to compute the negation model for given node. This step could be replaced with the MXP algorithm and that could improve the execution times.

References

Sebastian Rudolph. Foundations of description logics. In *Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*, pages 76–136, 2011. doi: 10.1007/978-3-642-23032-5_2. URL https://doi.org/10.1007/978-3-642-23032-5_2.

Stuart James Fitz-Gerald and Bob Wiggins. Staab, s., studer, r. (eds.), handbook on ontologies, series: International handbooks on information systems, second ed., vol. XIX (2009), 811 p., 121 illus., hardcover £164, ISBN: 978-3-540-70999-2. *Int J. Information Management*, 30(1):98–100, 2010. doi: 10.1016/j.ijinfomgt.2009.11.012. URL https://doi.org/10.1016/j.ijinfomgt.2009.11.012.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. ISBN 0-521-78176-0.

Júlia Pukancová and Martin Homola. Tableau-based abox abduction for the ALCHO description logic. In *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017.*, 2017. URL http://ceur-ws.org/Vol-1879/paper11.pdf.

Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. doi: 10.1016/0004-3702(87)90062-2. URL https://doi.org/10.1016/0004-3702(87)90062-2.

Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artif. Intell.*, 41(1):79–88, 1989. doi: 10.1016/0004-3702(89)90079-9. URL https://doi.org/10.1016/0004-3702(89)90079-9.

Franz Wotawa. A variant of reiter's hitting-set algorithm. *Inf. Process. Lett.*, 79(1):45–51, 2001. doi: 10.1016/S0020-0190(00)00166-6. URL https://doi.org/10.1016/S0020-0190(00)00166-6.

Kostyantyn M. Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. MergeXplain: Fast computation of multiple conflicts for diagnosis. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina*. AAAI Press, 2015.

Júlia Pukancová and Martin Homola. Tableau-based abox abduction for description logics: Preliminary report. In *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.*, 2016. URL http://ceur-ws.org/Vol-1577/paper_23.pdf.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

Appendices