A Machine Learning Approach for Optimizing Heuristic Decision-making in OWL Reasoners

Razieh Mehri-Dehnavi

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

November 2019

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Razieh Mehri-Dehnavi

Entitled: A Machine Learning Approach for Optimizing Heuristic Decision-making in OWL Reasoners

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Catherine Mulligan

_____ External Examiner
Dr. Bruce Spencer

_____ External to Program
Dr. Ferhat Khendek

_____ Examiner
Dr. Juergen Rilling

_____ Examiner
Dr. Rene Witte

_____ Thesis Supervisor
Dr. Volker Haarslev

Approved by _____

Dr. Leila Kosseim, Graduate Program Director

January 10, 2020 _____

Dr. Amir Asif, Dean
Gina Cody School of Engineering & Computer Science

ABSTRACT

A Machine Learning Approach for Optimizing Heuristic Decision-making in OWL Reasoners

Razieh Mehri-Dehnavi, Ph.D.

Concordia University, 2019

Description Logics (DLs) are formalisms for representing knowledge bases of application domains. The Web Ontology Language (OWL) is a syntactic variant of a very expressive description logic. OWL reasoners can infer implied information from OWL ontologies. The performance of OWL reasoners can be severely affected by situations that require decision-making over many alternatives. Such a non-deterministic behavior is often controlled by heuristics that are based on insufficient information. This thesis proposes a novel OWL reasoning approach that applies machine learning (ML) to implement pragmatic and optimal decision-making strategies in such situations.

Disjunctions occurring in ontologies are one source of non-deterministic actions in reasoners. We propose two ML-based approaches to reduce the non-determinism caused by dealing with disjunctions. The first approach is restricted to propositional description logic while the second one can deal with standard description logic.

The first approach builds a logistic regression classifier that chooses a proper branching heuristic for an input ontology. Branching heuristics are first developed to help Propositional Satisfiability (SAT) based solvers with making decisions about which branch to pick in each branching level.

The second approach is the developed version of the first approach. An SVM (Support Vector Machine) classier is designed to select an appropriate

expansion-ordering heuristic for an input ontology. The built-in heuristics are designed for expansion ordering of satisfiability testing in OWL reasoners. They determine the order for branches in search trees.

Both of the above approaches speed up our ML-based reasoner by up to two orders of magnitude in comparison to the non-ML reasoner.

Another source of non-deterministic actions is the order in which tableau rules should be applied. On average, our ML-based approach that is an SVM classifier achieves a speedup of two orders of magnitude when compared to the most expensive rule ordering of the non-ML reasoner.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Volker Haarslev for his countless support and mentoring throughout my PhD. I could have not finished this thesis without his inspiration, encouragement, valuable advice, and assistance. I was very lucky to do my Ph.D. under his excellent supervision.

I wish to extend my thanks to my committee members for their time and patience as well as their insightful remarks and constructive criticism during the past few years that helped me to formulate this thesis.

I also would like to thank all the staff members of the Department of Computer Science and Software Engineering at Concordia for their direct and indirect helps during my studies at Concordia University.

Finally, this work would not be possible without the unconditional support and help from my family, colleagues and friends throughout my Ph.D. and the completion of this dissertation.

Contents

List of Figures

List of Tables

List of Abbreviations

ABox	**A**ssertional **Box**
ATP	**A**utomated **T**heorem **P**rover
CD	**C**ompletely **D**efined
CNF	**C**onjunctive **N**ormal **F**orm
DAG	**D**irect **A**cyclic **G**raph
DL	**D**escription **L**ogic
DLCS	**D**ynamic **L**argest **C**ombined **S**um
DLIS	**D**ynamic **L**argest **I**ndividual **S**um
DPLL	**D**avis **P**utnam **L**ogemann **L**oveland
GCI	**G**eneral **C**oncept **I**nclusion
MI	**M**utual **I**nformation
ML	**M**achine **L**earning
NNF	**N**egation **N**ormal **F**orm
ORE	**O**WL **R**easoner **E**valuation
OWL	**W**eb **O**ntology **L**anguage
PCA	**P**rincipal **C**omponent **A**nalysis
QBF	**Q**uantified **B**oolean **F**ormulas
RBox	**R**ole **Box**
SAT	**SAT**isfiability
SMT	**S**atisfiability **M**odulo **T**heory

SVM **S**upport **V**ector **M**achine

TBox **T**erminological **Box**

Chapter 1

Introduction

1.1 Motivation

With an ever-increasing flow of data on the web, it is important to give both meaning and structure to its content. Semantic Web enables this task by using ontologies. Ontology languages are used to describe the web in a form that is understandable by machines. Depending on the context and usage of data, ontology languages are categorized into logic, graph, and frame-based [31]. The Web Ontology Language (OWL) is based on Description Logic (DL).

Description logic represents the application domain in the form of concepts and relationships between those concepts. DL ontologies consist of two levels: TBox and ABox. TBox describes the terminology of the application domain and ABox contains individual assertions.

DL or OWL reasoners are types of engines used to deduce new facts from the existing knowledge in ontologies. Reasoners are employed to facilitate machines' works with ontologies. They help to infer implicit information from ontologies by performing different reasoning tasks such as ontology

consistency, class satisfiability, hierarchical classification of named classes, query related tasks, etc. Speeding up the process of such reasoning tasks has always been a popular topic in the description logic community. Hence, a large amount of research has been devoted to the design of tableau optimization techniques for OWL reasoners in order to accelerate these reasoning tasks.

Tableau optimization techniques are categorized into different categories based on their purposes. These categories are Preprocessing, Satisfiability, TBox, ABox, and Nominals (refer to Section 2.3). Preprocessing makes the reasoning process much faster by simplifying axioms in input ontologies. Optimization techniques for satisfiability testing help with reducing the cost of tableau-based reasoning caused by non-determinism. TBox optimization contains the classification of concepts since the speed of reasoning can be improved by having a hierarchy relation of concepts. ABoxes are often large and applying optimization techniques such as breaking them into smaller ABoxes could have a drastic impact on the speed of reasoners. Nominals as special concept names are often very difficult to handle for reasoners and can make the reasoning process very slow; applying optimization techniques can avoid such slowness in the reasoning process.

Many of the tableau optimization techniques have a heuristic nature, meaning they require proper decision-making. Often, these optimization techniques employ non-deterministic rules due to the use of tableau algorithms. Not only non-deterministic rules such as disjunction exist in these techniques; but some of these techniques encounter other non-deterministic actions such as choosing between different algorithms or different combinations of rules.

For example, the *ToDo list* technique (refer to Section 2.3) controls the application of expansion rules. The optimal heuristic order for applying rules in ToDo list will have a great impact on reasoning speed. Moreover, in the literature review of this thesis, we investigate and gather other possible sources of tableau optimization techniques that in our opinion require heuristic decision-making (refer to Section 3.2).

1.2 Objective

Heuristic-guided optimization techniques in OWL reasoners are designed to be set manually (by a human). Therefore, they could slow down or speed up reasoners dramatically when encountering different ontologies. Each ontology has unique features that makes it different from other ontologies, i.e., each heuristic has a different impact on different ontologies. Hence, learning the right setting for heuristics when dealing with various ontologies can greatly improve these techniques.

Machine learning (ML) improves the decision-making process of optimization techniques in reasoners. Machine learning uses different algorithms and strategies to build models. Later, those models are used for predicting the proper action to take while making decisions in reasoners. To the best of our knowledge, machine learning techniques have never been integrated into OWL reasoners to deal with uncertain situations involving rules.

To fulfill our purpose, a machine learning model can be built to select the right heuristic in these optimization techniques. The learning-derived model is built based on features computed and extracted from ontologies. Therefore, the manual fine-tuning of heuristics for these reasoners can be

replaced by a new ML-based approach implementing automatic fine-tuning of heuristics to make the right choice.

1.3 Contribution

The basic DL $\mathcal{ALC}$ [41] forms the core of most DL languages and OWL. $\mathcal{ALC}$ contains as language elements conjunction, disjunction, negation, existential and universal quantification. A tableau algorithm for $\mathcal{ALC}$ contains various tableau rules. One of these rules is non-deterministic and deals with disjunctions, i.e., it decides which disjunct from a selected disjunction should be added to the current tableau. OWL reasoners typically apply various optimization techniques for selecting a disjunct with the goal to speed up reasoning by decreasing the size of the remaining search space. Many of these techniques are based on heuristics. A similar problem consists of determining the best order of rule applications to maximally speed up reasoning. Such a decision is usually based on heuristics, e.g., to determine whether the disjunction or the existential rule should be applied next. While heuristics-based optimization techniques are employed to make decisions in such situations, there does not yet exist an ML-based approach that controls automatic decision-making in such uncertain situations.

Our ML-based approaches for these uncertain situations encountering ontologies are based on an independent systematic analysis of heuristics to uncover ontology patterns and their associated features that are relevant for each specific optimization technique.

In this thesis, we focus on three of the tableau optimization techniques that require proper decision-making:

- *Semantic Branching*: Among the many optimization techniques, semantic branching (refer to Section 2.3.2) is directly affected by the nondeterminism caused by disjunctions. In this case, we intend to make the selection process for disjunctions more effective by applying an ML-based approach. Our approach, which is introduced in Section 4.3.1, is based on the selection of a proper branching heuristic for each input ontology.

- *Heuristics for Choosing Expansion-Ordering*: Further, we consider yet another optimization technique that contains expansion ordering heuristics with potential for improvement (refer to Section 2.3.2). This technique finds the proper order of disjuncts in disjunction expressions of ontologies. OWL reasoners have different built-in expansion-ordering heuristics developed for this purpose where the reasoner will select from them. An ML-based approach, which is introduced in Section 4.3.2, helps to find the right expansion-ordering.

- *Ordering the Application of Expansion Rules:* Another optimization technique that provides room for enhancement is called *ToDo list* or ordering of expansion rules (refer to Section 2.3.2). With *ToDo list* mechanism, one can arrange the order of applying different rules by giving each a priority. Our ML-based approach, which is introduced in Section 4.4, finds the right priority for each rule.

1.4 Structure of this dissertation

This dissertation is organized as follows:

- In Chapter 1, we introduced the motivation and objective of this thesis.
- Chapter 2 contains the preliminaries of description logic and tableau optimization techniques, as well as the required background knowledge on machine learning that is required to understand this thesis. Later, a brief overview of ontology features is given as well.
- Chapter 3 presents the literature review. We first propose the source of non-deterministic actions in tableau optimization techniques. Later, the related work and contributions of this thesis are discussed.
- In Chapter 4, the ML-based solutions developed by us to aid with decision-making situations in OWL reasoners are explained in detail. For each approach, the experiments and results are followed.
- Chapter 5 presents the conclusion and future work.

Chapter 2

Preliminaries and Theoretical Background

2.1 Introduction

This chapter provides relevant background for understanding this thesis. In Section 2.2, the main concepts of description logic are provided. That also includes the tableau-based reasoning procedure. In Section 2.3, most of the tableau optimization techniques that are designed for OWL reasoners are categorized and explained in detail. In Section 2.4, the background information on machine learning required for this thesis is given. This includes logistic regression, SVM, as well as feature engineering techniques. Finally, a brief overview of the categories of ontology features is given in Section 2.5.

2.2 Description Logic

Description logic (DL) is a logic-based knowledge representation language that is used to describe the knowledge of a domain and infer entailed knowledge from given knowledge. To build such a knowledge base, description logic uses as terminologies Concepts, Roles, and Individuals.

- Concepts describe a class of objects that share similar characteristics, e.g., the concept Man contains all individuals who are a man.
- Roles are binary relationships between individuals, e.g., role hasChild could be a relationship between two individuals where one is an instance of the concept Parent and the other is an instance of the concept Child.
- Individuals stand for objects or instances of classes, e.g., Josh could be an instance of the concept Parent meaning Josh is a parent.

Based on the concept language used to describe the composition of concepts and roles, different rules are employed, e.g., negation ($\neg$), conjunction ($\sqcap$), disjunction ($\sqcup$), universal value restriction of roles ($\forall$), and existential value restriction of roles ($\exists$). For example, consider the concept expression:

$$\text{Female} \sqcap \geq 2\,\text{hasChild} \sqcap \forall \text{hasChild.Female}$$

that describes females who have at least two children and all of their children are females. Three parts are combined through conjunction. The first part includes individuals who belong to the concept Female, the second part describes individuals connected to at least two individuals via the hasChild

role (using an at-least number restriction) and the third part describes individuals ($\forall$) that are connected via the hasChild role only to individuals of the class Female.

Description logic introduces various languages that permit specific rules in their syntax. To support our main point in this thesis, we only focus on $\mathcal{ALC}$, which can be extended by various concept constructors. For instance, *number restrictions* constrain the number of role successors ($\geq nR$ and $\leq nR$, where $n \in \mathbb{N}$).

Brief Introduction to Syntax and Semantics of $\mathcal{ALC}$ The formal semantics of a concept language is defined by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ in which $\Delta^{\mathcal{I}}$ is a domain and $\cdot^{\mathcal{I}}$ is an interpretation function. The domain is a non-empty set and the interpretation function maps: every instance name a to an element ($a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$); every atomic concept A to a subset ($A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$) and every atomic role R to a binary relation ($R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$). An interpretation $\mathcal{I}$ satisfies a concept C if and only if $C^{\mathcal{I}} \neq \emptyset$.

DL axioms belong to one of the three types: TBox, RBox, and ABox.

- TBox is the terminological part of an ontology and it contains a finite set of General Concept Inclusion (GCI) axioms. A GCI axiom is of the form of $C \sqsubseteq D$ where C and D are concepts. An interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Then, we say that D subsumes C and C subsumes D. An equivalence axiom can be written as two GCI axioms. Therefore, $C \equiv D$ also called definitional GCI can be written as $C \sqsubseteq D$ and $D \sqsubseteq C$. $C \sqsubseteq D$ is called a primitive GCI if C is an atomic concept and D a concept description.

- RBox is a finite set of role inclusion axioms $R \sqsubseteq S$ where R and S are roles. An interpretation $\mathcal{I}$ satisfies $R \sqsubseteq S$ if and only if $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$.

- ABox is the assertional part of an ontology that consists of a finite set of assertions about individuals (or instances of concepts) such as $C(a)$ where a is declared an instance of C, i.e., $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and roles where $R(a, b)$ states that a and b are individuals in the relationship R, i.e., $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.

An interpretation $\mathcal{I}$ satisfies a TBox $\mathcal{T}$ and its associated RBox $\mathcal{R}$ if and only if $\mathcal{I}$ satisfies all axioms in $\mathcal{T}$ and $\mathcal{R}$. An interpretation $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ and its associated TBox $\mathcal{T}$ and RBox $\mathcal{R}$ if and only if $\mathcal{I}$ satisfies all axioms in $\mathcal{T}$ and $\mathcal{R}$, and all assertions in $\mathcal{A}$. The syntax and semantics of $\mathcal{ALC}$ are presented in Table 2.1.

TABLE 2.1: Syntax and semantics of $\mathcal{ALC}$

Concept Constructors	**Syntax**	**Semantics**
Top	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Bottom	$\bot$	$\bot^{\mathcal{I}} = \emptyset$
Concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, A is atomic
Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Universal	$\forall R.C$	$\{x \mid \forall y : (x, y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
Existential	$\exists R.C$	$\{x \mid \exists y : (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

The standard DL inference services that are relevant to this thesis are concept satisfiability, TBox satisfiability (or consistency), i.e., $\top^{\mathcal{I}} \neq \emptyset$, and concept classification, which computes a taxonomy or subsumption hierarchy over all atomic concepts in a TBox $\mathcal{T}$.

Concept Inference Services

- Satisfiability: A concept C is said to be satisfiable considering the TBox $\mathcal{T}$ if and only if there is a model $\mathcal{I}$ in $\mathcal{T}$ where $C^{\mathcal{I}}$ is not empty or $\mathcal{T} \not\models C \equiv \bot$

- Subsumption: The concept C is subsumed by concept D considering the TBox $\mathcal{T}$ if and only if $\mathcal{T} \models C \sqsubseteq D$ or $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for every model $\mathcal{I}$ of $\mathcal{T}$.

- Equivalence: Two concepts C and D are equivalent considering the TBox $\mathcal{T}$ if and only if $\mathcal{T} \models C \equiv D$ or $C^{\mathcal{I}} \equiv D^{\mathcal{I}}$ holds for every model $\mathcal{I}$ of $\mathcal{T}$ or $C \sqsubseteq D$ and $D \sqsubseteq C$.

- Disjointness: Two concepts C and D are disjoint considering the TBox $\mathcal{T}$ if and only if $\mathcal{T} \models C \sqcap D \equiv \bot$ or $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ holds for every model $\mathcal{I}$ of $\mathcal{T}$.

It is worth to mention that the above tasks can also be translated to each other:

- Unsatisfiability/ Subsumption: C is unsatisfiable if and only if C is subsumed by $\bot$.

- Subsumption/Disjointness: D subsumes C if and only if $\neg D$ and C are disjoint.

- Disjointness/Equivalence: Two concepts C and D are disjoint if and only if $C \sqcap D$ is equivalent to $\bot$.

- Equivalence/Unsatisfiability: C and D are equivalent if and only if $(C \sqcap \neg D) \sqcup (D \sqcap \neg C)$ is not satisfiable.

Tableau-based Reasoning Tableau-based reasoning was first introduced for the $\mathcal{ALC}$ language but then extended for many concept languages. The algorithm's structure is based on completion graphs. To find out whether an ontology is satisfiable, the tableau-based reasoning procedure starts with loading the input ontology. We assume that all concept expressions are in negation normal form, i.e., a negation sign only occurs in front of atomic concepts. Tableau-based reasoning is controlled by a set of completion rules that are specific each for DL language and creates a graph by adding a first node. The status of this node is then updated and possibly other new nodes are added to the graph based on the tableau completion (or expansion) rules.

Completion Graph A Completion Graph $G = \langle V, E, \mathcal{L} \rangle$ is a directed graph and represents a tableau. Each node $x \in V$ is labeled with a set of concepts $\mathcal{L}(x)$. Each edge $(x, y) \in E$ is labeled with a set of roles $\mathcal{L}(x, y)$. A graph has a clash for a node x if $\mathcal{L}(x)$ contains $\bot$ or $\{\neg A, A\}$ with A an atomic concept. A graph is complete when no more expansion rules can be applied. A completion graph for $\mathcal{ALC}$ is expanded using tableau completion (or expansion) rules shown in Table 2.2, until no rule can be applied, i.e., the result is a complete graph. A complete graph is called clash-free if it does not contain a clash. If a complete graph for a concept C is clash-free, then C is satisfiable. Otherwise, C is unsatisfiable.

TABLE 2.2: Tableau completion rules for $\mathcal{ALC}$

$\sqcap$-**Rule**	**If** $(C \sqcap D) \in \mathcal{L}(x)$ and $\{C, D\} \not\subseteq \mathcal{L}(x)$
	then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C, D\}$
$\sqcup$-**Rule**	**If** $(C \sqcup D) \in \mathcal{L}(x)$ and $\{C, D\} \cap \mathcal{L}(x) = \emptyset$
	then
	set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{E\}$ with $E \in \{C, D\}$
$\forall$-**Rule**	**If** $\forall R.C \in \mathcal{L}(x)$ and there is a node y
	with $R \in \mathcal{L}(x, y)$ and $C \notin \mathcal{L}(y)$
	then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\exists$-**Rule**	**If** $\exists R.C \in \mathcal{L}(x)$ and there is not any node x
	with $R \in \mathcal{L}(x, y)$ and $C \in \mathcal{L}(y)$
	then
	set $\mathcal{L}(x, y) = \mathcal{L}(x, y) \cup \{R\}$ and
	$\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$ where y is a new node

2.3 Tableau Optimization Techniques

We group the tableau optimization techniques into five categories focusing on different parts of description logic. These groups are: Preprocessing, Satisfiability, TBox, ABox, and Nominals. The list of these optimization techniques can be seen in Figure 2.1. The red star marks in the figure indicate the optimization techniques that are considered for enhancement in this thesis.

2.3.1 Preprocessing

In order to speed up the reasoning process in DL systems, input should be in a more appropriate form. Simplifying the input helps to find tautologies, i.e., avoid redundancy. Normalising helps to detect contradictions earlier by identifying syntactical equivalences. Moreover, one should avoid the use of general axioms, which is costly due to their non-deterministic status. Absorption builds primitive definition axioms from general axioms.

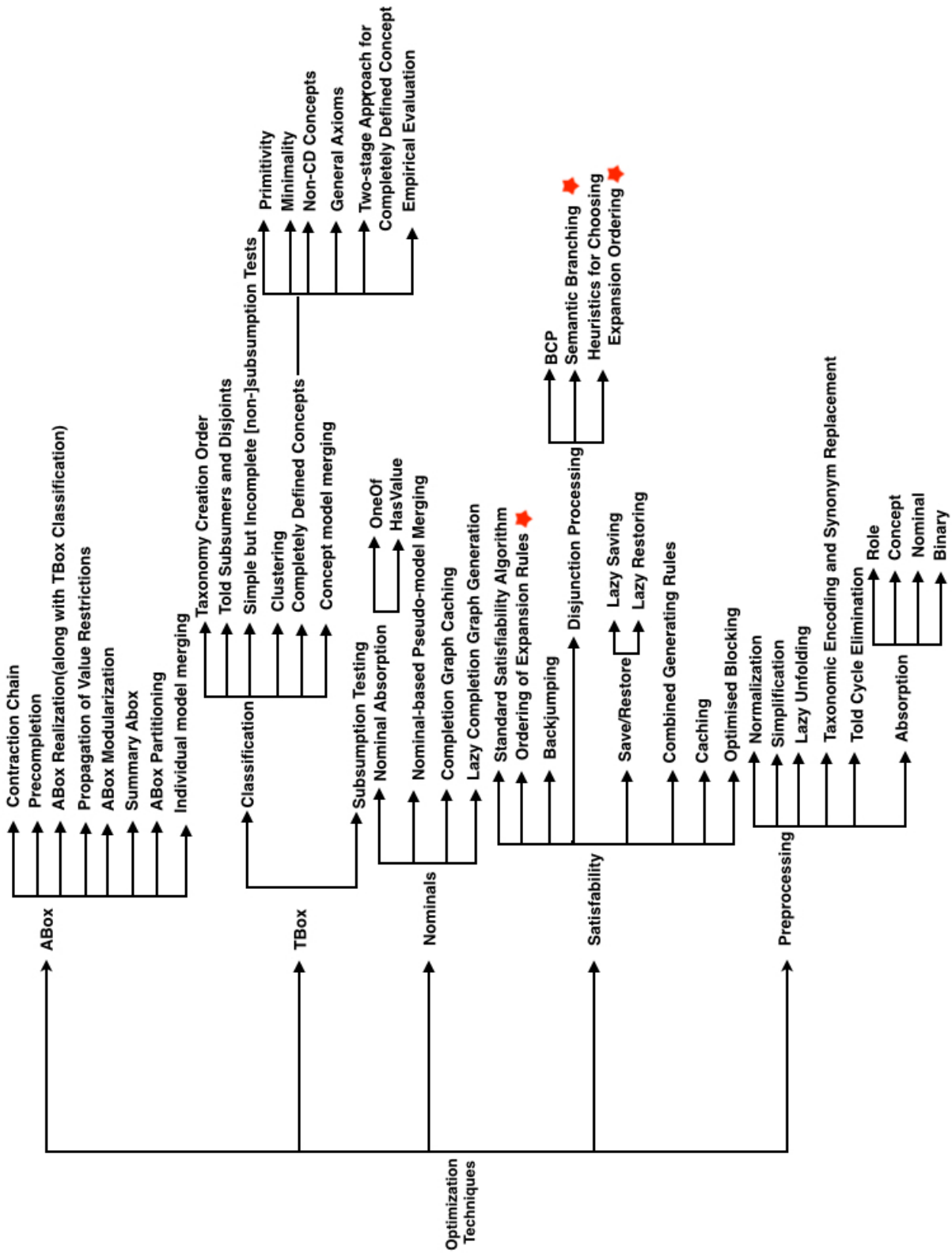

FIGURE 2.1: Tableau optimization techniques

Normalisation and Simplification

It may not always be constructive to assume the input of tableau algorithms to be in negation normal form since it could delay clash detection by ignoring obvious contradictions. For example, applying negation normal form to $(A \sqcap B) \sqcap \neg(A \sqcap B)$ gives $(A \sqcap B) \sqcap (\neg A \sqcup \neg B)$, which is not very efficient since it ignores the direct contradiction between the concept expressions $(A \sqcap B)$ and $\neg(A \sqcap B)$ in the first place. Therefore, with negation normal form, clashes can only be recognized on atomic concepts after applying costly methods such as backtracking. It is costly especially if concepts are large [20]. To avoid this, one may consider the syntactical form of all concepts (both atomic and expression) and try to detect their syntactically equivalent concepts [38].

To make it easier to find the equivalent concepts, one should normalise concept expressions. Nomalisation helps to detect the duplicates much easier. One of the most common ways to normalise the concepts is to use *DeMorgan's law*, which uses useful replacements of operators to generate more normalized expressions. For example, rules 4 and 6 in Table 2.3 respectively shows the replacement of $\sqcup$ and $\exists$ by negated $\sqcap$ and $\forall$.

TABLE 2.3: Normalisation function

Norm Function	Normalised
$Norm(A)$	A (A is an atomic concept name)
$Norm(\neg C)$	$Simp(\neg Norm(C))$
$Norm(C_1 \sqcap \cdots \sqcap C_n)$	$Simp(\sqcap\{Norm(C_1)\} \cup \cdots \cup \{Norm(C_n)\})$
$Norm(C_1 \sqcup \cdots \sqcup C_n)$	$Norm(\neg(\neg C_1 \sqcap \cdots \sqcap \neg C_n))$
$Norm(\forall R.C)$	$Simp(\forall R.Norm(C))$
$Norm(\exists R.C)$	$Norm(\neg\forall R.\neg(C)$

To complete the optimization process, these normalisation rules may also carry some simplification rules mentioned in Table 2.4.

TABLE 2.4: Simplification function

Simp Function	Simplified
$Simp(\neg C)$	$\begin{cases} \bot & if C \equiv \top \\ \top & if C \equiv \bot \\ Simp(D) & if C \equiv \neg D \\ \neg C & otherwise \end{cases}$
$Simp(\sqcap S)$	$\begin{cases} \bot & if \bot \in S \\ \bot & if \{C, \neg C\} \subseteq S \\ \top & if S = \emptyset \\ Simp(S \backslash \{\top\}) & if \top \in S \\ Simp(\sqcap\{P\} \sqcup S \backslash \{\sqcap\{P\}\}) & if \sqcap \{P\} \in S \\ \sqcap S & otherwise \end{cases}$
$Simp(\forall R.C)$	$\begin{cases} \top & if C = \top \\ \forall R.C & otherwise \end{cases}$

The above-mentioned techniques can be classified into the following groups: cheap and expensive. The cheap rules, e.g., $C \sqcap \top \equiv C$, can be easily applied to all of the TBox members and are always constructive but expensive rules, e.g., $A \sqcup B \equiv B$ if $A \sqsubseteq B$, may not always be helpful.

Lazy Unfolding

Lazy unfolding is one of the most powerful methods among all of the optimization techniques. It ignores axioms until it becomes necessary to unfold a concept during the subsumption and satisfiability testing.

Lazy unfolding is even more effective when the TBox is divided into two parts containing unfoldable and general axioms, $\mathcal{T}_u$ and $\mathcal{T}_g$, respectively; and then apply lazy unfolding to the unfoldable part. $\mathcal{T}_u$ and $\mathcal{T}_g$ together form the

main TBox, also they are disjoint from each other. Moreover, we can include another part called $\mathcal{T}_r$ containing axioms defining role domains and ranges. An axiom can be included in the unfoldable TBox if it is definitional, meaning that it has an atomic concept on its left side and the definition of that concept on the right side. For example, $A \sqsubseteq C$ defines the concept A and it is called primitive axiom. $A \equiv C$ is also definitional but non-primitive. The rest of the axioms are included in the general TBox part.

Tableau expansion rules shown in Table 2.5 are used to unfold concepts in an unfoldable TBox. Dividing a TBox into general and unfoldable parts helps

TABLE 2.5: Unfolding rules for an unfoldable TBox ($\mathcal{T}_u$)

Rule	If	Then
U_1-*rule*($\equiv$)	**1.**$A \in \mathcal{L}(x)$ and $(A \equiv C) \in \mathcal{T}_u$ **2.**$C \notin \mathcal{L}(x)$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{C\}$
U_2-*rule*($\equiv$)	**1.**$\neg A \in \mathcal{L}(x)$ and $(A \equiv C) \in \mathcal{T}_u$ **2.**$\neg C \notin \mathcal{L}(x)$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{\neg C\}$
U_3-*rule*($\sqsubseteq$)	**1.**$A \in \mathcal{L}(x)$ and $(A \sqsubseteq C) \in \mathcal{T}_u$ **2.**$C \notin \mathcal{L}(x)$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{C\}$

us to only apply internalisation to the general TBox. Internalisation is a simplification technique that is applied to the inclusion axioms of a TBox. The purpose of this simplification is to express all axioms of the TBox as a single concept called $C^{\mathcal{T}}$ by conjoining them. Therefore, it first assumes that every axiom in the TBox can be written as an inclusion axiom because as said before one can even write an equivalence axiom as two inclusion axioms, i.e., $C \equiv D$ as $C \sqsubseteq D$ and $D \sqsubseteq C$. Then, every inclusion axiom such as $C \sqsubseteq D$ will be converted to $\top \sqsubseteq \neg C \sqcup D$. Internalisation helps us dealing with general axioms by adding disjunctions to the TBox but it is not very efficient since many disjunctions will be added to all concept labels ($\mathcal{L}(x)$).

Table 2.6 shows the tableau expansion rules for general axioms. The axioms

TABLE 2.6: Tableau expansion rules for the general TBox ($\mathcal{T}_g$)

Rule	If	Then
G_1-*rule*($\equiv$)	**1.**$(C \equiv D) \in \mathcal{T}_g$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{(D \sqcup \neg C)\}$
	2.$(D \sqcup \neg C) \notin \mathcal{L}(x)$	
G_2-*rule*($\equiv$)	**1.**$(C \equiv D) \in \mathcal{T}_g$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{(\neg D \sqcup C)\}$
	2.$(\neg D \sqcup C) \notin \mathcal{L}(x)$	
G_3-*rule*($\sqsubseteq$)	**1.**$(C \sqsubseteq D) \in \mathcal{T}_g$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{(D \sqcup \neg C)\}$
	2.$(D \sqcup \neg C) \notin \mathcal{L}(x)$	

in $\mathcal{T}_r$, can be unfolded using tableau expansion rules shown in Table 2.7.

TABLE 2.7: Unfolding rule for the range TBox ($\mathcal{T}_r$)

Rule	If	Then
R-rule	**1.**$\top \sqsubseteq \forall S.C \in \mathcal{T}_r$	$\mathcal{L}(y) \rightarrow \mathcal{L}(y) \cup \{C\}$
	2.there is an $S-$neighbour y of x with $C \notin \mathcal{L}(y)$	

Absorption

As mentioned before, applying tableau expansion rules are costly for general axioms since they produce non-deterministic axioms. To avoid this, absorption takes advantage of lazy unfolding and tries to have as many axioms as possible in the unfoldable and domain/range parts, $\mathcal{T}_u$ and $\mathcal{T}_r$, respectively; so it maximizes the effectiveness of lazy unfolding [17]. In other words, it tries to absorb axioms from $\mathcal{T}_g$ into $\mathcal{T}_u$ or $\mathcal{T}_r$. In addition to concept and role absorption, nominal absorption may also be applied if there is any nominal in the axioms [44].

Before applying any absorption rule, one may normalise general axioms in $\mathcal{T}_g$ and consider their normalised form for absorption. For example, every axiom in the form of $A \sqsubseteq B$ will be converted to $\top \sqsubseteq (\neg A \sqcup B)$. This helps ignoring axioms such as $\top \sqsubseteq \top$ that is satisfiable or consider the whole TBox as unsatisfiable in the case, the axiom $\top \sqsubseteq \bot$ exit.

After normalising the TBox, the absorption rules for concepts, roles, and nominals can be applied as shown in Table 2.8.

TABLE 2.8: Absorption rules

Rules	Absorb from	Absorb into
Concept Absorption	$\top \sqsubseteq \neg A \sqcup C$ from $\mathcal{T}_g$	$A \sqsubseteq C$ into $\mathcal{T}_u$
Role Absorption	**1.** $\top \sqsubseteq (\forall R.C) \sqcup D$ from $\mathcal{T}_g$	$C \sqsubseteq \forall Inv(R).D$ into $\mathcal{T}_r$
	2. $\top \sqsubseteq (\leqslant nR.C) \sqcup D$ from $\mathcal{T}_g$	$C \sqsubseteq \forall Inv(R).((\leqslant nR.C) \sqcup D)$ into $\mathcal{T}_r$
Nominal Absorption	**1.** $\top \sqsubseteq \neg(o_1 \sqcup \ldots o_n) \sqcup D$ from $\mathcal{T}_g$	$o_i \sqsubseteq D$ into $\mathcal{T}_u$ $1 \leqslant i \leqslant n$
	2. $\top \sqsubseteq \neg(\exists R.(o_1 \sqcup \ldots o_n)) \sqcup D$ from $\mathcal{T}_g$	$o_i \sqsubseteq \forall Inv(R).D$ into $\mathcal{T}_u$ $1 \leqslant i \leqslant n$

Along with those absorption rules, some transformation rules may also need to be applied to some axioms in order to make the adoption of absorbing from $\mathcal{T}_g$ to $\mathcal{T}_u$ much easier. Also, told cycles that lead to non-termination should be removed. Told cycles can be direct or indirect [47]. In the case of direct, it happens when a definition axiom, e.g., $A \sqsubseteq C$ or $A \equiv C$ has the atomic concept A also added to the left-hand side of the axiom (concept expression C), i.e., $A \sqsubseteq C \sqcap A$ or $A \equiv C \sqcap A$. The above-mentioned rules are applied until no more rules can be applied to T_g or T_g is empty.

TABLE 2.9: Axiom transformation rules in $\mathcal{T}_g$

Convert from	into
$\top \sqsubseteq B \sqcup C$ where B is an atomic concept and $B \equiv D$ is in $\mathcal{T}_u$	$\top \sqsubseteq D \sqcup C$
$\top \sqsubseteq \neg B \sqcup C$ where B is an atomic concept and $B \equiv D$ is in $\mathcal{T}_u$	$\top \sqsubseteq \neg D \sqcup C$
$\top \sqsubseteq (D_1 \sqcap D_2) \sqcup C$	$\top \sqsubseteq D_1 \sqcup C$ and $\top \sqsubseteq D_2 \sqcup C$

2.3.2 Satisfiability

Testing the satisfiability of a concept in tableau-based systems might be costly due to non-deterministic expansions. Reasoners can avoid such cost by using some optimization techniques, which saves memory or time or both. Expansion-Ordering Heuristics dictate the order of disjunctions for expansion. A proper heuristic selection greatly reduces the size of the search space for reasoners. The Ordering of Expansion Rules is a very essential technique in the tableau algorithm, which decides what rules to expand first. Backjumping uses dependency lists to avoid redundant backtracking in completion graphs. Local Simplification and Semantic Branching both attempt to reduce the cost of non-determinism in tableaux. Caching is a way to store the results from traced branches in completion graphs and uses them in branches currently being explored, i.e., avoids repetitive computations.

Backjumping

To avoid redundant backtracking or thrashing in tableau-based systems, one can use the backjumping technique. In contrast to backtracking, backjumping does not just backtrack from a clash to the most recent choice point but also considers whether a choice point is related to a clash. This is achieved by creating a dependency set for each selected disjunct and labeling the concepts with sets containing the non-deterministic choices in each branch [47].

For instance, consider the following set of disjunctions:

$$\{p \sqcup q, s \sqcup t, \Phi_1 \sqcup \Psi_1, \ldots, \Phi_n \sqcup \Psi_n, \neg p \sqcup \neg s\}$$

As shown in the example of Figure 2.2, at least five initial disjunctions exist in the TBox that are labeled with numbers from 1 to 5. As indicated in Figure

2.2, there also may exist many states between state 3 and state 4. Each node added to the tableau tree will be labeled with a new number following the last assigned number. For example, by taking the first disjunct p from the first disjunction $p \vee q$, we label it with 6.

For each node, the numbers in square brackets refer to the dependency set for that disjunct. For example, disjunct p depends on the disjunction $p \vee q$; in other words, node 6 depends on the disjunction labeled with 1; therefore, its dependency set is $[1]$.

All disjunctions are expanded. If a clash is detected the expansion of the current branch stops. In this example, the first clash is detected in node 10 with a clash between $\neg p$ from node 10 and p from node 6. The branch is closed since a clash is detected. To decide which node to backtrack to, one chooses the most recent node that causes this clash considering that the disjunction should have an unexplored disjunct. From the two disjuncts of nodes 10 and 6 that cause this clash, we inspect their dependencies ([5] and [1]) in order to choose the backtracking point. Now as said, we find $5 : \neg p \vee \neg s$ as the most recent disjunction that has an unexplored branch. Afterward a new node 11 containing $\neg s$ is created. The dependency set for this label would be its parent and the closed branch, i.e., 5 and 6, respectively. Another clash is detected. This time the clash is between $\neg s$ in node 11 and s of node 7. Analyzing the dependencies of both nodes, we backtrack to the most recent dependency that has at least one unexplored branch. The node $5 : \neg p \vee \neg s$ is not considered since both of its branches have already been explored, so, node $2 : s \vee t$ is selected. This is a situation where backjumping becomes useful. Instead of backtracking over all disjunctions, i.e., 5,4,3,..., 2, we only process the dependency nodes that caused the clash, i.e., 5 and

2. Therefore, if we have n disjunctions between the disjunctions 3 and 4, this technique also helps us to bypass all of them during backjumping but normal backtracking needs to explore them all. This creates a possible search space of exponential size (2^n) compared to backjumping, which requires only linear space (n).

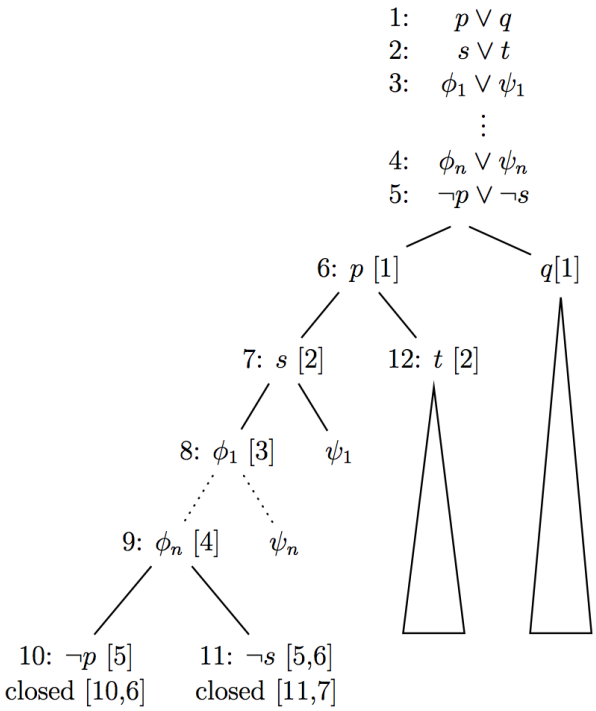

FIGURE 2.2: Backjumping (from Zhen[28])

Local Simplification

The local simplification technique is used to reduce the number of non-determinism caused by disjunctions. The local simplification also called Boolean Constraint Propagation is usually applied before expanding disjunctions and it tries to simplify them and avoid the cost of non-deterministic expansions [20].

Boolean Constraint Propagation (BCP) expands a disjunction with a disjunct while all other disjuncts have their negations in the TBox except for the selected disjunct. BCP applies the following rule to simplify the concept in $\mathcal{L}(x)$:

TABLE 2.10: Boolean Constraint Propagation (BCP)

$$\frac{\neg C_1, \ldots, \neg C_n, C_1 \sqcup \cdots \sqcup C_n \sqcup D}{D}$$

Semantic Branching

Traditional tableau algorithms use a not very efficient algorithm called syntactic branching. Consider the expansion of $(A \sqcup B_1) \sqcap (A \sqcup B_2)$. If we assume that A is unsatisfiable, i.e., it is equivalent to $\bot$, the expansion leads to four branches shown in Figure 2.3.

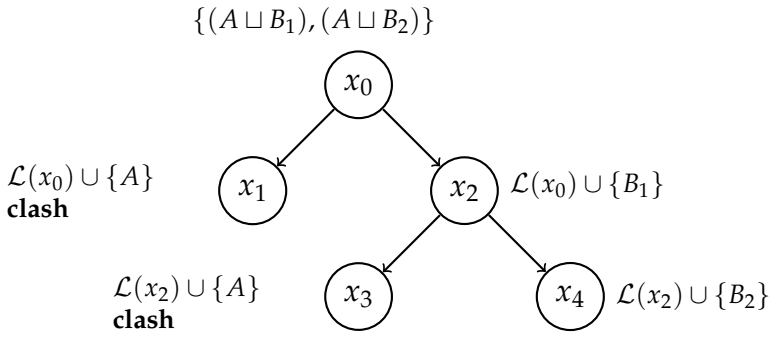

FIGURE 2.3: Syntactic branching assuming A is unsatisfiable (after Faddoul [11])

The first branch updates the status of state x_0 and adds A to it; that results in state x_1. Since A is assumed to be unsatisfiable, we encounter a clash in this branch and consider this branch as closed. Then, state x_0 is updated again to state x_2 where we create the next branch that chooses B_1; no clash

has been encountered yet; therefore, we move to the next expression that is again non-deterministic; now we choose A for the third branch. Since A is unsatisfiable, we have a clash in state x_3, i.e., it is also closed, and check the last branch for satisfiability/unsatisfiability. The fourth branch adds B_2 to state x_4 and no clash is seen here; therefore, $(A \sqcup B_1) \sqcap (A \sqcup B_2)$ is satisfiable.

In order to prove the satisfiability of a concept expression, we only need to know if one (unclosed) branch is satisfiable but in this example, we could not detect any satisfiable branch until we had explored all branches. In the case, where the initial concept expression is unsatisfiable, we need to show that all branches are unsatisfiable and have to explore all branches anyway.

As shown in the above example, the problem with syntactic branching is that two branches encountered the unsatisfiability of concept A. Due to the redundant search space that syntactic branching might explore, several reasoners apply semantic branching instead.

Semantic branching uses the Davis-Putnam-Logemann-Loveland (DPLL) algorithm that splits two branches by making them disjoint with each other [15]. DPLL has been developed to solve the SAT problem in propositional logic. In a tableau-based algorithm, while branching in the first level, instead of having A and B_1 respectively as the first and second branch, we would have A and $B_1 \sqcap \neg A$, respectively. This ensures the disjointness of the branches and no repetitive fails in different branches are encountered. For semantic branching, Figure 2.4 shows that there are only two explored branches in the tree and the unsatisfiability of A is only encountered once.

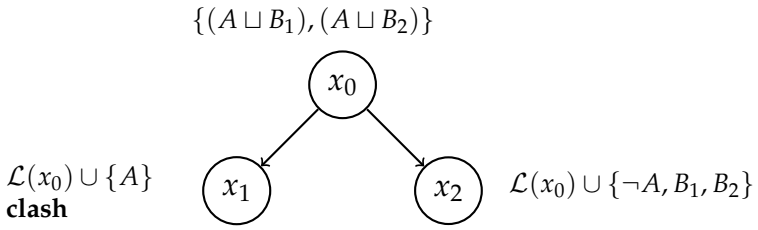

FIGURE 2.4: Semantic branching assuming A is unsatisfiable (after Faddoul [11])

Unit propagation in DPLL DPLL accepts boolean formulas as input that consist of a conjunction of clauses in which each clause consists of a disjunction of literals (Conjunctive Normal Form). It is assumed that an empty clause is unsatisfiable and an empty formula is satisfiable. There exist different versions of DPLL. It has also been extended for Satisfiability Modulo Theories (SMT) [7].

To prove that a boolean formula is satisfiable, a SAT solver should at least find one truth-value assignment for the variables of that formula, where a literal stands for a variable or its negation. DPLL incrementally builds the truth assignment for variables by selecting one variable each time. The procedure repeats until all the variables are assigned successively to satisfy the formula.

Unit propagation is an important part of the DPLL procedure. Unit propagation only applies to unit clauses. Since every clause in a boolean formula should be TRUE, to satisfy the entire formula, we first consider unit clauses, i.e., literals representing unit clauses need to be TRUE. If we have a literal A, we set A's variable to TRUE and for $\neg A$, we set A's variable to FALSE. Now, by applying the new truth value to the entire formula, a new formula is generated that may contain other unit clauses. Therefore, unit propagation is applied again until no more unit clauses exist in the formula. For example, in the formula

$$(A \sqcup B) \sqcap (\neg A \sqcup C) \sqcap (\neg C \sqcup D) \sqcap (A)$$

(A) is a unit clause containing the variable A; therefore, A should be TRUE. From this assignment, one concludes that the clauses A, $(A \sqcup B)$ can be removed from the formula and $\neg A$ can be eliminated from clause $(\neg A \sqcup C)$. The new generated formula would be $(C) \sqcap (\neg C \sqcup D)$ that again contains another unit clause (C). Therefore, another unit propagation is applied and the final formula is obtained as $(C) \sqcap (D) \sqcap (A)$.

Heuristics for Choosing Expansion-Ordering

The order of choosing disjuncts in disjunctions can greatly reduce the size of search space, i.e., the reasoning time for DL reasoners. Hence, several heuristics have been defined for this purpose based on the metrics of concepts involved in disjunctions. Those metrics are determined based on the frequency of the concepts occurring in an ontology, size of the concepts and depth of their quantifiers. The heuristics of expansion-ordering are explained in more detail in Section 4.3.2.

Ordering the Application of Expansion Rules

To decide which rules to expand first, DL reasoners introduce an approach using a ToDo list that controls the order of applying rules by giving them a priority [47]. The ToDo list is a queue containing as entries concepts and their nodes that are stored by some order.

The *ToDo list* Architecture Performing reasoning tasks can be costly in tableau-based systems due to non-deterministic expansions. To reduce space

and time costs, optimization techniques such as the ordering of applying expansion rules are used to decide what rules to expand first.

In the traditional top-down approach using the trace technique, the $\exists$-rule has the lowest priority; therefore, no unnecessary graph expansion happens until there are no more rules to apply except the $\exists$-rule. Additionally, the trace technique only considers the local trace and not a fully expanded tree since it removes the already traced branches to save space. Therefore, it is difficult to apply the trace technique in the presence of inverse roles. The reason is that not only the successors but also the predecessors will be involved and must be expanded recursively if needed, while the approach only keeps hold of local expansions. Moreover, if traditional tableau-based systems apply non-deterministic rules such as the disjunction rule ($\sqcup$) earlier than any other rules, the potential search space will increase.

To overcome the above restrictions, a ToDo list does not only store the entries in a particular order but also creates different queues based on the priority of different rules. Therefore, every time an entry is requested from the ToDo list, it is chosen from the queue with the highest priority. The process ends when all queues are empty or a clash is detected. If all queues have the same priority and only $\exists$ has the lowest priority; then the *ToDo list* technique works in the same way as the traditional top-down trace technique [46, 24].

In addition to the rules for the previously introduced $\mathcal{ALC}$ concept constructors, we consider rules for number restrictions ($\geqslant n\, R.C$, $\leqslant m\, R.D$ with $n, m \in \mathbb{N}$), which specify lower and upper bounds for role successors. The concepts $\exists R.C$ ($\forall R.C$) can be also expressed as $\geqslant 1\, R.C$ ($\leqslant 0\, R.\neg C$). The corresponding $\leqslant$-rule is considered as non-deterministic since it might require

the application of the non-deterministic choose-rule for number restrictions (see Baader et al. 1996 [3] for more details).

Caching

Compared to the trace technique, which saves space by closing the already traced branches, caching saves time/memory by reusing the satisfiability results of already traced branches for another node with similar concept expressions in the same tableau. In other words, it reduces the number of nodes (the size of models) by reusing other nodes (already built models).
When a node is fully expanded, i.e., no more rules can be applied; then, one could say that the successor node is independent of the previous node. Therefore, to prove the satisfiability /unsatisfiability of the successor node, we only need to check the concepts added to the successor since we are already aware of the satisfiability of the concepts from the previous node. Also, the added concept might have a satisfiability status in the cache and does not need to be checked for satisfiability again. Besides, this is one of the good reasons for delaying the expansion of the existential rule to the end since expanding it earlier may change the initial nodes a lot, and in this case, one cannot take the full advantages of the caching technique.
Caching uses hash tables to store the satisfiability/unsatisfiability of nodes. Therefore, before any expansion of a node, the satisfiability of the node will be checked in the cache. If the node has a satisfiability status, then it will not be checked again for satisfiability and the result will be reused.
Moreover, suppose that cache keeps the satisfiable and unsatisfiable sets in S and U, respectively; then, given a node called x containing all concepts of a

branch, we conclude:

- If $x \in S$ and $y \sqsubseteq x$, then y is also satisfiable.
- If $y \in U$ and $y \sqsubseteq x$, then x is also unsatisfiable.

2.3.3 TBox

TBox optimization involves building a hierarchy structure from its concepts. Classification allows reasoners to build a subsumption hierarchy of concepts that speeds up inferences. It also helps reasoners to answer subsumption queries. Therefore, optimization in classification could have a drastic effect on the reasoning procedure.

To classify a knowledge base, one needs to consider subsumption relations between named concepts in a TBox. The process of classification is iterative. The hierarchy structure is represented as a directed acyclic graph used to show the concepts and their relations. Nodes are concepts and edges represent a partial ordering between those concepts [4].

To classify a KB, first, the hierarchy structure is initialized with a node labeled with $\bot$ subsumed by a node labeled with $\top$. If KB is unsatisfiable; then, there is only one node in the hierarchy and it is labeled with both $\bot$ and $\top$. To find the position of a concept in the hierarchy, each time a concept is added for classification, we determine its parents (the nodes that subsume the concept) and children (the nodes that are subsumed by the concept). To carry this out, the top-down search and bottom-up search are applied respectively for parents and children of the added concept.

To find the position of C in the hierarchy according to its parents, we start from $\top$. If C is subsumed by D, we look into the children of D, if D does not

have any child that subsumes C, then D is considered as a parent of C. The same procedure is applied for finding the children of C this time by the help of the bottom-up search. During the process, if we find out that D is both parent and child of a concept C; then, C and D have the same position in the hierarchy since they are equivalent, i.e., $D \equiv C$

In this section, some of the techniques to reduce the number of subsumption tests while classifying the concepts into a taxonomy are introduced.

Taxonomy Creation Order

For creating the taxonomy of concepts, it is needed to test subsumptions between the concepts. The number of tests depends on the order that we allow concepts to be added to the taxonomy. Therefore, the best method is the one that applies the concepts in the order in which we need the least number of subsumptions tests.

One optimizing technique is to enter concepts in definition order, which means when there are two concepts C and D where C *uses*/*directly uses* D; then, D should be allowed in the taxonomy before C. C *directly uses* D when D occurs in the definition of C and *uses* stands for transitive closure of *directly uses*. Therefore, a concept will be classified after all of the concepts it uses are classified.

To reduce the number of subsumption tests, Baader et al. [5] suggested to also remove the bottom-up search tests when all of the TBox axioms are unfoldable; in other words, $\mathcal{T}_g = \emptyset$; because, for all primitive concepts of $\mathcal{T}_u$, their child is $\perp$. This only can happen when the general TBox axioms set ($\mathcal{T}_g$) is empty. For example, considering:

$$\mathcal{T}_u = \{C_1 \sqsubseteq D, C_2 \sqsubseteq D\} \text{ and } \mathcal{T}_g = \{\top \sqsubseteq C_1 \sqcap C_2\},$$

If we ignore the bottom-up search for the primitive concepts C_1 and C_2, we may ignore $C_1 \sqcap C_2$ having $\top$ as its child in $\mathcal{T}_g$. Besides, from $\mathcal{T}_u$, we know $C_1 \sqcup C_2 \sqsubseteq D$. Without considering the axioms in $\mathcal{T}_g$, we cannot conclude $\top \sqsubseteq C_1 \sqcup C_2 \sqsubseteq D$, i.e., $\top \sqsubseteq D$ but from both $\mathcal{T}_g$ and $\mathcal{T}_u$ since $C_1 \sqcap C_2 \sqsubseteq C_1 \sqcup C_2$ holds, we conclude: $\top \sqsubseteq C_1 \sqcap C_2 \sqsubseteq C_1 \sqcup C_2 \sqsubseteq D$, i.e., $\top \sqsubseteq C_1 \sqcup C_2 \sqsubseteq D$.
To further reduce the number of bottom-up searches, the above-mentioned approach is modified from definition order to quasi-definition order, which is the same as before except in this case, we shall ignore the relations between primitive concepts in the TBox if they are referenced by role quantifiers [18]. For example, from

$$C \sqsubseteq D, D \sqsubseteq \exists R.C$$

we only conclude $C \sqsubseteq D$ subsumption. This conclusion can be violated in some cases where there is an inverse role. Considering:

$$C \sqsubseteq D, D \sqsubseteq \exists R.\forall R^{-}C$$

$D \sqsubseteq C$ is concluded form the second axiom; therefore, we cannot ignore this subsumption, and from both axioms, we have $D \equiv C$.

Told Subsumers and Disjoint

Another way to reduce the number of subsumption tests for classifying the concepts is to take advantage of breadth-first traversal. To do this, one only needs to check whether C is subsumed by D when all of the concepts that subsume D also subsume C; otherwise, there is no need to test the subsumption between C and D. Moreover, one should make sure that all of the told

subsumers of a concept to be classified before the concept itself. A told subsumer of the concept C is D when $C \sqsubseteq D$.

Among all of the told subsumers of a concept, the most specific one is considered as a parent of that concept. To check if a subsumer is the most specific among all of the subsumers of the concept C, one may check the children of that subsumer, if none of the children of the subsumer subsume the concept, then the subsumer is the parent of C.

In addition to the top-down breadth-first traversal to find the parent of a concept, the bottom-up breadth-first traversal can also be used to find the children of a concept. Besides, the told subsumer can be used to propagate the subsumption. For example, If D is not a subsumer of C and A is a subsumer of D, then A is not subsumer of C either. Another term used in contrast with told subsumer is told disjoint that is used to determine the non-subsumption. C is told disjoint of D when $C \sqsubseteq \neg D \sqcap E$.

Clustering

While classifying a concept C, if $C \sqsubseteq D$, we shall also consider the children of D. Suppose D has children $D_1, \ldots, D_n$; to classify C, the children of D should also be checked for subsumption, i.e., $C \sqsubseteq D_i$ where $i = 1, \ldots, n$. To avoid checking all of the subsumptions of $C \sqsubseteq D_i$, a clustering technique is used. The technique labels the union of children of the concept D with a new name, e.g., $D_{ij} = D_i \sqcup D_{i+1} \cdots \sqcup D_j$. There could be different clusters of D's children. Then, instead of checking the subsumption for each of the children of D with C, we check the subsumption between C and the clusters. For example, if $C \not\sqsubseteq D_{ij}$ and $D_{ij} = D_i \sqcup \cdots \sqcup D_k \sqcup \cdots \sqcup D_j$ then it is obvious that $C \not\sqsubseteq D_k$.

Having several clusters of D's children, to determine the subsumption of one of the children, at least one of the clusters should be explored completely.

Completely Defined (CD) Concepts

Another method to reduce the number of subsumption tests is to define Completely Defined (CD) concepts. A primitive concept C in a TBox defined as $C = C_1 \sqcap \cdots \sqcap C_n$ is CD if the following holds:

- *Primitivity*: C_i is a primitive concept for $i = 1, \ldots, n$
- *Minimality*: For $i = 1, \ldots, n$, there is no two C_is that are told subsumers of each other.

CD concepts are the concepts that do not need any subsumption test if there are only CD concepts in the acyclic TBox. Concepts can easily be classified by definition order since they are completely defined by their told subsumers. Therefore, the structure of a TBox plays an important role to make a TBox with CD concepts much easier for classification. Some of the techniques to help the TBox taking advantage of CD concepts are:

- CD concepts without non-primitive concepts in their definition can easily be classified but when non-primitive concepts are not excluded from the right side of the primitive axiom they cannot be ignored. For example, from the TBox

  $$C \sqsubseteq C_1 \sqcap C_2 \sqcap C_3$$

  applying definition order, one can easily conclude that since C is a CD concept also C_1, C_2 and C_3 are primitive concepts; therefore, when C is

classified C_1, C_2 and C_3 are considered as the parents of C and the child of C is $\bot$.

Changing the TBox to:

$$C \sqsubseteq C_1 \sqcap C_2 \sqcap C_3, A \equiv C_1 \sqcap C_2$$

Since $C_1 \sqcap C_2$ is non-primitive, the CD classification approach cannot be applied as above. To avoid this situation, primitive synonyms should be considered. In this case, A is considered as a primitive synonym of $C_1 \sqcap C_2$. Therefore, first, C_1C_2 that stands for $C_1 \sqcap C_2$ is classified; then, A will be in the same position as C_1C_2 in the taxonomy.

- Checking the minimality of the concepts in the definition of CD concepts may be needed again after absorption, etc.

- CD classification can be extended to TBoxes having both CD and non-CD concepts. Most of the interesting TBoxes have both kinds of concepts. The best way to deal with such TBoxes is to separate CD and non-CD parts and first classify the CD concepts. For example, the TBox

$$C \sqsubseteq C_1 \sqcap C_2 \sqcap C_3, A \equiv C_1 \sqcap C_2$$

includes both CD and non-CD concepts. CD part has C, C_1, C_2, and C_3 concepts; so, the only concept in non-CD part is: A. To classify, first, we consider the CD part. After classification, C has parents C_1, C_2 and C_3. Then, we classify the non-CD part in which we conclude that A has parents C_1 and C_2 and child C.

- Sometimes it is not possible to apply CD classification in the presence of general axioms; therefore, one should try to empty $\mathcal{T}_g$. For example, in the TBox

$$\mathcal{T}_u = \{C \sqsubseteq \top, A \sqsubseteq D, B \sqsubseteq D\}, \mathcal{T}_g = \{\top \sqsubseteq A \sqcup B\}$$

  applying CD classification, C will have $\top$ as its parent but when ignoring $\mathcal{T}_g$, we cannot conclude that A, B, and C will be the children of D and D is equivalent to $\top$. Therefore, CD classification is only applied when $\mathcal{T}_g$ is empty.

Concept Model Merging

Pseudo-model merging is a possible way of speeding up TBox reasoning. It is a non-subsumption test between concepts. The same algorithm also applies to ABox reasoning, which will be explained briefly in Section 2.3.4. The algorithm is proposed for $\mathcal{ALCNH}_{\mathcal{R}^+}$ [18].

Instead of doing the consistency test using costly algorithms such as tableau-based, it caches the pseudo-models of the involved concepts and finds the interactions between their completions. If the pseudo-models can be merged then the subsumption does not hold.

For example, having $(A \sqcup B) \sqcap C$ as a concept description, we want to cache its pseudo-model. We would have two options since it has two completions $A \sqcap C$ and $B \sqcap C$ [49].

The pseudo-model for each concept is composed of a tuple $< M^A, M^{\neg A}, M^{\exists}, M^{\forall} >$.

$$M^A = \{A|a : A \in \mathcal{A}'\}$$

$$M^{\neg}A = \{A|a : \neg A \in \mathcal{A}'\}$$
$$M^{\exists} = \{R|(\exists R.C)(a) \in \mathcal{A}'\}$$
$$M^{\forall} = \{R|(\forall R.C)(a) \in \mathcal{A}'\}$$

For every two concepts, we check whether there is any interaction between their models. One should compare two concepts with tuples $< M_1^A, M_1^{\neg A}, M_1^{\exists}, M_1^{\forall} >$ and $< M_2^A, M_2^{\neg A}, M_2^{\exists}, M_2^{\forall} >$ in such a way that if the following counterparts can be merged: M_1^A with $M_2^{\neg A}$, $M_1^{\neg A}$ with M_2^A, $M_1^{\exists}$ with $M_2^{\forall}$ and $M_1^{\forall}$ with $M_2^{\exists}$.

Pseudo-model merging is sound but not complete. For example, consider we want to check if $A \sqsubseteq A \sqcap B$ holds. To check this, we may need to have an axiom with conjunction since pseudo-model merging is about to find out if pseudo-models of conjunction parts can be merged. To do that, instead of checking $A \sqsubseteq A \sqcap B$, we check its negative equivalence, which is $A \sqcap \neg(A \sqcap B)$. Now, we try to find out the pseudo-model of conjuncts A and $\neg(A \sqcap B)$. A's pseudo-model is $M_1 =< A, \emptyset, \emptyset, \emptyset >$ but $\neg(A \sqcap B)$ or simply $\neg A \sqcup \neg B$ has two possible pseudo-models $M_{2a} =< \emptyset, A, \emptyset, \emptyset >$ or $M_{2b} =< \emptyset, B, \emptyset, \emptyset >$. There is an interaction between M_1^A and $M_2^{\neg A}$ and they cannot be merged but there is not any interaction between M_1^A and $M_2^{\neg B}$ and they can be merged. Considering both, we finally conclude that the two models M_1 and M_2 can be merged. This shows the fact that if even two models have interactions with each other, we cannot say for sure if they are non-mergeable and we should look for the next possible path that leads to the models interacting with each other while building the pseudo-models.

Now, consider another example where we want to check if $\forall R.C \sqsubseteq \forall R.D$ holds. Therefore, we should check if $\forall R.C \sqcap \neg\forall R.D$ or simply $\forall R.C \sqcap \exists R.\neg D$ does not hold. The model for $\forall R.C$ is $M_1 =< \emptyset, \emptyset, \emptyset, R >$ and the model

for $\exists R.\neg D$ is $M_2 =< \emptyset, \emptyset, R, \emptyset >$. There is an interaction between $M_1^{\forall}$ and $M_2^{\exists}$ since they both have R. This is where deep merging should be applied. Until now, we only saw the effect of flat merging. Deep merging also looks at the concepts that roles had relations with; so, for $M_1^{\forall}$, we should look for the pseudo-model of C and for $M_2^{\exists}$, we look for the pseudo-model of $\neg D$. The pseudo-models of C and $\neg D$ do not have any interaction and they are mergeable. Deep merging is only applied for concept merging and not individual merging.

2.3.4 ABox

The size of ABoxes is often larger than TBoxes due to the many concept and role assertions they contain. Therefore, optimizing ABoxes is very effective for reasoning as much as, if not more than, TBoxes. In this section, two ABox optimization techniques are explained. Both techniques try to reduce the size of large ABoxes to make reasoning much easier and faster.

ABox Modularization

ABox modularization is a way of reducing instance checking by considering a small subset of relevant assertions instead of the whole ABox [48]. Theretofore, modularization helps us to save both memory and time.

ABox modularization is mainly designed for $\mathcal{SHI}$ that is a semi-expressive description language. We define an ontology $\mathcal{O}$ as <$\mathcal{T}$,$\mathcal{R}$,$\mathcal{A}$>. Ontology $\mathcal{O}$ is considered to be consistent meaning there is an interpretation $\mathcal{I}$ that $\mathcal{I} \models \mathcal{O}$. ABox modularization M is composed of set of ABoxes $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ in which $\mathcal{A}_i$ is an ABox module. For both concept and role assertions, we can say

- M entails a concept assertion $C(a)$ if $\exists \mathcal{A}_i \in M. < \mathcal{T}, \mathcal{R}, \mathcal{A}_i > \models C(a)$

and

- M entails a role assertion $R(a_1, a_2)$ if $\exists \mathcal{A}_i \in M. < \mathcal{T}, \mathcal{R}, \mathcal{A}_i > \models R(a_1, a_2)$

One always tries to generate an ABox modularization that is both sound and complete. Considering ontology O as <$\mathcal{T}$,$\mathcal{R}$,$\mathcal{A}$>, M is sound when it only derives assertions entailed by $\mathcal{A}$ and M is complete when it derives every assertion entailed by $\mathcal{A}$.

Therefore, choosing proper modules is very important when it comes to reasoning. For an example of ABox modularization, suppose ontology $O_a =< \mathcal{T}_a, \mathcal{R}_a, \mathcal{A}_a >$, which is defined below:

$$\mathcal{T}_a = \{\text{Chair} \equiv \exists \text{headOf.Department}\},$$

$$\mathcal{R}_a = \{\text{headOf} \sqsubseteq \text{memberOf}\},$$

$$\begin{aligned} \mathcal{A}_a = \{&\text{Department(ee)}, \text{Professor(mae)},\\ &\text{UndergraduateCourse(c4)}, \text{UndergraduateCourse(c5)},\\ &\text{Student(sam)}, \text{Student(sue)}, \text{Student(zoe)}, \text{headOf(mae, ee)},\\ &\text{teaches(mae, c4)}, \text{teaches(mae, c5)},\\ &\text{takes(sam, c4)}, \text{takes(sue, c5)}, \text{takes(zoe, c5)}\} \end{aligned} \tag{2.1}$$

By generating two modules from $\mathcal{A}_a$, we have:

$$\mathcal{A}_{a1} = \{\text{Department(ee)}, \text{headOf(mae, ee)}, \text{Professor(mae)}\}$$

$$\begin{aligned}\mathcal{A}_{a2} = \{&\text{UndergraduateCourse}(\text{c4}), \text{UndergraduateCourse}(\text{c5}),\\ &\text{UndergraduateCourse}(\text{c4}), \text{UndergraduateCourse}(\text{c5}),\\ &\text{Student}(\text{sam}), \text{Student}(\text{sue}), \text{Student}(\text{zoe}),\\ &\text{teaches}(\text{mae}, \text{c4}), \text{teaches}(\text{mae}, \text{c5}),\\ &\text{takes}(\text{sam}, \text{c4}), \text{takes}(\text{sue}, \text{c5}), \text{takes}(\text{zoe}, \text{c5})\}\end{aligned} \tag{2.2}$$

From $\mathcal{A}_{a1}$, we conclude that *mae* is an instance of Chair since based on headOf (mae, ee) and $\text{Department}(\text{ee})$ in $\mathcal{A}_{a1}$ and $\text{Chair} \equiv \exists \text{headOf.Department}$ in $\mathcal{T}_a$, it is obvious that ee is an instance of Department, i.e., mae is an instance of Chair.

As seen from above example, we still conclude the same assertions even after modularization but by considering the following modules from $\mathcal{A}_a$:

$$\begin{aligned}\mathcal{A}_{a3} = \{&\text{Department}(\text{ee}), \text{UndergraduateCourse}(\text{c4}),\\ &\text{UndergraduateCourse}(\text{c5})\end{aligned}$$

$$\begin{aligned}\mathcal{A}_{a4} = \{&\text{Professor}(\text{mae}), \text{Student}(\text{sam}), \text{Student}(\text{sue}), \text{Student}(\text{zoe}),\\ &\text{headOf}(\text{mae}, \text{ee}), \text{teaches}(\text{mae}, \text{c4}), \text{teaches}(\text{mae}, \text{c5}),\\ &\text{takes}(\text{sam}, \text{c4}), \text{takes}(\text{sue}, \text{c5}), \text{takes}(\text{zoe}, \text{c5})\}\end{aligned} \tag{2.3}$$

We cannot conclude that mae is an instance of Chair since we cannot conclude solely from $\mathcal{A}_{a4}$ that ee is an instance of Department.

The ABox modularization can be divided into component-based modularization and intentional-based modularization.

Component-based modularization only looks at the assertional part of the ontology to decide how to break the ABox into modules. It maps the ABox to a graph in which nodes represent individuals asserted as instances of concepts and edges represent relationships asserted between individuals. Such modularization has already been seen in the previous examples.

If there exist many role assertions in an ABox, component-based modularization generates a small number of large ABoxes that is not very efficient. One can overcome this issue by splitting role assertions. Intentional-based modularization analyses the TBox of an ontology in order to find out how to break role assertions. For example, consider an ontology $\mathcal{O}_b$ as $<\mathcal{T}_b,\mathcal{R}_b,\mathcal{A}_b>$ with:

$$\mathcal{T}_b = \{\top \sqsubseteq \forall \text{takes.Course}\},$$

$$\mathcal{R}_b = \{\},$$

$$\mathcal{A}_b = \{\text{Course}(\text{c5}), \text{Student}(\text{zoe}), \text{takes}(\text{zoe}, \text{c5}), \text{teaches}(\text{mae}, \text{c5})\}$$

From $\mathcal{T}_b$, we already know takes propagates Course; also since Course(c5) is an element of $\mathcal{A}_b$, we know that c5 is an instance of Course, the role takes(zoe, c5) can be split. Moreover, the role teaches(mae, c5) is useless and can be removed since it does not propagate anything. To know exactly which roles should be split, we might use a $\forall$–info structure [48]. It helps us to determine the concept descriptions that are propagated over role descriptions. Before applying $\forall$–info structure, a TBox should be in normal form, i.e., all its GCI axioms should be in form $\top \sqsubseteq C$ in which C is in negation normal form.

Then, ∀–info structure contains subconcept descriptions of all ∀–concept descriptions from the closure of the TBox. For example, the TBox:

$$\begin{aligned}\mathcal{T}_c = \{&\top \sqsubseteq \forall \text{takes.Course}, \exists \text{takes.Course} \sqsubseteq \text{Student},\\ &\exists \text{memberOf}.\top \sqsubseteq \text{Person}, \text{GraduateStudent} \sqsubseteq \text{Student},\\ &\text{UndergraduateStudent} \sqsubseteq \text{Student}\}\end{aligned} \tag{2.4}$$

Converting $\mathcal{T}_c$ to the normal form:

$$\begin{aligned}\mathcal{T}_c = \{&\top \sqsubseteq \forall \text{takes.Course}, \top \sqsubseteq \forall \text{takes.}\neg\text{Course} \sqcup \text{Student},\\ &\top \sqsubseteq \forall \text{memberOf}.\bot \sqcup \text{Person}, \top \sqsubseteq \neg\text{GraduateStudent} \sqcup \text{Student},\\ &\top \sqsubseteq \neg\text{UndergraduateStudent} \sqcup \text{Student}\}\end{aligned} \tag{2.5}$$

The ∀–info structure would be:

$$info^{\forall}_{\mathcal{T}_c}(R) = \begin{cases} \{\text{Course}, \neg\text{Course}\} & if R = \text{takes} \\ \{\bot\} & if R = \text{memberOf} \\ \emptyset & \text{otherwise} \end{cases}$$

After identifying the concept descriptions that are propagated via role descriptions, we should split the target roles.

We define an ABox split function in such a way that if there is not any role assertion between two individuals a and b, we do not have any role to split; therefore, the ABox is the same before and after the split. The other case is where there is a role assertion between two individuals ann and c_1; shown in Figure 2.5. In this case, we need to consider two distinct anonymous individuals ann^* and ${c_1}^*$ and try to split the role assertion into two role assertions. The two new role assertions would be $\text{teaches}(\text{ann}, {c_1}^*)$ and $\text{teaches}(\text{ann}^*, c_1)$.

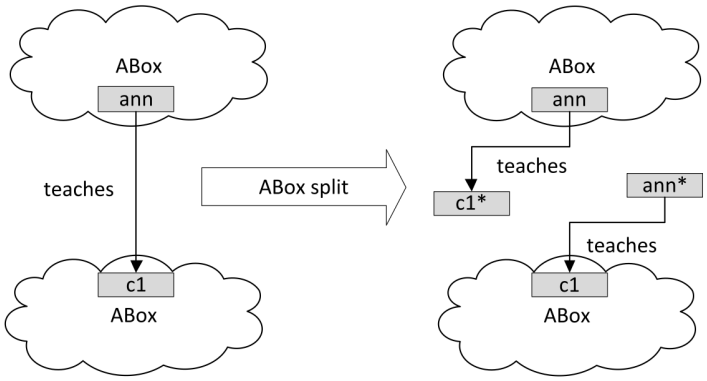

FIGURE 2.5: ABox splitting (from Fokoue et al. [48])

Now, the component-based modularization is applied as described before.

ABox Summary

Another ABox optimization method, which is introduced in [12], is ABox summary. It tries to summarize ABoxes representing databases, i.e, reduces the reasoning process for those ABoxes. This technique is proposed for $\mathcal{SHI}$. Many ABoxes have redundant concept assertions. As seen from Figure 2.6, the original ABox is summarized to the ABox on the right hand side such a way that all of its individuals that have the same relationships are considered to be the same node (aggregating similar individuals in the same concept) and the ones that are not the same (c_1 and c_2) remain in their nodes.
Further, the ABox summary technique also applies some filtering techniques to reduce the summarized ABoxes that grantee to be scalable for very large ABoxes. More information on this technique is given in [12].

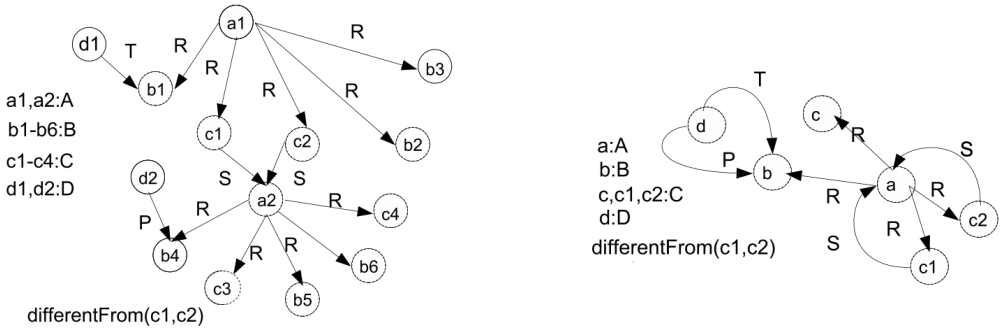

FIGURE 2.6: ABox summary
node a represents a_1 and a_2, node b represents b_1 to b_6, node d represents d_1 and d_2 and node c represents c_3 and c_4 (from Wandelt et al. [12])

Individual Model Merging

Pseudo-model merging is a method that avoids costly inconsistency tests. To check whether an individual is not an instance of a concept, it checks if the pseudo-model of that individual can be merged with the pseudo-model of the negation of that concept. For example, if one wants to check if $a : C$ is true; it should check if $A \sqcup \{a : \neg C\}$ is not satisfiable. Here, A stands for an ABox. Then, it checks for pseudo-models of individual a and $\neg C$ and sees if they are mergeable. By pseudo-model of a, we mean the pseudo-models of the concepts that a belongs to in the ABox. The same formulas for finding pseudo-models as discussed in 2.3.3 can be applied for this case too.
When two models have a clash between them that does not necessarily mean they cannot be merged since their dependency sets can say if the clash is deterministic or not meaning that a completion graph could have chosen another way to bypass the clash. In other words, pseudo-model merging is sound but incomplete as already discussed in Section 2.3.3.

2.3.5 Nominals

Nominals can be seen as concepts with only one instance. The constructor which allows the definition of a concept containing nominals is called OneOf. For example, the concept Continent is defined as: Continent $=$ $\{$europe, asia, america, antartica, africa, oceania$\}$ in which europe, asia, america, antartica, africa and oceania are the nominals of the concept Continent. Besides, another constructor of nominals called hasValue defines an existential restriction over nominals of a concept. For example, Catholic $\sqsubseteq$ Person $\sqcap$ $\exists$follows.pope defines Catholic as a Person who follows pope where pope is a nominal [44].

Many reasoners ignore the presence of nominals while optimizing ABoxes and TBoxes. However, with nominals in a TBox, classification, and satisfiability of concepts can be affected by ABox assertions. Moreover, ontologies with numerous GCI axioms related to nominals cannot be handled with former absorption techniques. In the following, two nominal optimization techniques are reviewed: Nominal Absorption, which is related to absorbing GCI axioms including nominals and nominal-based pseudo-model merging, are used for non-subsumption detection of concepts that deal with nominals.

Nominal Absorption

It is very important to apply the absorption techniques for nominals since GCI axioms are non-deterministic and may consume a large amount of space. Since the former absorption techniques cannot be applied to nominals, in

the following, nominal absorption for both cases of nominals OneOf and hasValue are presented [44]:

- OneOf: Suppose,

  $$\text{WineColor} \equiv \{\text{red}, \text{rose}, \text{white}\} \text{ and } \text{WineColor} \sqsubseteq \text{WineDescriptor}$$

  Both of the above deal with GCI axioms and cannot be absorbed using the former absorption techniques. Therefore, we add $\neg\text{WineColor} \sqcup \{\text{red}, \text{rose}, \text{white}\}$ to all of the nodes in tableau; considering $\{\text{red}, \text{rose}, \text{white}\} = \{\text{red}\} \sqcup \{\text{rose}\} \sqcup \{\text{white}\}$. Adding this disjunction to the tableau is very expensive and causes non-determinism. To avoid this issue, the following nominal absorption technique is used: The inclusion axiom $\text{WineColor} \equiv \{\text{red}, \text{rose}, \text{white}\}$ can be absorbed into a primitive definition

  $$\text{WineColor} \sqsubseteq \{\text{red}, \text{rose}, \text{white}\}$$

  and ABox assertions

  $$\text{WineColor}(\text{red}); \text{WineColor}(\text{rose}); \text{WineColor}(\text{white})$$

  , which is equal to GCI $\{\text{red}, \text{rose}, \text{white}\} \sqsubseteq \text{WineColor}$
  Now, with this absorption, the disjunction $\{\text{red}, \text{rose}, \text{white}\}$ is not very expensive since it is only applied to WineColor individuals in the tableau and not every individual. Therefore, absorption helped us to localize the cost of disjunctions to a small number of nodes instead of every node.

- hasValue: Consider,

$$\text{Riesling} \equiv \text{Wine} \sqcap \leq 1\text{madeFrom} \sqcap \exists \text{madeFrom}.\{\text{RieslingGrape}\}$$

If we consider a GCI from above axiom:

$$\text{Wine} \sqcap \leq 1\text{madeFrom} \sqcap \exists \text{madeFrom}.\{\text{RieslingGrape}\} \sqsubseteq \text{Riesling}$$

i.e.,

$$\top \sqsubseteq \neg\text{Wine} \sqcup \geq 2\text{madeFrom} \sqcup \forall \text{madeFrom}.\neg\{\text{RieslingGrape}\} \sqcup \text{Riesling} \tag{2.6}$$

So, Wine can be defined as

$$\text{Wine} \sqsubseteq \geq 2\text{madeFrom} \sqcup \forall \text{madeFrom}.\neg\{\text{RieslingGrape}\} \sqcup \text{Riesling}.$$

Adding the definition of Wine to every concept containing Wine causes more non-determinism that should be avoided. To avoid this issue, the following nominal absorption technique is used:
In this technique, GCI

$$\exists p.\{o\} \sqsubseteq C$$

is absorbed to

$$\{o\} \sqsubseteq \forall p^{-}.C$$

as shown in Figure 2.7 in which $\{o\} \sqsubseteq C$ is equal to $C(o)$.
Therefore, in our example, we absorb:

$$\exists \text{madeFrom}.\{\text{RieslingGrape}\} \sqsubseteq \neg\text{Wine} \sqcup \geq 2\text{madeFrom} \sqcup \text{Riesling}$$

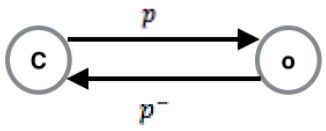

FIGURE 2.7: Nominal absorption graph

into

$$\{\text{RieslingGrape}\} \sqsubseteq \forall \text{madeFrom}^{-}.\neg\text{Wine} \sqcup \geq 2\text{madeFrom} \sqcup \text{Riesling}$$

Since $\{o\} \sqsubseteq C$ is equal to $C(o)$, the above can be written as:

$$(\forall \text{madeFrom}^{-}.\neg\text{Wine} \sqcup \geq 2\text{madeFrom} \sqcup \text{Riesling})(\{\text{RieslingGrape}\})$$

Compared to the former absorption applied to the definition of Wine, this time the number of disjunctions are the same but they are localized to the individual RieslingGrape that is smaller than the number of individuals of Wine.

Nominal-based Pseudo-model Merging

The pseudo-model merging optimization technique for classification can also take advantage of the presence of nominals in order to detect non-subsumption between concepts. The technique is useful only if there are concepts with existential restriction on nominals in a TBox (hasValue), e.g., RedWine $\sqsubseteq$ Wine $\sqcap \exists$hasColor.$\{$red$\}$. To check the non-subsumption relation between two concepts, the pseudo-model merging technique checks for the information gained from the completion graph of the first concept that is used for its satisfiability test that involved its nominals. The information gained from the completion graph of the first concept is used to check the

non-subsumption relation between two concepts, i.e., by using that info the number of satisfiability tests can usually be reduced. For example, consider the two concepts' RedWine and ItalianWine:

$$\text{RedWine} \sqsubseteq \text{Wine} \sqcap \exists \text{hasColor}.\{\text{red}\}$$

and

$$\text{ItalianWine} \sqsubseteq \text{Wine} \sqcap \exists \text{producedIn}.\{\text{italy}\}$$

We want to check the non-subsumption relation between RedWine and ItalianWine. To do that, we check if $\text{ItalianWine} \sqcap \neg\text{RedWine}$ holds; therefore, we extract the pseudo-models for both concepts ItalianWine and $\neg$RedWine, i.e., the pseudo-models for $\text{Wine} \sqcap \exists \text{producedIn}.\{\text{italy}\}$ and $\neg(\text{Wine} \sqcap \exists \text{hasColor}.\{\text{red}\})$.
The pseudo-models of both concepts can be merged since we can have models that do not interact with each other. Therefore, we conclude that the subsumption $\text{ItalianWine} \sqsubseteq \text{RedWine}$ does not hold [44].

2.4 Machine Learning

Machine learning is a branch of artificial intelligence that uses a trained model for predicting unseen data without manual intervention. Machine learning techniques can be categorized into four groups:

- Supervised learning in which the training data is provided with labels. The trained model is used to predict the labels for unseen data. This can be seen as a classification problem in which the data is classified based on the labels that are given.

- Unsupervised learning in which the training data is not provided with any labels and the model is built to find meaningful patterns in the data without knowing any labels. This can be seen as a clustering problem in which the data is clustered based on the found patterns.

- Semi-supervised learning is used to train the data with both labeled and unlabeled data provided. Obtaining labeled data could be costly depending on the problem; in this case, one can take advantage of different semi-supervised learning techniques to make use of a mix of unlabeled and labeled data for their training data.

- Reinforcement learning is used in order to gain the maximum reward based on the agent's interaction with the environment. The model learns to take any action that leads to the maximum reward.

In this section, first, we focus on two supervised machine learning techniques called logistic regression and SVM. Later, we focus on feature engineering techniques that also includes an unsupervised learning technique called Principal Component Analysis (PCA).

2.4.1 Logistic Regression

Logistic Regression is a supervised learning technique, which predicts a label (class)[1] for each input data.

In comparison with linear regression, which predicts a continuous dependent variable for each independent variable (instance), logistic regression

[1]For the rest of this chapter, the words "label" and "class" are used interchangeably.

predicts a categorical dependent variable (class) for each independent variable.

The model built for logistic regression considers the dependent variable as a conditional probability given as $P(Y = 1|X = x)$ that indicates the probability of choosing a specific class (identified as 1) for an instance x. If the resulting probability is more than 0.5, then the instance belongs to class 1; otherwise, it belongs to class 0 [42].

The represented model for linear regression is a linear equation that relates independent variables ($\mathbf{x}$ with n features $x_1, x_2, ..., x_n$) to dependent variables ($h(\mathbf{x})$):

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{n} w_i x_i \tag{2.7}$$

x_0 is equal to 1 and $\mathbf{w}$ stands for parameters or weights.
Using the same model for logistic regression leads to:

$$P(Y = 1|\mathbf{x}) = \mathbf{w}^T\mathbf{x} \tag{2.8}$$

that is not promising, since the probability should be bounded between 0 and 1. Therefore, the modified equation represented for logistic regression is:

$$P(Y = 1|\mathbf{x}) = \frac{e^{\mathbf{w}^T\mathbf{x}}}{1 + e^{\mathbf{w}^T\mathbf{x}}} \tag{2.9}$$

Multinomial Logistic Regression

If the number of classes to predict is more than two, then the logistic regression technique is called multinominal logistic regression and it is defined by:

$$P(Y = H_j|\mathbf{x}) = \frac{e^{\mathbf{w}_{H_j}^T \mathbf{x}}}{\sum_{H'} e^{\mathbf{w}_{H'}^T \mathbf{x}}} \tag{2.10}$$

where $\sum_H P(H|\mathbf{x}) = 1$, $H = H_1, H_2, ..., H_j, .., H_k$. For all k classes, the probability of an instance labeled as each of those classes will be obtained; finally, among the obtained probabilities the maximum one defines the class that the instance belongs to.

To obtain the best value for parameters ($\mathbf{w}$), the *Newton-Raphson* method is used to maximize the approximation of the predictor function. To obtain the best $\mathbf{w}$, *Newton-Raphson* uses *Hessian* and *Gradient* that are the second and first derivative of the error function ($J(\mathbf{w})$), respectively. The following formula is repeated several times to obtain the best $\mathbf{w}$ ($\mathbf{w}^0$ is initiated with a vector of 0s):

$$\mathbf{w}^{i+1} = \mathbf{w}^i - \frac{J'(\mathbf{w}^i)}{J''(\mathbf{w}^i)} \tag{2.11}$$

The training data pairs (x_i, y_i) are ready to be used for the *Newton-Raphson* method. x_i is an input point in training data and y_i is the associated label to this input. While computing $\mathbf{w}_{H_j}$, the training instances with class H_j will be identified as class 1 and others are 0.

The error function ($J(\mathbf{w})$) in logistic regression is defined as:

$$J(\mathbf{w}) = -\sum_{i=1}^{m} y_i log(h(x_i)) + (1 - y_i) log(1 - h(x_i)) \tag{2.12}$$

To avoid overfitting, the regularized term $\lambda \sum_{i=1}^{m} w_i^2$ is added to the cost function where λ is a regularization parameter (*Hessian* and *Gradient* formulas will also be changed accordingly).

2.4.2 Support Vector Machine (SVM)

SVM is a supervised classifier technique, which is best used for binary classification. SVM creates a decision boundary called hyperplane in order to separate data points (training data) into two different classes. Each data point belongs to one of the classes 1 or 2.

As an example, Figure 2.8 shows data points from two different classes 1 and 2 represented as × and ● ; respectively. There is an infinite number of hyperplanes that separate two classes; however, the objective is to find an optimal hyperplane that creates a maximum distance between the two closest data points from two classes. The optimal hyperplane in Figure 2.8 is shown in bold. The margin is the location between the hyperplane and each data point shown in Figure 2.9. If the margin is small enough to separate the data points perfectly, it is called hard margin (Figure 2.9). However, there is a possibility that a model with 100% accurate prediction cannot be generalized for unseen data. This issue is called overfitting of the model. To avoid overfitting, the margin should be softened (widened) to allow some misclassification as shown in Figure 2.10. This helps to create a more generalizable model.

SVM can handle the data points that are not linearly separable by using a non-linear hyperplane. In this case, the non-linear kernel functions are used to transform the original data point to a higher dimensional feature space to make them linearly separable in the new space as shown in Figure 2.11. Possible kernels are RBF (Radial Basis Function) and Polynomial [8].

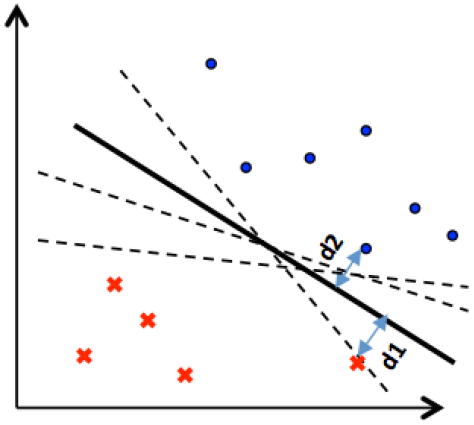

FIGURE 2.8: Optimal hyperplane for separating data points

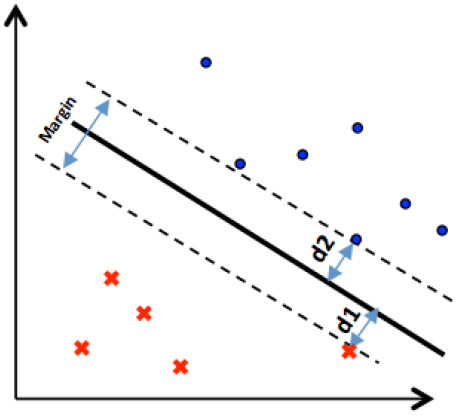

FIGURE 2.9: Hard margin

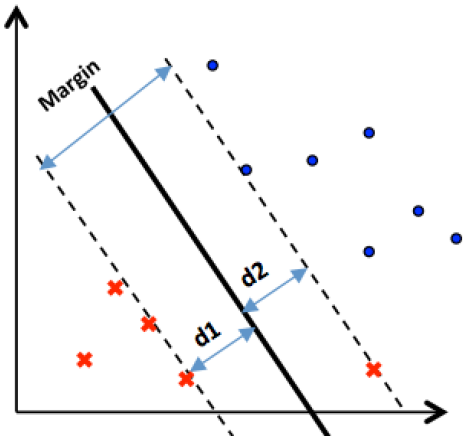

FIGURE 2.10: Soft margin

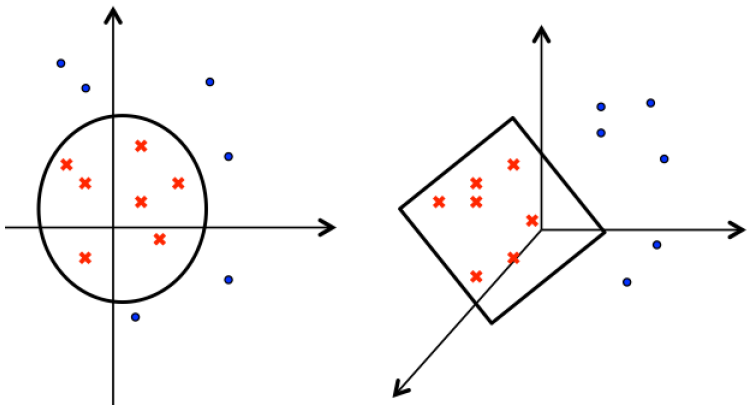

FIGURE 2.11: Mapping original data to a higher dimensional feature space

2.4.3 Feature Engineering

Feature Transformation

Feature transformation techniques increase the accuracy of the learning model by creating new features based on the original features. These techniques may include standardization as well as dimensional reduction techniques.

Standardization helps to scale the data point, i.e., forcing them to have the same scale by assigning their mean to zero and standard deviation to one.

Dimensional reduction techniques are used to reduce the high dimension of feature spaces. One of the well-known feature reduction techniques is called PCA, which transforms features to a set of new features that are non-linearly correlated [50].

Principal component analysis (PCA) When data is complex with too many correlated features, in order to reduce the difficulties of data, one may consider analyzing the data and its features. However, it is nearly impossible

to interpret all the patterns that result from analyzing each feature against other features. PCA is a statistical technique used to reduce the dimensionality of features in order to reduce data complexity while keeping as much information as possible of the original data. In most cases, PCA increases the accuracy of models by creating a new combination of features. In this case, PCA uses a linear combination of features with different weights. Each principal component is a linear combination of features with a maximum variance but uncorrelated with other principal components. Interpreting these principal components against each other is less expensive. Standardization as described above is usually an important pre-step for PCA that helps to highlight useful features.

Feature Selection

In machine learning problems, it is not always easy to predict what features among all features contribute more to the accuracy of the built model unless a feature selection technique is used. Feature selection techniques give scores to features based on their importance and finally select the most important features to build a model. The more a feature is correlated to the class prediction the more influence it has. One of the well-known feature selection techniques is Mutual Information (MI).

Mutual Information Mutual information measures the association between features and classes and scores the features accordingly.

Mutual information formula is as follows:

$$MI(F,C) = H(F) + H(C) - H(F,C) = H(F) + H(F|C) = \sum_{f \in F} \sum_{c \in C} p(f,c) \log \frac{p(f,c)}{p(f)p(c)} \tag{2.13}$$

where F is a set of features and C is a set of classes. $H(X)$ is a marginal entropy for discrete random variable X, which calculates the amount of not known knowledge about X. $H(X|Y)$ and $H(X,Y)$ are the conditional entropy for variable Y and joint entropy between X and Y; respectively. f and c have probability distributions of $p(f)$ and $p(c)$. $p(f,c)$ is their joint probability distribution and $p(f)p(c)$ is their product distribution [9].

2.5 Ontology Features

Machine learning aids with effective and successful decision-making in different states related to reasoning about ontologies, e.g., assigning the most optimal DL reasoner to a specific ontology based on the ontology's features [36]. Furthermore, ontology features play a pivotal role in predicting reasoner performance. Many works have revealed the impact of ontology features on machine learning approaches for reasoners [23, 22, 2]. Ontologies can be very complex and variant, i.e., choosing relevant features are critical for increasing the prediction accuracy of applications related to ontology learning [40]. Additionally, Automated Theorem Prover (ATP) frameworks such as E-MaLeS proposed learning-based feature tuning [26]; however, depending on the specification of ontology learning problems, these frameworks may or may not be generalized.

Ontology features are categorized into four categories [1]:

1. Ontology size related feature contains the basic measures of an ontology's signature and axioms, e.g., number of named classes, object properties, logical axioms, etc.

2. Ontology expressivity feature describes the features related to the DL family name of an ontology.

3. Structural feature is related to class hierarchy, richness, and cohesion of an ontology. For example, maximum depth of an ontology hierarchy, number of subclasses, subproperties, ratio of classes having attributes, etc.

4. Syntactical feature represents different ratios and frequencies related to classes (and their different types), properties, individuals, axioms and constructors. For example, the *ABox ratio* feature is the ratio of ABox axioms to all other axioms.

In this thesis, in addition to some of the already established features from the above categories, we also created some features that are believed to be useful for this research. More details on those features and the reasoning behind choosing them for each ML-based approach are given in Chapter 4.

2.6 Summary

This chapter covered the main concepts required to understand the next chapters of this thesis. The provided knowledge contains description logic, tableau optimization techniques, machine learning, and features related to

ontologies. In the next chapter, we focus on the literature review, which contains related work and the main contribution of this thesis.

Chapter 3

Literature Review

3.1 Introduction

OWL reasoners have been integrated with various optimization techniques introduced in the previous chapter. These techniques are designed to speed up the reasoning process in OWL reasoners. Based on their effectiveness and usage, they could have a drastic impact on the performance of these reasoners. These tableau-based optimization techniques could further be improved due to their heuristic nature.

In this chapter, the literature review is provided in two parts. In the first part (Section 3.2), we study the possibility of providing decision-making strategies for optimization techniques in order to deal with their non-deterministic behavior. Then among these optimization techniques, we select three of them that we think are more affected by heuristic decision-making. Later, in Section 3.3 and 3.4, we discuss the related work and our contributions in this thesis, respectively.

3.2 Learning from Tableau Optimization Techniques

In this section, some of the optimization techniques of Chapter 2 that require decision-making for their non-deterministic behavior are introduced. Knowing the source of non-deterministic actions in those optimization techniques helps us to build algorithms based on machine learning, which are able to learn in order to make the best decision while encountering various heuristic options.

The absorption techniques from the preprocessing category of optimization techniques (see Section 2.3.1) are very useful for decreasing the number of GCI axioms as the main source of non-determinism in many ontologies, i.e., helping to make the search space smaller. Using the best combination of the absorption techniques could speed up DL reasoners. In this case, we can learn how and in which order to apply the combination of those absorption techniques to get the best performance out of the reasoner. Moreover, the absorption techniques can be combined with other optimization techniques based on their functionality. For example, compared to concept absorption, role absorption has the advantage of not reducing the concepts while removing GCI axioms; therefore, the CD classification technique (see Section 2.3.3) will remain applicable even after role absorption.

As said in Section 2.3.2, semantic branching is used to reduce the cost of non-determinism in syntactic branching. If the single disjunct that we chose for splitting the tree branches is large, it may not be that efficient to apply semantic branching. Examining the size of disjunct may help to choose

between syntactic and semantic branching. However, it is already proven that semantic branching is still a better choice even if the single disjunct is a large concept [20].

Moreover, in both syntactic and semantic branching, we have non-deterministic decisions to make due to the existence of disjunctions in ontologies. For semantic branching, we may learn which single disjunct is a better choice and leads to a better outcome. Therefore, we build a learning-based technique that predicts which concept as a disjunct will have a drastic impact on the reasoning speedup.

Additionally, there exists an optimization technique that determines the order of applying disjuncts in disjunctions (see Section 2.3.2). Making the best decision among the different expansion-ordering heuristics developed for this purpose could have a huge impact on the reasoning speedup.

Further, the proper order of applying rules can be learned as well in which a priority is assigned to each rule (see Section 2.3.2).

When it comes to caching (see Section 2.3.2), machine learning could help us to identify some pattern combinations such as a pattern similar to the subset, superset satisfiability/unsatisfiability dependency. Moreover, a cache store that keeps the nodes with their satisfiability status could be very memory consuming since it should keep the nodes and their satisfiability status until the end of the tableau algorithm process. To avoid this memory consumption, one can learn which nodes should be stored until the end of the reasoning process. Also, some of the nodes could be removed from storage based on their usage where this knowledge could be achieved by learning.

In order to maximize the effect of classification-based optimization techniques, we may consider other tree traversals and look for the one that

has the maximum speedup effect (see Section 2.3.3). Also, some improvements might be possible by modifying the type of tree traversals; therefore, a learning-based strategy can predict which type has the best effect overall.

Besides, to minimize the number of subsumption tests during the classification, it is important to consider the order of adding named concepts. Some techniques have already been suggested for this purpose; such as definition order, told subsumers, replacing expensive subsumption tests, clustering (see Section 2.3.3). With the help of machine learning, we may predict which of those techniques could maximize the effectiveness of the classification optimization more than the others while also minimizing the number of subsumption tests.

As said in Section 2.3.4, component-based modularization can generate a small number of large modules that is not very sufficient especially in the case of many role assertions; therefore, intentional-based modularization has been introduced. Even though intentional-based modularization generates smaller modules that are easier for reasoning, the number of times we need to apply the $\forall$–info structure is very important. It might have too many roles; therefore, the performance of $\forall$–info may not be better compared to component-based modularization.

Also, as mentioned, there might exist different options for generating ABox modules. Some of those modules may not be proper according to the stated example of Section 2.3.4. This problem can be solved by a technique called bridge rules, which connect the useful information between ABox modules with possible connections but this solution causes overhead [48]. Therefore, one can come up with a strategy that learns how to avoid such useless modularizations in the first place as well as choosing the modules

that lead to the best outcome.

As seen from the examples in Section 2.3.5, when nominals occur in axioms, the nominal absorption techniques are more effective than other absorption techniques such as concept absorption. First, we can make sure if this applies to every case and then build a strategy that predicts if nominal absorption must be preferred over other forms of absorptions in some cases.

Since the model merging algorithm may contain non-determinism similar to the tableau-based algorithm (see Section 2.3.5), one can choose between two options where one leads to non-mergeable pseudo-models and the other proves the opposite. Therefore, we can manage the selection process with a learning-based strategy to avoid useless steps.

3.3 Related Work

Semantic branching (see Section 2.3.2) is a primary solution for the SAT problem in propositional logic. A SAT solver's runtime varies based on the given instance and the heuristic algorithm (branching heuristic) used for choosing variables in each decision level. In recent years, many novel branching heuristics have been developed for SAT solvers. One of these is LRB [29], which is based on reinforcement learning. To solve a SAT problem efficiently, another approach was developed to apply reinforcement learning in order to select the branching heuristic with a maximum long-term reward to give the most proper order for variables with the least overall cost [27]. In another work, a different learning technique called multinomial logistic regression is used to solve Quantified Boolean Formulas (QBF) that are more expressive

than propositional logic [39]. However, the proposed solution is only applicable to QBF expressions with binary clauses.

As stated above, a well-known decision-based method applied by SAT solvers involves the application of different branching heuristics in order to determine which concept (or branch) must be chosen when dealing with disjunctions [32]. Unlike SAT solvers, DL reasoners are designed to handle more than propositional satisfiability. Since search strategies and branching heuristics are shown to be very effective for SAT and Satisfiability Modulo Theory (SMT) based problems [7], encoding DL expressions to SMT might yield a potential solution. However, the encoded results might become very large and thus intractable for real-world ontologies [14]. While in propositional logic every concept is considered an atomic concept, in DL expressions, concepts are rather rich and may have complicated structures. In this case, applying branching heuristics such as MOMS (see Section 4.3.1) in every tree level is not equivalently effective for OWL reasoners. The reason is that these methods do not take into account the dependencies from other branches and could even reduce the reasoner performance [4, Ch. 9]. However to enhance the effects of backjumping [6], one could select the branches that contain concepts with old dependencies [21, Ch. 7]. Nonetheless, this approach could not boost the performance as much either. Further, a feedback-based heuristic technique was proposed in order to reduce the expansion of disjuncts with a clash. It uses the characteristics of the already expanded clash-free disjuncts for detecting and prioritizing the not yet expanded clash-free disjuncts [44]. The technique is mainly effective for ontologies with a large number of nominals.

Since optimizing the expansion of each disjunction can dramatically

speed up reasoner performance, designing similar heuristic strategies that can benefit satisfiability testing for OWL reasoners is both required and attainable. Hence, Tsarkov and Horrocks [46] introduced a new heuristic strategy for ordering branches (sub-concepts). The suggested heuristics take into account the metrics of the concepts (or sub-concepts) such as frequency of concepts in an ontology, size of concepts and depth of their quantifiers. These metrics presumably play a major role in heuristic decisions in the satisfiability tests of the reasoner.

Moreover, selecting suitable heuristics by utilizing machine learning algorithms in satisfiability testing or similar tasks has substantially enhanced the runtime for variants of SAT, SMT and QBF solvers [27, 39, 25]. For resolution-based theorem provers, different statistical machine learning methods are used to predict the plausibility of search branches for faster query results [43]. There are also learning-based approaches for detecting the best possible expansion [44]. The proposed method by Sirin et al. [44] reduces the expansion of inherently clash generating disjuncts. It orders the disjuncts based on the characteristics they share with already expanded clash free disjuncts. Furthermore, some reasoners dramatically benefit from caching [37]. Caching can be used indirectly for the (un)satisfiability status of (sub/super) concepts in disjunctions [16, 10]. Caching is used with quantifiers and in this case, the status of generated successors can be cached for look-alike concepts.

Furthermore, arranging the order of tableau rules employed by OWL reasoners also requires a decision-making procedure. As mentioned in Section 2.3.2, a more effective approach compared to the top-down approach is based on a ToDo list architecture, where heuristics are applicable at a global level [46]. A local ordering may not be quite applicable, and thus not useful

for learning purposes due to the runtime overhead of the learning process at each tableau level.

3.4 Contributions

Non-determinism related to disjunctions can be resolved using a simple method based on boolean solvers, such as SAT and QBF solvers. Therefore, similar to the approach used for QBF solvers [39], we employ multinomial logistic regression. Our method, however, applies machine learning to OWL reasoners and not to SAT solvers. Although propositional logic is less expressive than description logic [4], a significant percentage of OWL ontologies contain concept descriptions resembling propositional logic. Also, our method does not restrict the number of clauses or variables in logic expressions. In order to integrate our approach into OWL reasoners, for each clause in the reasoner's input, the order of variables is changed. The new order is specified based on the order given by solving an input ontology as a SAT problem using machine learning; in other words, the reasoner uses our fully trained model to choose the new order. Thus, the clauses in each branching level of reasoning have a new order that could lead to a smaller search space. To summarize, an ML-based selection strategy for branching heuristics has been applied to improve semantic branching in satisfiability testing for OWL reasoners [33]. However, the approach is limited to propositional description logic.

To expand the approach beyond propositional logic, we apply machine learning to choose among the built-in expansion-ordering heuristics that are developed for input with more than just propositional logic [35]. These

heuristics are built based on some structural characteristics of ontologies that may have an impact on the order of selecting disjuncts. More details on these heuristics are given in Section 4.3.2. The ML-based method determines the proper heuristic among the built-in heuristics. Later, the chosen heuristic is used to determine the order of selecting disjuncts in each disjunction. The goal is to at least avoid encountered timeouts of the reasoning process for ontologies by learning to choose the right heuristics.

Finally, to improve uncertain decision-making related to the order of rules, we applied machine learning to the heuristic-based *ToDo list* optimization in DL reasoners [34]. While performing reasoning tasks on ontologies the order of applying rules has a great impact on reducing the search space. To help in finding a proper solution, we have also offered some heuristics for the orders of applying rules as presented in Section 4.4. Our ML-based approach identifies the best order of applying different rules during the reasoning process.

3.5 Summary

This chapter presented the literature review. We investigated the possibility of heuristic decision-making for improving tableau optimization techniques. Further, the previous (learning-based) methods used to speed up different types of solvers such as SAT, QBF, and SMT solvers are explored. Finally, we discussed our contribution to the related work.

In the next chapter, our ML-based methodology for optimizing each of the three optimization techniques that are selected for the improvement of their heuristic decision-making is presented.

Chapter 4

ML-based Approaches to Handle Decision-making

4.1 Introduction

From the previous chapter, we studied the potential for handling decision-making in the tableau optimization techniques of OWL Reasoners. This chapter introduces our methodologies for improving decision-making in those techniques. These methodologies are based on machine learning and they are developed to comply with three of the well-known tableau optimization techniques including *Semantic Branching*, *Heuristics for Choosing Expansion-Ordering*, and *Ordering of Expansion Rules*. The evaluation of these approaches is also demonstrated in this chapter.

Section 4.2 presents the main common steps for the three approaches. In Section 4.3, two approaches for selecting the right disjunct in disjunctions are presented. The first approach (see Section 4.3.1) is an ML-based approach that learns to select the right disjunct for *Semantic Branching* optimization

technique. The second approach (see 4.3.2) is another ML-based approach that learns the right disjunct in *Heuristics for Choosing Expansion-Ordering* optimization technique. The final approach that predicts the right order of applying rules (for *Ordering of Expansion Rules* optimization technique) is presented in Section 4.4.

4.2 Main Steps

This section introduces the main steps of our approaches for handling decision-making in OWL reasoners. The ML-based approaches are based on an independent systematic analysis of heuristics to uncover ontology patterns and their associated features that are relevant for each specific optimization technique.

Figure 4.1 shows the five main steps taken for the ML-based strategy. In Step 1, the input ontology is loaded. In Step 2, the features related to the input ontology are computed. Step 3 contains the process of building the model from training data (ontology features derived from training data and their associated labels). Step 4 predicts a proper label for a given input ontology. Finally, in Step 5, the predicted label is used for optimizing the reasoning task. Each of these steps for each approach is explained in detail later.

This section also presents our experimental results. We selected JFact[1], a Java port of FaCT++[2], as our prototype reasoner to conduct our experiments. Although the developed approaches are applicable to every reasoner that can accept super sets of $\mathcal{ALC}$, we chose JFact for its maintainability, extensibility,

[1]https://github.com/owlcs/jfact/
[2]https://bitbucket.org/dtsarkov/factplusplus/src

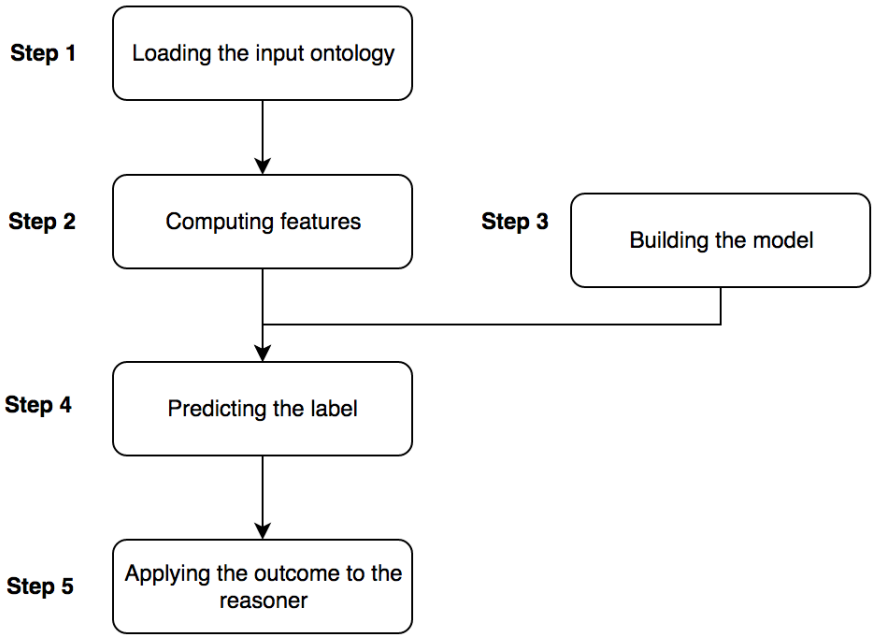

FIGURE 4.1: The machine learning based steps for predicting a label for an input ontology

and robustness. In the remainder of this thesis, ML-JFact refers to our ML-based JFact prototype while JFact refers to the unmodified JFact reasoner.

4.3 Learning to Select the Right Disjunct

This section introduces two ML-based approaches that were designed to make the selection process for disjuncts more effective. The first approach uses branching heuristics to select a proper disjunct and the second approach uses the built-in heuristics of DL reasoners (aka expansion-ordering heuristics) in order to make a proper decision. The second approach is the evolved version of the first one since it covers ontologies beyond propositional logic.

4.3.1 Approach I-A: Learning the Right Disjunct Using Branching Heuristics

In the following, first, the branching heuristics (the labels[3] to be predicted for each input ontology) are discussed. Later, the list of features to help build the learning models is presented. Finally, the procedure and experimental results are presented.

Labels: Branching Heuristics

As already noted, the order of applying variables (concepts) in semantic branching can significantly affect the resulting runtime. Based on different branching heuristics, variables can have different scores. These branching heuristics try to prune the search tree by targeting specific variables and clauses.

The formula below determines the score of each variable ($score\_var(Var)$) by considering the score of its literals in each branch, so it could balance between the branches to increase a variable's score. The first branch considers the positive from $score\_lit(Lit^+)$ and the second branch considers the negative from $score\_lit(Lit^-)$:

$$score\_var(Var) = score\_lit(Lit^+) + score\_lit(Lit^-) \tag{4.1}$$

Lit^+ and Lit^- are the positive and negative forms of the variable Var.

[3]For the rest of this chapter, the words "label" and "class" are used interchangeably for heuristics. Heuristics are the predicted outputs of our ML-based approaches for given input ontologies.

Choosing the variable with the highest score ($score\_var(Var)$) saves time by leading to a smaller search space. Moreover, by knowing the scores of both literals or branches ($score\_lit(Lit^+)$ and $score\_lit(Lit^-)$), one can also choose between the branches. In our implementation, we are only concerned about changing the order of selecting variables in the reasoning process.

The well-known branching heuristics that are used in our implementation are as follows.

First literal The first literal occurring in the boolean formula is returned.

MOMS The literal with the maximum number of occurrences in the clauses with minimum-sized is considered.

$$MOMS(l) = \text{occurrences of } l \text{ in minimum-sized clauses.} \tag{4.2}$$

MOMSF It is the alternative of *MOMS*. If $f(x)$ is the number of occurrences of the variable x in the clauses of minimum-sized, we return the variable maximizing

$$(f(Lit^+) + f(Lit^-)) * 2^k + (f(Lit^+) * f(Lit^-)) \tag{4.3}$$

where k is a defined constant. In Freeman [13, Ch. 7] some factors that lead to choosing the best priority function are indicated.

MAXO The literal with the maximum number of occurrences in the boolean formula that has the greatest score.

$$MAXO(l) = \text{number of occurrences of } l \text{ in the formula.} \tag{4.4}$$

JW The Jeroslaw-Wang chooses the literal that maximizes the following equation where n_j is the number of literals in the clause C_j:

$$JW(l) = \sum_{j, l \in C_j} 2^{-n_j} \tag{4.5}$$

JW2 The two-sided Jeroslaw-Wang is the same as *JW*, but to choose a variable it considers its score by adding the score of its positive and negative literals together.

DLCS (Dynamic Largest Combined Sum) It counts the number of clauses in which the literal and its negation occur in and assigns them respectively with *CP* and *CN*:

$$DLCS(Var) = CP(Var) + CN(Var) \tag{4.6}$$

The variable with maximum *DLSC* is selected. If $CP \geq CN$, the positive form of the variable (l) is chosen and if $CP < CN$, the negative form ($\neg l$) is chosen. *DLSC* is faster than *JW*.

POSIT it is similar to *DLCS*, but the search is only done on clauses with minimum-sized.

DLIS (Dynamic Largest Individual Sum) It is the same as *DLCS*, but it chooses the maximum value among all *CP* and *CN* values; then, it picks the variable with the maximum value. *DLIS* performs faster than *DLCS* and on some benchmarks, it performs twice as fast as *JW*.

To obtain a label (or proper branching heuristic) of each training data, all heuristics are applied to the training data and for each data, the heuristic that

runs fastest will be considered as the label of that training data. Algorithm 1 shows the procedure of determining labels for training data. The training data and their associated labels are used later to build our learning model. The learning model is then used to predict the labels for the test data (ontologies in sample tests).

Although more novel branching heuristics have been developed recently, none of them is the most effective solution for all benchmarks. For example, *MOMS* may only be good for the logic expressions with many binary or unary clauses [30], but when considering all possible inputs and including features such as the number of unary and binary clauses, *MOMS* may lead to very effective results. Therefore, the branching heuristics were chosen in such a way to consider input files with different attributes.

Clause-based Features

Ten features are chosen for building our model. These features are based on the clause structure of logic expressions in ontologies. They are a good indication for the unit propagation procedure because they reduce the search space (discussed in Section 2.3.2) that chooses the best branching heuristic. The features are shown in Table 4.1.

Procedure: Choosing the Right Branching Heuristic

Figure 4.1 shows the main steps of predicting a label for an input ontology. The input ontology (test data) will be converted into Conjunctive Normal Form (CNF) (Step 1). This input format is required for clause-based SAT solvers that are employed in generating ML-based models. After the features

Algorithm 1 Find labels of training data for Approach I-A

Input: Training Data
Output: Labels for Training Data

```
1:  procedure FINDLABELS(training-data)
2:      n ← number of training data
3:      k ← number of branching heuristics
4:      for td ← 1 to n do
5:          for bh ← 1 to k do
6:              T[td, bh] ← ComputeTimeSolve(td, bh)
7:          end for
8:      end for
9:      Label = min T(td, bh) where td ∈ 1...n and bh ∈ 1...k
10:     return Label
11: end procedure
12: function COMPUTETIMESOLVE(td,bh)
13:     startTime ← currentTime
14:     SatSolver(td, bh)
15:     executionTime ← currentTime − startTime
16:     return executionTime
17: end function
18: function SATSOLVER(td,bh)
19:     if td is solved then
20:         return
21:     else
22:         update input td after using heuristic bh to solve td
23:         SatSolver(td, bh)
24: end function
```

TABLE 4.1: Selected input features

Ontology Clause-based Features
Number of variables (var)
Number of clauses (cls)
var/cls
$(var/cls)^2$
$(var/cls)^3$
Ratio of binary clauses to all clauses
Ratio of ternary clauses to all clauses
Ratio of horn clauses (clauses with exactly one positive literal) to all clauses
Number of positive literals
Number of negative literals

of the input ontology are computed (Step 2), the built model helps to predict a label for the input ontology. In Step 3 the built model is based on a machine learning technique called Multinomial Logistic Regression. Multinomial logistic regression helps with the prediction when there exist multiple labels (branching heuristics). Afterward, the proper label is predicted for an input ontology (Step 4). Finally, based on a reasoner's implementation, the order of disjuncts could be changed for the input ontology (as a preprocessing step before reasoning starts) because the disjunction rule always selects disjuncts from left to right. Alternatively, the implementation of the disjunction rule could be changed to dynamically select a proper disjunct based on the computed model (Step 5). For ML-JFact we selected the reordering technique.

Results for Approach I-A

The training data used for this experiment are files in CNF format. Our training data are from different benchmarks available on SATLIB.[4] Those benchmarks are JNH, AIM, PARITY, Flat, CBS, RTI, BMS, and Uniform Random (the statistics are shown in Table 4.2). The benchmarks available from SATLIB are limited to a specific range of variable and clause numbers; for example, there are only a few CNF files with less than 50 variables and all of the files have at least 20 variables. To avoid this restriction, we added about 147 additional CNF files. These additional files contain small CNF inputs with between 1 to 50 variables. The final training data contains 1400 files including both satisfiable and unsatisfiable CNFs. Table 4.3 shows some attributes of the training data.

[4] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

TABLE 4.2: Training data chosen from SATLIB (number of samples)

Library	JNH	AIM	PARITY	Flat	CBS	RTI	BMS	UF3SAT	Others	Overall
Training samples	47	24	9	79	623	24	80	367	147	1400
Satisfiable	14	16	9	79	623	24	80	279	107	1231
Unsatisfiable	33	8	0	0	0	0	0	88	40	169

TABLE 4.3: Overall training data attributes

Number of variables			Number of clauses		
Min	Max	Avg	Min	Max	Avg
1	558	103	1	1801	434

In the above experiment, we only deal with propositional logic input. Therefore, the training data contains files in CNF DIMACS format to be solved with a SAT solver. A standard SAT solver, which is based on the DPLL algorithm, is used for training data.

To verify whether providing more training data can lead to more accurate results, we use a learning curve to show how increasing the number of training data can increase/decrease the accuracy of the classifier. As shown in Figure 4.2, we think that adding more training data will probably not change the accuracy of the classifier significantly.

As stated in Section 4.3.1, the nine branching heuristics (labels) used for this experiment are *FirstLiteral*, *MOMS*, *MOMF*, *JW*, *JW2*, *POSIT*, *ZM*, *DLSC*, and *DLIS*. Based on the Algorithm 1, for each training data, the selected branching heuristic is used until the solver process is finished. The same pattern applies to the sample tests. Table 4.4 shows for each heuristic the number of training samples with an improved runtime.

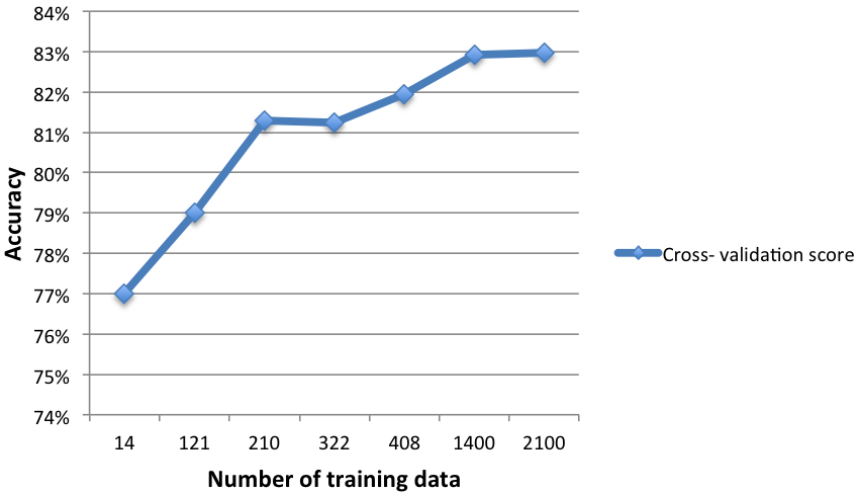

FIGURE 4.2: Cross-validation learning curve

TABLE 4.4: Number of 1400 training samples with an improved runtime for each heuristic

Heuristics	First Literal	DLSC	DLIS	MOMS	MOMSF	POSIT	ZM	JW	JW2
Count	92	159	129	90	292	170	111	137	220

In our experiment, we used 40 sample tests where the number of clauses is between 150 and 400 and the number of variables between 35 and 90. Half of the samples are satisfiable and half are unsatisfiable. A 7-fold cross-validation was carried out on 1400 training samples with a learning accuracy of 83%.

The performance of 40 sample tests is shown in Table 4.6. For each sample the table lists the JFact runtime, the ML-JFact runtime, the number of JFact backjumps, the number of ML-JFact backjumps, the runtime speedup (defined as JFact runtime divided by ML-JFact runtime), the backjump reduction (defined as the number JFact backjumps divided by the number of ML-JFact backjumps), and also the number of clauses, variables, and the SAT status. Table 4.6 also shows that there is a direct relationship between the speed improvement and the reduction in the number of backjumps. For example, for

the sample number 27, the number of backjumps went from 7,699,075 down to 271,423 (reduction of 28) while the runtime improved from 106,941 milliseconds to 4,393 milliseconds (speedup of 24).

TABLE 4.5: Speedup improvement

Improvement	in milliseconds	Speedup ratio
Maximum	102,548	311.2
Average	18,500	14.3
Sum	731,106	1.86

The overall performance of the algorithm is given in Table 4.5. It lists the maximum and average as well as the sum of all runtime improvements (in milliseconds) and their corresponding speedup factors. This shows that applying ML-based techniques improves the speed of satisfiability checking by one to two orders of magnitude.

Moreover, the percentage of improvement based on (un)satisfiability suggests that the learning technique is more successful for satisfiable cases. The rate of success for unsatisfiable and satisfiable input are 53.01% and 78.25%, respectively. These percentages could be due to the fact that the training data contains more satisfiable cases.

Discussion

Changing the order of disjuncts in disjunctions does not affect JFact's functionality; therefore, this ensures both soundness and completeness of ML-JFact.

Since the goal is to achieve a significant runtime improvement (e.g., at least 10%), we mostly focus on sample tests that perform in a sufficient amount of time (e.g., at least several seconds) such that the improvement

TABLE 4.6: *Approach I-A*: Performance of 40 sample tests (runtimes in milliseconds)

No.	Runtime (without learning)	Runtime (with learning)	No. of backjumps (without learning)	No. of backjumps (with learning)	Runtime speedup	Backjump reduction	No. of variables	No. of clauses	Sat/ Unsat status
1	22,651	10,569	1,892,308	834,000	2.14	2.26	60	250	S
2	16,431	10,871	1,368,222	899,510	1.51	1.52	60	245	S
3	25,247	3,699	2,252,016	267,111	6.82	8.431	65	257	S
4	26,270	2,051	2,223,617	133,915	12.80	16.60	65	265	S
5	24,351	7,577	2,085,350	817,211	3.21	2.55	59	250	U
6	597,447	534,735	56,537,985	50,330,597	1.11	1.12	60	160	U
7	6,105	1,290	667,257	92,138	4.73	7.24	34	150	S
8	7,313	1,248	784,748	94,423	5.85	8.31	34	151	S
9	7,148	1,958	700,315	158,318	3.65	4.42	40	156	S
10	52,989	31,587	4,380,675	2,609,718	1.67	1.67	75	325	U
11	16,031	9,866	940,941	273,202	1.62	3.44	80	346	U
12	25,959	2,883	2,223,617	191,346	9.004	11.62	70	270	S
13	15,438	9,182	1,277,983	712,342	1.68	1.79	59	240	U
14	17,097	9,001	1,365,016	731,959	1.89	1.86	65	243	U
15	20,229	10,222	1,771,462	789,925	1.97	2.24	59	261	U
16	14,577	3,094	866,465	189,226	4.71	4.57	80	350	U
17	15,276	3,556	886,084	223,494	4.29	3.96	85	354	U
18	16,770	5,511	1,019,601	350,294	3.04	2.911	90	354	U
19	12,657	1,388	741,914	86,578	9.11	8.56	90	342	S
20	40,452	30,015	3,245,607	2,423,444	1.34	1.33	75	333	U
21	37,079	16,871	2,664,404	1,180,861	2.19	2.25	75	400	U
22	39,078	31,866	3,000,095	1,688,700	1.22	1.77	75	395	U
23	7,268	1,800	709,942	158,874	4.03	4.46	45	160	S
24	7,469	24	676,268	37	311.20	18277.51	60	162	S
25	70,240	8,537	4,400,237	507,550	8.22	8.66	75	325	U
26	3,690	1,573	199,824	90,270	2.34	2.21	75	377	S
27	106,941	4,393	7,699,075	271,423	24.34	28.36	75	305	S
28	23,989	5,146	1,582,981	327,533	4.66	4.83	70	325	U
29	18,779	800	1,212,269	32,067	23.47	37.8	75	325	S
30	12,343	3,072	755,159	154,546	4.01	4.88	91	354	U
31	54,945	13,671	3,939,491	800,453	4.01	4.92	75	325	S
32	5,699	3,807	464,096	304,361	1.49	1.52	55	250	U
33	19,493	9,511	1,593,995	740,132	2.04	2.15	59	243	U
34	6,866	1,641	677,743	136,166	4.18	4.97	45	160	S
35	15,616	290	918,259	5,308	53.84	172.995	75	325	S
36	6,644	373	362,336	7,634	17.81	47.46	74	321	S
37	6,364	1,545	579,600	108,859	4.11	5.32	45	163	S
38	5,744	753	551,661	49,356	7.62	11.17	34	155	S
39	110,952	15,244	6,627,625	850,760	7.27	7.8	75	343	U
40	8,779	6,090	514,896	359,794	1.44	1.43	70	343	U

is obvious. For example, the samples with less than 50 variables and 100 clauses mostly perform in less than one second; therefore, improving their speed is not our primary concern.

On the other hand, the sample number 6 from Table 4.6 (with 60 variables and 160 clauses) has an initial runtime of 597,447 milliseconds and the runtime is improved by 62,712 milliseconds after learning. Therefore, the percentage of improvement is 10.49%, which is still considered as good.

Our approach does not improve the runtime for every input. For one out of every five input files (20%), the runtime is mostly unchanged or increased by only less than 10%. For example, the increased runtime of about 2% after learning is considered as a tolerable performance loss.

To reduce the number of cases without a significant speedup, we believe it is a good idea to discover specific CNF patterns where a learning improvement is not expected. For example, in one of the identified patterns for every clause, there also exist clauses with the exact same variables of that clause but every other possible combination of their literals. An example of such pattern is the following description logic axiom.

$$
\begin{aligned}
D \equiv\ & (A \sqcup E \sqcup F) \sqcap (A \sqcup \neg E \sqcup F) \sqcap \\
& (A \sqcup E \sqcup \neg F) \sqcap (A \sqcup \neg E \sqcup \neg F) \sqcap \\
& (\neg A \sqcup E \sqcup F) \sqcap (\neg A \sqcup \neg E \sqcup F) \sqcap \\
& (\neg A \sqcup E \sqcup \neg F) \sqcap (\neg A \sqcup \neg E \sqcup \neg F) \sqcap \\
& (B \sqcup C) \sqcap (B \sqcup \neg C) \sqcap \\
& (\neg B \sqcup C) \sqcap (\neg B \sqcup \neg C)
\end{aligned}
$$

4.3.2 Approach I-B: Learning the Right Disjunct Using Expansion-ordering Heuristics

Similar to the previous approach, this approach also focuses on properly ordering disjuncts in disjunction expressions of ontologies for satisfiability testing. However, compared to the previous one, it covers more than propositional logic. Moreover, it focuses on the different built-in expansion-ordering heuristics developed specifically for this purpose. These built-in heuristics in reasoners determine the order for branches in search trees while each heuristic choice impacts the performance of various ontologies depending on the ontologies' syntactic structure and probably other features as well. An ML-based approach helps to detect the right expansion-ordering heuristic.

In the following, first, the labels for the ML-based approach are introduced. Later, the features and ML-based procedure are presented and the experiments are given.

Labels: Expansion-ordering Heuristics

Many DL reasoners are accompanied by various configuration settings that help to integrate optimization techniques for different reasoning tasks. One of the parameters in their configuration settings provides customizable expansion-ordering heuristics that determine the sorting order for concepts in disjunctions. These expansion-ordering heuristics are set for both satisfiability and subsumption tests. Since the current focus is only on satisfiability testing, for the rest of this section, expansion-ordering heuristics are only referring to heuristics for satisfiability testing.

TABLE 4.7: Configuration labels

Config#	1	2	3	4	5	6	7	8	9	10	11	12
Labels	SAP	SDP	FAP	FDP	DAP	DDP	SAN	SDN	FAN	FDN	DAN	DDN

The configuration setting options for expansion-ordering heuristics are in the form of a string[5] "`MOP`" defined as follows: "`M`" is a sorting name character field, which has the options:[6] `S` (stands for size), `D` (stands for depth), and `F` (stands for frequency). "`O`" is an ordering type character field, which has the options: `D` for descending and `A` for ascending. Finally, "`P`" is a preference character field, which is either `P` for preferring generating rules or `N` for not preferring generating rules. Rules that deal with $\exists$ or $\geqslant$ concept constructors and add new nodes to a tableau are called generating rules. This leads to 12 heuristics as identified in Table 4.7. If no sorting order is provided ("`MOP`" is empty), the reasoner uses its default order.

Before any reasoning process begins, the reasoner computes all of the necessary statistics about each concept (such as size, frequency, depth, etc.) and stores them for later use that involves imposing orders for concepts in disjunctions.

Type-based and Disjunction-based Features

Ontology features play a significant role in reasoning tasks that involve decision-making. Therefore, to implement a robust ML strategy, features are

[5]For compatibility, we adopt this notation from JFact reasoner.

[6]These options are inspired by JFact reasoner. For the sorting name character field, two other statistics such as the number of branches and generating rules in concepts have been recognized. However, they have not been suggested as options to be chosen from in the configuration setting of the source codes for neither JFact nor FaCT++ [46]. For the sake of simplicity, we do not use them here.

defined to cover the structural characteristics as well as the disjunction related metrics of ontologies. Overall 39 features were used to build our model. These features are based on the type metrics of ontologies as well as the disjunction related metrics. Based on our experiments, a selection of features that were shown to be the key ones in increasing the accuracy of our model listed in Table 4.8.

TABLE 4.8: Main features of ontologies used in the built models

Ontology Related Metrics
Number of nominals
Number of instances
Number of classes
Average population
Number of GCI
Number of generating rules ($\exists$ and $\geqslant$)
Ratio of TBox axioms to all axioms
Ratio of RBox axioms to all axioms
Ratio of ABox axioms to all axioms
Number of Object Properties
Number of Inverse Object Properties
Number of Subclasses
Number of Equivalent Classes
Number of Disjoint Classes
Disjunction Related Metrics
Number of disjunctions
Average of the disjunctions' averages of sub-concepts sizes
Average of the disjunctions' averages of sub-concepts depths
Average of the disjunctions' averages of sub-concepts frequencies
Maximum number of children in disjunctions
Average number of children in disjunctions
Number of positive concepts is disjunctions
Number of negated concepts is disjunctions
Ratio of positive concepts to all concepts in disjunctions
Ratio of negated concepts to all concepts in disjunctions

The *Average Population* feature is defined as the ratio of instances to classes

in an ontology. This and some other features such as GCI consider the syntactic structure of ontologies that determine the similarity of these ontologies to particular types of ontologies [46].

Procedure: Determining the Right Order for Expansions

Learning to make effective decisions in non-deterministic situations are highly promising for reasoners. These decisions are particularly required for sorting either atomic or non-atomic concepts in disjunctions.

As already noted, the default sorting orders for expansion-ordering heuristics in reasoners are usually based on some initial assumptions obtained from past experiments that have analyzed a few syntactic features from a handful of ontologies, whereas other —possibly more important— features were disregarded. Moreover, as reported by Tsarkov and Horrocks [46], there is no general single sorting strategy that works best for all ontology types. Therefore, it is preferential to come up with a learning methodology that chooses the right sorting heuristic for each ontology based on its features such as the syntactic structure characteristics of the ontology.

Building a multi-class classification learning model that chooses from all 12 configurations[7] of Table 4.7 is not practical since the number of labels (configurations) is too high and this leads to a model with less than 50% accuracy. Instead, a more sensible solution is used in which for each configuration a separate binary classification model is built that determines if that heuristic is the right choice. Therefore, instead of a multi-class classifier with 12 labels,

[7]For simplicity, in this chapter, the terms (expansion ordering) heuristics and configurations are used interchangeably.

we have 12 binary classifiers where each predicts if its assigned heuristic is a "Good" or "Bad" choice.

Algorithm 2 shows for each training data, how the "Good" or "Bad" label is assigned to each configuration. In this algorithm, the threshold is a boundary that determines if a configuration associated with data is a "Good" or "Bad" choice. If the configuration's runtime is less than the threshold, its label is "Good"; otherwise, its label is "Bad". Later, in Section 4.3.2, we explain how the threshold (the average of mean plus standard deviation of the runtimes on the training data) is calculated.

Support Vector Machine (SVM) [8] is among one of the machine learning techniques that handles binary classification with a highly dimensional feature space effectively; therefore, it is used to build our highly dimensional model. Principal Component Analysis (PCA) [50] and Mutual Information [9] are also used for feature extraction.

Figure 4.1 shows the procedure of predicting a label for an input ontology. First, the features for the input ontology are computed (Step 1 & 2). Since we use a binary classification learning (from Step 3), for the input ontology a "Good" or "Bad" label should be predicted for each configuration of Table 4.7 (Step 4). The predictions are based on the built models for the configurations on the training data. Afterward, based on the priorities given in Table 4.12 and different scenarios mentioned in Table 4.13 (that will be explained in more detail in Section 4.3.2), the proper configuration from Table 4.7 is assigned to the input ontology. Finally, the reasoner chooses a concept from disjuncts based on the order determined from the assigned (sorting) configuration (Step 5).

Algorithm 2 Find labels of training data for Approach I-B and Approach II

Input: Training Data
Output: Labels for the Configurations of all Training Data

```
1:  procedure FINDLABELS(training-data)
2:      n ← number of training data
3:      k ← number of configurations
4:      for i ← 1, n do
5:          for j ← 1, k do
6:              T(i, j) ← Runtime of training data i with configuration j
7:          end for
8:      end for
9:      threshold ← average of mean plus standard deviation of all
        configurations on all training data
10:     for i ← 1, n do
11:         for j ← 1, k do
12:             if T(i, j) ≤ threshold then
13:                 Label(i, j) ← "Good"
14:             else
15:                 Label(i, j) ← "Bad"
16:         end for
17:     end for
18:     return Label
19: end procedure
```

Results for Approach I-B

To perform our experiments, ontologies from the OWL Reasoner Evaluation (ORE) 2014 competition[8] repository were used. The ontologies collected from ORE 2014 must be eligible for building the learning model, i.e., they must be affected by the expansion ordering heuristics due to the existence of non-deterministic situations. Based on that, initially, we were able to export around 3000 ontologies that cause non-deterministic situations and their expressivity is at least in $\mathcal{ALC}$.

Additionally, two more eligibility criteria were considered for collecting the training data. First, if for a single ontology, all 12 configurations lead to a timeout, then that ontology is discarded because it is not clear if the ontology is affected by those heuristics. The considered timeout, for this experiment, is 500,000 milliseconds (8.33 minutes). Due to the high complexity of the reasoning process, obtaining the exact termination runtimes for ontologies that lead to a timeout is not practical, thus, no number was assigned to them.

Second, if an ontology has very close runtimes (less than 2 seconds difference) for all 12 configurations, one may doubt its eligibility. The reason is that the difference could be just a runtime overhead from the system. To ensure this, multiple runtimes are examined and if for all runtimes the maximum and minimum runtimes belong to the same configuration, those ontologies are included in the training data because it is obvious that the heuristics are effective. Moreover, the ontologies that are inconsistent are discarded too.

Considering all the above conditions for all ontologies from ORE 2014 repository, only 143 were considered to qualify as training data, and 25% of these were saved for the test part.

[8] `http://dl.kr.org/ore2014/ontologies.html`

TABLE 4.9: Standard deviation and mean for the thresholds of all 12 configurations (numbers in milliseconds)

Config#	1	2	3	4	5	6
std	45240	41093	44969	30237	39744	31692
mean	90670	74557	84560	70336	82350	68626
std+mean	135910	115650	129529	100573	122094	100318
Config#	7	8	9	10	11	12
std	70340	48232	45791	31736	52981	39864
mean	119611	83761	85775	71612	99992	71706
std+mean	189951	131993	131556	103348	152973	111570

As stated previously, for each configuration (heuristic), a binary classification model is built. For each binary model of a configuration, there are two labels: "Good", which indicates that a heuristic is the right choice and "Bad", which indicates the wrong choice. To determine a threshold for identifying the "Good" and "Bad" labels in training data, we consider the average of the mean plus standard deviation of the training data for all those configurations. The standard deviation is added in order to take into account the outlier cases as well. Table 4.9 shows the mean, standard deviation and their sum for each configuration. The timeout cases are excluded while examining the data distributions. Threshold is defined as the average of the mean plus standard deviation for all configurations, which is about 127,122 ms (about 2 minutes).

To build each binary classifier, a 10-fold cross-validation SVM classifier is applied to the training data. To achieve the model with the highest accuracy, a grid search is conducted in order to find the best parameters such as the number of features in mutual information for feature selection, number of PCA components, SVM kernels, etc. For configurations 7 and 8 in Table 4.12,

SVM with Radial Basis Function (RBF) kernel outperformed linear SVM. RBF is a kernel function that separates the not linearly separable data by mapping it to a higher dimensional feature space.

A total of 35 ontologies have been chosen as test data. The experiments use additionally a few ontologies from DL'98 repository [19] and three specific ontologies with different ontology structures. Table 4.10 shows the ML-JFact runtimes of all 35 tests on all the heuristics (configurations). The symbol '*' indicates that a label has been falsely predicted. For example, 4201* (Config 5 of sample 7 in Table 4.10) indicates that the predicted label is "Bad" where the actual runtime is "Good". The configurations automatically selected for the samples are shown in bold.

The rightmost column shows the JFact runtimes for its default heuristics. This default configuration was determined based on experiments by Tsarkov and Horrocks [46] (and likely other unpublished experiments). The default sorting configuration is set to `FDN` for ontologies with the number of GCIs beyond a specific threshold but few or zero ABox assertions. The default configuration is set to `SDP` for the ontologies with more than 100 nominals but fewer GCIs and for all other ontologies, it is set to `SAP`.

The F-Score in Table 4.11 shows the overall performance of our classifier on all 35 test data for each configuration. F-Score indicates the balance between precision and recall. It focuses on both false positives samples (falsely predicted as "Good") and false negatives samples (falsely predicted as "Bad").

The accuracy of each configuration and its priority based on its accuracy on the training data are given in Table 4.12. If two configurations hold an

TABLE 4.10: *Approach I-B*: ML-Jfact runtimes (in milliseconds) for 35 satisfiability test cases selected from the ORE 2014 dataset; runtimes in the rightmost column from JFact (*TO* indicates a timeout; a bold font indicates the runtimes for the configuration chosen by ML-JFact and falsely predicted labels are marked by '*')

Sample	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6	Config 7	Config 8	Config 9	Config 10	Config 11	Config 12		JFact Std. Config
1	219016	98874	215667	90479	219501	104240	225228	98056	212528	**87984**	214417	109944		206076
2	50136	1701	51144	1658	**50219**	1791	50967	1786	50037	1683	49214	1809		1762
3	250	238	211	240	**219**	225	258	3714	240	228	228	8511		210
4	791	1625*	641	218	**807**	1487*	818	2210	606	319	791	2245		771
5	*TO*	33833	*TO*	28795	*TO*	36608	*TO*	*TO*	*TO*	**22632**	*TO*	*TO*		*TO*
6	255	3607	588	620	**266**	662	289	3650	590	641	271	615		255
7	4041*	3082*	**4111**	3138*	4201*	3243*	3088*	3081	4679	3085*	4208*	3170		3929
8	660	*TO**	*TO**	445	**875**	*TO**	907	*TO**	*TO**	435	896	*TO**		703
9	*TO**	234	*TO**	242	**TO***	240	*TO**	244	*TO**	250	*TO**	257		*TO*
10	10487	2356	5190	1954	**10419**	2429	12158	*TO**	5231	1970	11371	*TO**		10403
11	*TO*	13074	*TO*	10867	*TO*	15434	*TO*	13233	*TO*	**10152**	*TO*	14901*		*TO*
12	5156	25706*	12763	1850	**5703**	5693	*TO**	TO	12882	1339	5257	5685		26578
13	52562	13425	49296	11156	**50658**	16650	55552	14108	48152	11475	50508	16532		54102
14	8769	3797	8762	3815	**8744**	9739	8397	3693	8605	3182	8543	8816		8615
15	27569	30690*	30752*	24417*	**26763**	24450*	27193*	23954	28244*	24752	26981	24492		36137
16	64886	19113	60452	17049	**61710**	22812	69656	18843	60778	16559	63575	22548		67340
17	**194303***	31035	164548	14655	167601	22773	203327	72764*	187606*	14606	159787	74447		165275
18	263574	112662	258198	104167	270668	121018	282923	112816	253426	**101449**	264520	121824		270144
19	223	261	256	239	**238**	233	254	2002	254	230	234	8685		224
20	1081	*TO*	6375	1118	**2989**	1826	1855	*TO*	6329	877	2930	1870		*TO*
21	81995	317447*	61637	181266*	**59849**	328572*	84153	210318*	61372	177632*	60545	197244*		81484
22	1599	*TO**	12796	1840	**6939**	4029	41723	*TO*	13344	1183	6056	3896		*TO*
23	399	755	467	585	**547**	539	429	2415	469	616	726	842		378
24	*TO*	60399	*TO*	47739	*TO*	49117	*TO*	*TO*	*TO*	**48104**	*TO*	*TO*		*TO*
25	1566	2006	2100	1077	**1318**	1638	1093	7406	2043	1021	977	4465		1503
26	62399	1886	62606	1802	**61531**	1939	62470	1908	62140	1807	60337	1867		1816
27	5318	*TO*	27560	2603	**9103**	8188	166927*	*TO*	28023	1621	8096	8430		*TO*
28	51638	1757	51803	1696	**49528**	1779	51764	1821	50893	1654	50765	1767		1684
29	49109	5880	81882	5545	**42791**	5671	62527	6848	83461	5502	60410*	5802		49407
30	8972	6687*	8333	6518*	**8228**	8580	8805	6366	8227	6468*	8091	8516		8863
31	778	745	817	741	**719**	711	764*	1027	6150	761	757	1082		721
32	457	444*	411	410*	**422**	447	468	48481	414	472*	453	48076*		412
33	287	41545	3274	7987	**316**	22566	298	42172	3177	7735	324	22407		308
34	254	264	239	224	**243**	286	235	8267	248	236	283	8600		255
35	5717	5722	6744	4897	**12081**	4966	7413	9278	6769	4958	23110	7328		5378

TABLE 4.11: F-Score of test data from ORE 2014

Config#	1	2	3	4	5	6	7	8	9	10	11	12
F-score	94%	84%	94%	92%	96%	92%	88%	92%	92%	93%	94%	91%

equal accuracy, then the priorities are assigned from left to right, e.g., configuration 3 has a higher priority than configuration 4. This is an arbitrary decision that does not have any impact on the overall performance of ML-JFact. The priorities are used to determine which heuristic is a better choice for an input test ontology. The main strategy is to ignore the "Bad" order sets and choose the "Good" order sets. In this case, we end up with three scenarios as shown in Table 4.13.

TABLE 4.12: Priorities of configurations for "Good" labels based on 10-fold cross-validation accuracy

Config#	1	2	3	4	5	6	7	8	9	10	11	12
Accuracy	95%	83%	89%	89%	97%	91%	86%	82%	87%	93%	91%	84%
Priority	2	11	6	7	1	4	9	12	8	3	5	10

An example of Scenario 2 can be seen in sample 5 of Table 4.10 in which the configurations 2,4,6 and 10 are "Good". Since configuration 10 has the highest priority among these configurations from Table 4.12, it will be chosen. As shown in Table 4.10, there are six cases (samples 5, 11, 20, 22, 24 and 27) in which ML-JFact outperformed JFact by one or two orders of magnitude.

Table 4.14 shows the maximum speedup as well as the average speedups of the test cases for ML-JFact compared to JFact. By using a timeout of 1,800,000 milliseconds, the average speedup increases from 11.6 to 39.56 and the maximum from 167.28 to 602.208. Moreover, Table 4.15 shows that the average and sum of the runtimes ML-JFact is 4 to 5 times smaller compared to JFact; we consider this a significant improvement.

TABLE 4.13: Scenarios

Scenarios	**Actions**
1) When there is only one "Good" label among all the configurations (others are "Bad").	The learner chooses the only configuration with the "Good" label.
2) When there are multiple configurations with the "Good" label.	Among the configurations with the "Good" label, the learner will choose the configuration with the highest accuracy/priority (the priorities based on Table 4.12 for Approach I-B and Table 4.19 for Approach II).
3) When all of the configurations' labels are predicted as "Bad" (the classifier will not indicate whether this label has a timeout runtime or it runs in between threshold value and 8 minutes).	The configuration with the lowest accuracy (the priorities based on Table 4.12 for Approach I-B and Table 4.19 for Approach II) will be chosen with the hope for a false "Bad" prediction.

TABLE 4.14: Speedup factor of test data

Speedup ratio	Timeout (500,000 ms)	Timeout (1,800,000 ms)
Maximum	167	602
Average	11.6	39.56

TABLE 4.15: Sum and average runtime (in milliseconds) of all test data with timeout=1,800,000 ms

	ML-JFact	JFact
Sum	2,741,960	13,604,733
Average	78,341	388,707

Discussion

Approach I-B guarantees soundness and completeness of ML-JFact since changing the order in disjunctions does not affect JFacts's behavior.

The runtime overhead for measuring the features varies from 5 to 15 seconds or could exceed this amount for very large ontologies. Since some of these features are related to the `DAG` structure obtained at the beginning of JFact, a small part of this time is due to the interoperation or switching time between Python (machine learning source code) and Java (reasoner source code). However, this time can be reduced in future developments by optimizing the implementation. The other part of the overhead is due to the calculation of metric features related to the ontologies' structure that will take only a couple of seconds for smaller-sized ontologies (with less than 30 MB), but for large-sized ontologies (with more than 30 MB), this could take somewhere between 15 to 50 seconds. The number of very large-size ontologies makes up only 6% of our data.

Besides, almost all of these structural features are the same metrics that are measured[9] at the beginning of Protege[10] framework after loading an ontology. Protege-OWL API is an easy to use GUI environment for ontology reasoning and other applications related to ontologies. It is vastly used by users in different domains that involve ontology operations. The loading time could vary in Protege depending on the size of an ontology. For developing a built-in plugin of the learned-based reasoner for Protege, these features are already measured in Protege while loading ontologies, i.e., there is no need to calculate them again.

[9]The metrics can be found in the metric section in the home display of Protege.
[10]`http://protege.stanford.edu`

In order to increase our set of test ontologies, we also considered $\mathcal{EL}$ ontologies. $\mathcal{EL}$ is a subset of $\mathcal{ALC}$ that allows as concept constructors only $\sqcap$ and $\exists$. However, if a reasoner does not implement $\mathcal{EL}$-specific optimizations, then a subsumption test might internally negate concept expressions. These new expressions are not in the $\mathcal{EL}$ fragment anymore, e.g., conjunctions are transformed into disjunctions. This makes it possible to also consider $\mathcal{EL}$ ontologies for disjunction heuristics. After experimenting on $\mathcal{EL}$ ontologies from the repository, no interesting samples were found due to one of two scenarios: (i) Most of these ontologies run in just a few seconds; (ii) No change with different heuristics is reported for these ontologies. The reason could be due to the rather simple structure of $\mathcal{EL}$ ontologies.

4.4 Approach II: Learning the Right Order of Applying Tableau Rules

We developed an ML-based approach to improve the *ToDo list* optimization technique described in Section 2.3.2. *ToDo list* or ordering of applying expansion rules is a substitute for a traditional top-down approach that had been developed for tableau-based reasoners. The *ToDo list* mechanism allows one to arrange the order of applying different rules by giving each a priority. Our goal is to find the right priority for each rule using the ML-based approach. In the following, the labels and features, as well as the procedure for building an SVM-based classifier are discussed. Our experiments are presented as well.

Labels: Heuristics for the Order of rules

The current default order of applying tableau rules in DL reasoners is mostly based on the exposure to a limited number of tested ontologies [46]. Yet, the effect of different orderings varies from case to case depending on the features of ontologies and cannot be generalized for all ontologies in these reasoners. Moreover, Tsarkov and Horrocks [46] already concluded that there is no universal strategy to ensure the best ordering heuristics of rules in the ToDo list for all types of ontologies. Hence, the best scenario could be an application of an ML-based strategy that considers the features of an ontology in order to predict the best order.

To tackle this as a supervised machine learning problem, the labels need to be identified. Here, each label is considered as an order set that defines the priority of rules in the ToDo list. Having more than two order sets results in a multi-class classification problem with the number of classes equal to the number of possible order sets. Hence, for each input ontology (OWL file), its label is an assigned order set that results in the shortest runtime for that ontology.

Reasoners have a configuration setting that specifies the ordering of rules in the ToDo list (Section 2.3.2). The selected configuration is represented as a string of characters where each character specifies a rule (we adopted JFact's notation here). This string is identified as "IAOEFLG" in which "I" stands for a concept *Id* number, "A" for conjunction ($\sqcap$), "O" for disjunction ($\sqcup$), "E" for existential quantifier ($\exists$), "F" for universal quantifier ($\forall$), "L" for less than or equal ($\leqslant$), and "G" for greater than or equal ($\geqslant$) rules, respectively. Each of these characters is assigned to a number (with 0 being the highest priority)

indicating the associated rule priority, e.g., 0612354 specifies that *Id* has the highest priority over the other rules, then $\sqcup$ has the second priority, and so on.

The number of all possible order sets obtained from assigning numbers to the characters is $7^7 = 823543$ (considering that some rules can have identical priorities). In order to make the problem practical and applicable to the reasoner, the number of order sets needs to be reduced to only a few reasonable and effective orders. Therefore, before prioritizing, some of the rules are combined based on how they affect the reasoning process. *Id* is combined with $\sqcap$, $\geqslant$ is combined with $\exists$ since both are generating rules (they add new nodes to tableaux), $\forall$ is also deterministic but rather simple (and must be applied to all of its successors), i.e., it is singled out, and finally $\leqslant$, $\sqcup$ since both cause non-determinism. This leads us to define only seven order sets as shown in Table 4.16 (rules in the same cells have the same priority). *Id* and $\sqcap$ are ignored and given the first priority because they do not cause non-determinism or expansion, i.e., it is believed that their priority seldom has any effect on the reasoning time.

Structure-based and Statistics-based Features

The features considered for this approach are based on the statistics and structure of ontologies that also include metrics about ratios based on the occurrence of OWL language elements. They are listed in Table 4.17, which shows among other features the ratio of TBox, ABox, and RBox axioms to the total number of logical axioms.

Moreover, to add more useful features, we also added the occurrence ratio

TABLE 4.16: Labels for order sets; the order sets in the table are based on the form of the string "AOEFLG" in which A, O, E, F, L, and G stands for the rules dealing with the concept constructors $\sqcap$, $\sqcup$, $\exists$, $\forall$, $\leqslant$, and $\geqslant$, respectively (in this table we only include tableau rules, i.e., we ignore the first character in "IAOEFLG" denoted by "I"); 0 is the highest and 3 is the lowest priority

Label	order sets	Priority			
		0	1	2	3
1	012312	$\sqcap$	$\leqslant$,$\sqcup$	$\geqslant$,$\exists$	$\forall$
2	013213	$\sqcap$	$\leqslant$,$\sqcup$	$\forall$	$\geqslant$,$\exists$
3	000000	$\sqcap$,$\leqslant$,$\sqcup$,$\geqslant$,$\exists$,$\forall$	NA	NA	NA
4	032132	$\sqcap$	$\geqslant$,$\exists$	$\forall$	$\leqslant$,$\sqcup$
5	031231	$\sqcap$	$\forall$	$\geqslant$,$\exists$	$\leqslant$,$\sqcup$
6	021321	$\sqcap$	$\geqslant$,$\exists$	$\leqslant$,$\sqcup$	$\forall$
7	023123	$\sqcap$	$\forall$	$\leqslant$,$\sqcup$	$\geqslant$,$\exists$

of each concept constructor to all concept constructors. These ratio features are defined below.

$$Ratio_{\leqslant+\forall} = \frac{No_{\leqslant} + No_{\forall}}{No_{\leqslant} + No_{\forall} + No_{\sqcup} + No_{\sqcap} + No_{\geqslant} + No_{\exists}}$$

$$Ratio_{\geqslant+\exists} = \frac{No_{\geqslant} + No_{\exists}}{No_{\leqslant} + No_{\forall} + No_{\sqcup} + No_{\sqcap} + No_{\geqslant} + No_{\exists}}$$

$$Ratio_{\sqcup} = \frac{No_{\sqcup}}{No_{\leqslant} + No_{\forall} + No_{\sqcup} + No_{\sqcap} + No_{\geqslant} + No_{\exists}}$$

$$Ratio_{\sqcap} = \frac{No_{\sqcap}}{No_{\leqslant} + No_{\forall} + No_{\sqcup} + No_{\sqcap} + No_{\geqslant} + No_{\exists}}$$

In addition to the ones mentioned in Table 4.17, we also added as features the occurrence number of different types of OWL object properties including functional, transitive, symmetric, inverse functional object properties.[11] Another feature that we found to be useful in our case due to the importance

[11] `https://www.w3.org/TR/owl2-syntax/`

TABLE 4.17: Basic features for ontologies

Ontology Structural and Statistical Based Features
Number of Existential Value Restriction of Roles
Number of Universal Value Restriction of Roles
Number of Classes
Number of Conjunction Groups
Number of Disjunction Groups
Number of Disjoint Classes
Number of Object Properties
Number of Inverse Object Properties
Number of Nominals
Number of Instances
Number of Role Assertions
Number of Min Cardinality Assertions
Number of Max Cardinality Assertions
Number of Subclasses
Number of Equivalent Classes
Number of Sub Object Properties
Number of Domains
Number of Ranges
Number of Data Properties
Number of Data Properties Assertions
$Ratio_{\leqslant+\forall}$
$Ratio_{\geqslant+\exists}$
$Ratio_{\sqcup}$
$Ratio_{\sqcap}$
Ratio of ABox axioms to all axioms
Ratio of TBox axioms to all axioms
Ratio of RBox axioms to all axioms

of ABox individuals, is the Average Population, which indicates the ratio of concept instances to the number of classes in an ontology [45]. Obtaining some of the features such as the depth of the class hierarchy can be very expensive, especially for big-size ontologies. Moreover, they may not have an impact on changing the order of rules in reasoning tasks.

The reasoning task used in this approach is the hierarchical classification of an ontology, which computes the subsumption relation between named classes of an ontology. The term classification of ontologies is different from the term classification used for machine learning; therefore, we refer to the former as the hierarchical classification of ontologies to avoid any confusion in this thesis.

Since the reasoner's task is to find hierarchical relations in ontologies, to obtain more knowledge from a deeper level of the ontologies, we also considered the number of concept expressions patterns occurring in subsumption ($\sqsubseteq$) and equivalence ($\equiv$) axioms. The patterns for $\sqsubseteq$ are (we use the same patterns for $\equiv$ too):

$$C \sqsubseteq (\sqcup(\ldots)), C \sqsubseteq (\leqslant (\ldots)) \tag{4.7}$$

$$C \sqsubseteq (\sqcap(\ldots)) \tag{4.8}$$

$$C \sqsubseteq (\forall(\ldots)) \tag{4.9}$$

$$C \sqsubseteq (\exists(\ldots)), C \sqsubseteq (\geqslant (\ldots)) \tag{4.10}$$

In addition to that, the number of occurrences of each concept constructor in the parentheses from each of the above patterns will also be considered as separate features. For example, for the pattern $C \sqsubseteq (\sqcup(\ldots))$, we count the

number of occurrences of the $\forall$ concept constructor inside the parentheses.

$$C \sqsubseteq (\sqcup(\ldots, \underline{\forall R.C_1}, \ldots, \sqcup(C_2, \underline{\forall R.C_3}, \ldots), \ldots)) \tag{4.11}$$

The same is applied to other concept constructors for each of the patterns in (4.7)-(4.10).

Procedure: Determining the Right Priority for rules

Similar to Approach I-B, a multi-class classification learner is proposed to learn the best order among the seven order sets[12] for each input ontology. With the small number of training data available for this task, there are seven labels (configurations), which is pretty high number and leads only to a 50% accuracy; therefore, the alternative approach is to consider each order set as a separate binary classification problem and predict for an input ontology its proper order set. The procedure and algorithm to find labels for training data are almost similar to the previous approach (see Algorithm 2).

Moreover, Figure 4.1 shows the procedure of predicting a label for an input ontology. After the features for an input ontology are computed (Step 1 & 2), seven binary classifiers (built from Step 3) predict a "Good" or "Bad" label for the seven configurations of Table 4.16 (Step 4). Later, based on the priorities given in Table 4.19 and the scenarios of Table 4.13, a suitable (sorting) configuration from Table 4.16 is selected for the input ontology. Finally, the reasoner reorders the rules before performing the reasoning task on the input ontology (Step 5).

[12]For simplicity, in this chapter, order sets (heuristics) and configurations are used interchangeably.

Results for Approach II

For this experiment[13] we collected ontologies from ORE 2014.[14] Although the ORE 2014 corpus comprises a large number of ontologies, not all of them can be considered for our dataset. The extracted ontologies from the ORE 2014 library are the ones that trigger at least one rule, that is associated with one of the rules in each of the priority categories shown in Table 4.16. The reason is that the *ToDo list* technique is specifically designed to determine the order of some rules, i.e., the selected ontologies must share the same criteria for the label definitions. Furthermore, this study does not focus on any machine learning problems where the required features are missing. With this strategy, we collect around 1840 OWL files from the ORE 2014 repository.

Furthermore, another decision is required to examine if an ontology among the 1840 OWL files is an appropriate indication of the training data. To make this decision, the reasoning task (hierarchical classification of ontologies) is performed on the 1840 files and the average of three runtimes (in order to obtain reliable runtimes) is calculated for each data on each order set shown in Table 4.16. Then two further conditions are imposed.

Considering a (human) user's point of view, we make a few assumptions about useful speedups. First, if for an ontology the difference between the maximum and minimum runtimes with all seven order sets is less than two seconds, it will be excluded. For instance, let us assume the classification of an ontology takes 3347, 3357, 4537, 2851, 3066, 3408, 2951 milliseconds with the seven order sets. Then, the minimum and maximum runtimes are 2851

[13]The main procedure for approaches II and III are very similar (the main difference is the number of configurations) To avoid some possible confusion, in this section, we may repeat a few details from the previous section.

[14]`http://dl.kr.org/ore2014/ontologies.html`

(belongs to order set 4) and 4537 (belongs to order set 3) and the difference between them is 1686 milliseconds. The difference of less than two seconds could be caused by runtime overhead. To confirm this, we perform several runs; if the biggest and smallest runtimes do not belong to the same configuration in all of the runs, then the difference is indeed caused by a runtime overhead. For such ontologies, this indicates that the rule ordering does not have an impact on ontology reasoning (those ontologies could also be very simple and small). To ensure the homogeneity conditions for the training data (and test data) those ontologies are excluded.

Second, the training data that leads to timeouts for all seven orders are excluded too. The timeout considered for this experiment is 500,000 milliseconds (8.33 minutes). Since obtaining the exact termination time for timeout cases is very expensive, we do not associate them with any runtimes. Therefore, again it is not obvious whether the ontology is affected by any of the rule orderings. The above conditions led to 159 OWL files, of which 75% is used as training data and 25% as test data.

As mentioned previously, if each order set from Table 4.16 is considered as one class, it leads to a multi-class classification model with a high prediction error rate. This is due to the high number of labels without having a large set of training data. Therefore, we take another approach and convert the problem into seven binary classification problems (each order set as an independent machine learning problem) with two labels for each problem. A label "Good" for an order set indicates that the runtime for this order set is below the given threshold value (while performing hierarchical classification) and a label "Bad" indicates that the runtime is above the threshold value (a "Bad" label also includes a timeout case).

TABLE 4.18: Standard deviation and mean values for the threshold for 7 configurations (numbers in milliseconds)

Config#	1	2	3	4	5	6	7
std	94811	65705	86221	74743	69883	86980	69312
mean	56323	30427	42644	33428	33822	45085	30930
std + mean	151134	96132	128865	108171	103705	132065	100242

Similar to Approach I-B, the threshold in this experiment is obtained by exploiting the data distribution for all of the configurations. Table 4.18 shows (in milliseconds) the mean and standard deviation (std) for the training data in each configuration. The maximum value is 151134, which belongs to configuration 1 and the minimum value is 96132, which belongs to configuration 2. The average of the mean plus standard deviation from all of the configurations is considered as the threshold value. This value is 117,187 milliseconds (around 2 minutes).

To build the binary classification models of the configurations, again we use the SVM technique with a 10-fold cross-validation. We conducted a grid search to select the best models by choosing the best parameters such as the numbers of PCA components. Since PCA happens after feature selection, the number of PCA components to grid search over were also considered based on the maximum number of selected features. The optimal kernel used for all configurations except the second one is linear. RBF kernel outperforms the linear one by 10% for configuration 2.

Binary classifiers built for each configuration (order set) have two labels (classes): "Good" and "Bad" indicating if an ontology can be classified in less than 2 minutes or more than 2 minutes including timeouts, respectively. Similar to the previous approach, three main scenarios could occur and the system will react to them as shown in Table 4.13 and based on the priorities for

each configuration given in Table 4.19. Table 4.19 shows the cross-validation accuracy of all configurations after applying standardization and the combination of PCA with the *mutual information feature selection* technique that selects the 40 features with the highest scores.

TABLE 4.19: Priorities of configurations for "Good" labels based on their 10-fold cross-validation accuracy

Config#	1	2	3	4	5	6	7
Accuracy	76%	85%	77%	81%	75%	71%	83%
Priority for "Good" labels	5	1	4	3	6	7	2

Finally, to assess the impact of the built models on JFact, 39 unseen samples from ORE 2014 competition were randomly selected (that also observe the two earlier-mentioned conditions for the training data) and evaluated by their F-score (Table 4.21), which shows the overall success of our built classifier on the 39 test samples.

Table 4.20 shows the runtimes for the 39 samples with different order sets using ML-JFact. However, the last column shows the runtimes for JFact's default configuration which was set by its developers and is deemed to be based on the FaCT++ code [47] but such a configuration might not be adequate for ontologies that are currently available or might be encountered in the future. This indicates that compared to the JFact's default configuration that leads to a timeout for sample 10, ML-JFact runs by 2-3 orders of magnitude faster, respectively. The selected configurations for the samples are shown in bold.

Our approach is very helpful for the samples that have timeouts or run in more than 2 minutes in some order sets and instead one can choose another order set that runs in less than 2 minutes. For example, in Table 4.20,

since for sample 1, the configurations 1,2,3 and 7 are classified as "Bad" order sets, with the built model, ML-JFact will choose among the "Good" labeled configuration instead (configurations 4, 5, and 6).

Table 4.22 shows the maximum and average speedup factors of the ML-JFact compared to the worst case rule order sets, which is by average about two orders of magnitude faster. The speedup ratio of ML-JFact compared to JFact is also shown in the same table.

Discussion

JFact itself is sound and complete. Since changing the order of rules does not impact the reasoner's functionality, soundness and completeness of ML-JFact still hold.

In order to show the significance of ML-JFact, we offer a new way to assess the quality of systems. Our experiments demonstrate that Approach II provides more effectiveness for group reasoning about ontologies when compared to individual reasoning about ontologies. Group reasoning about ontologies here refers to reasoning about ontologies together (either sequentially or in parallel) rather than individually. In order to prove this, the sum of the runtimes of all test cases for JFact compared to ML-JFact is shown in Table 4.23. It shows that the sum for ML-JFact is 344,322 milliseconds less (near 1.5 times faster) than JFact. This may even reach nearly a factor of 2 when the timeout is increased to more than 500,000 milliseconds. Hence, the ML-based strategy is very beneficial when dealing with group reasoning of ontologies. That is usually the case for simultaneously processing a group of ontologies on high-performance computers as well as applications that

TABLE 4.20: *Approach II*: Runtimes (in milliseconds) for the 39 tests chosen from ORE 2014 competition (*TO* indicates a time-out and the configurations selected by ML-JFact are shown in bold; a '*' indicates falsely predicted labels)

Sample	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6	Config 7	JFact Default Configration
1	*TO*	*TO*	*TO*	**40146**	39200	42831	*TO*	40246
2	*TO*	**625**	2050	903	*TO*	*TO*	607	881
3	*TO*	*TO*	*TO*	**15688**	15253	19141*	*TO*	15727
4	10296	**9642**	25926	8617	8761	8875	9451	8467
5	14272	**13862**	52538	15319	15508	16023	13672	11240
6	4212	**4050**	297164*	*TO*	3808*	*TO*	3959	2701
7	*TO*	**417**	1372	412	408	414	407	382
8	167748	**63112**	*TO**	88423	212917	300729	75658	85404
9	*TO*	*TO*	*TO*	**192741***	198465*	210961*	*TO*	188800
10	379	373*	355	*TO**	*TO*	*TO*	**325**	*TO*
11	*TO*	**557**	936	598	*TO*	*TO*	557	595
12	*TO*	*TO*	*TO*	**39500**	38936	41690	*TO*	40160
13	*TO*	*TO*	*TO*	**36773**	36602	*TO*	*TO*	35125
14	*TO*	**14007**	17900	13916	*TO*	*TO*	13882	13942
15	*TO*	**6450**	9905	6647	*TO*	*TO*	6556	8263
16	*TO*	*TO*	*TO*	**42277**	41877	45745	*TO*	43638
17	*TO*	**720**	1171	794	867*	*TO*	677	870
18	*TO*	**773**	2612	856	836	*TO**	675	852
19	*TO*	*TO*	*TO*	**15298**	15626	17728*	*TO*	15269
20	2462	**2946**	3599	1553	1536	4586	2639	1699
21	*TO**	**518**	475	497	469	486	476	589
22	718*	**678**	3958	1620	1857*	1932*	664	1749
23	*TO*	**434**	671	417	413*	*TO*	430	388
24	65579	**3432**	18415	2510*	5156*	25909*	3494	2734
25	*TO**	**457**	471	480	*TO**	*TO**	457	456
26	*TO*	*TO*	*TO*	**324682***	329102*	350004*	*TO*	298338
27	230550	**16057**	178259*	3863	3879	27882	6575	3568
28	3717	**1058**	3454	1187*	1497	4050	1083	1146
29	*TO*	**16105**	*TO**	2928	*TO*	*TO*	16679	2628
30	7858*	6982*	99669*	6436*	**6404**	5748*	7048*	6344
31	327919	**27042**	264452*	4357	4397	52241	8343	3597
32	*TO*	**420**	853	436	424*	*TO*	415	443
33	*TO*	**545**	574	558	610*	*TO*	553	581
34	3489	**919**	2416	931*	1123	3602	910	848
35	*TO*	**449**	791	443	402*	*TO*	429	440
36	*TO**	**979**	6022	1714	*TO**	*TO**	991	1753
37	574	**566**	23380	606	586	569	581	615
38	91032	**96408**	124938*	2296	2325	2350	93088	2405
39	17058*	**3523**	31232	2172	2206	17998*	2469	1925

TABLE 4.21: F-score for all 39 tests chosen from ORE 2014

Config#	1	2	3	4	5	6	7
F-score	76%	96%	86%	90%	75%	71%	96%

TABLE 4.22: Speedup ratio improvement of all 39 test data

	Maximum	Average
ML-JFact to the worst case	1538.45	339.114
ML-JFact to JFact	1538.45	40.37

require to perform reasoning tasks on a group of ontologies together either simultaneously or sequentially.

TABLE 4.23: Sum and average runtimes (in milliseconds) of all 39 test data (ML-JFact vs. JFact)

	ML-JFact	JFact
Sum	1,000,585	1,344,808
Average	25,656	34,482

Although the ML-based strategy never selects a timeout runtime, one could still argue that JFact seems to encounter only one timeout case, which is better than each of the seven orders (Table 4.20), i.e., the default order by JFact is an optimal choice. Nonetheless, we may disagree since the default order (by JFact) is based on a set of very limited experiments [47] and it seems a rather arbitrary choice. To support our concern, we created 56 new random order sets (configurations) in order to compare the number of timeout cases for these random order sets against JFact's timeout cases.

Our analysis showed that for the 56 random order sets, on average 32% of the cases encounter timeouts (the minimum number of timeouts is 10% and the maximum is 72%); however, for the JFact standard configuration, only 3% of the runtimes are timeouts. For the seven order sets defined in Table 4.20, the timeouts make up on average 25% of the cases (minimum 8%

and maximum 55%). This shows our systematically defined order sets (7 order sets in Table 4.20) are mainly generalized from the most possible order sets whereas the default order suggested by JFact seems to rather be a lucky choice with fewer timeouts. Yet, compared to JFact, ML-JFact seems more practical for applications that involve reasoning about multiple ontologies together compared to individual cases in which JFact and ML-JFact seem to perform mostly similar.

Furthermore, if we consider the whole process of classifying ontologies in JFact this task is also accompanied by reasoner tasks such as checking the consistency of ontologies. Depending on the number of nominals (or size of an ABox), the consistency checking before the main hierarchical classification may take time (and even lead to a timeout). However, we believe these ontologies will also take almost the same amount of time for classification. The reason may be that too many nominals indicate a big-size ontology associated with many classes. In order to cover all possible ontologies, we provided both training data and samples that contain ontologies with a large or small number of nominals, or no nominals. Finally, for ML-JFact it usually takes between 5 to 10 seconds to calculate the features of an ontology.

In addition to the test data that was already considered in the experiments, we also included $\mathcal{EL}$ ontologies that do not necessarily trigger all possible rules in the data set. From our experiments, no interesting samples could be reported (either the ontologies ran in just a few seconds or no change was encountered when the heuristic is changed). Regardless of that, it is also worth mentioning that most of the features that influence our learning model are equal to zero for $\mathcal{EL}$ ontologies, i.e., they may have led to low accuracy and possibly not a good representation for the test set.

4.5 Summary

This chapter presented the ML-based approaches developed for improving decision-making in OWL reasoners. The empirical evaluation for each approach is given as well. In the next chapter, the conclusion and future work are given.

Chapter 5

Conclusion and Future Work

Heuristic decision-making in OWL reasoners over many alternatives can be resolved with the help of machine learning that relies on defining proper and correlated features for ontologies. Although ontologies are very complex infrastructures, we showed that the hierarchical classification of ontologies can be improved in order to make proper decisions during the reasoning process. In this thesis, we focused on two sources causing a non-deterministic behavior of OWL reasoners, which can be enhanced with the help of machine learning.

5.1 Summary

In this thesis, we developed three approaches based on machine learning, which are summarized in the following.

- **Approach I-A**: First, we focused on decreasing redundant search space exploration in disjunctions. Approach I-A is based on learning new

orderings for selecting variables at each branching level. Our first algorithm has demonstrated a significant improvement in our sample tests; however, it only accepts propositional description logic as input.

- **Approach I-B**: The extended version of Approach I-A that is Approach I-B focuses on the built-in expansion-ordering heuristics in OWL reasoners. These heuristics determine the order of concepts or subconcepts for satisfiability tests. Determining the right ordering helps to speed up satisfiability testing and many other reasoning tasks that are based on satisfiability testing. Compared to the first approach, this one accepts input beyond propositional logic. Our ML-based approach outperforms by an average of one to two orders of magnitude when the speedup is compared to the recommended default ordering heuristic.

- **Approach II**: Finally, we focused on improving the *ToDo list* approach in order to resolve the unsystematic application of rules in OWL reasoners (Approach II). The order of rules in the ToDo list is likely to have a drastic impact on improving the performance of reasoners. Thus, we learned a model that allows a reasoner to choose adequate heuristics for given ontologies and speed up the computation of classification hierarchies. For this purpose, we presented a learned model that outperforms the worst case ordering selection by one to three orders of magnitude. Moreover, compared to JFact, the ML-JFact is most beneficial when it comes to working with groups of ontologies.

5.2 Future Work

In the following, we survey some of the aspects of this thesis that can be improved for future work.

- **Alternatives for Approach I-A:** As part of our future work, for training purposes, OWL expressions could be treated as propositional logic or QBF, so they can take advantage of available training sets to be solved by QBF, SAT, or SMT solvers. Although, encoding description logic expressions that are beyond propositional logic to the input format acceptable by SAT, SMT, and QBF solvers might yield a solution but could result in the loss of data due to the special input format required by these solvers.

- **More reasoning tasks:** As for future work, more reasoning tasks such as classification could be sped up by considering the combination of expansion ordering heuristics for satisfiability and subsumption testing together.

- **More than two labels:** As of the future work, for both Approach I-B and Approach II, we plan to improve the built model in order to help with not only the right but also probably the best heuristic selection, i.e., the heuristic with the smallest runtime. This could be done by defining more thresholds that lead to determining more than only "Good" and "Bad" labels. We already proposed such an approach by defining a third label, called "Ok", which would be assigned to runtimes between the given threshold and timeout. However, maintaining a high

accuracy model that holds the prediction of more than two labels is difficult. Therefore, building a high accuracy model that can reach this goal is highly appealing.

- **Others:** A learning model may estimate the feature calculation time for very large-sized ontologies and predict whether it is worth for a reasoner to ignore and not apply ML learning techniques on those ontologies. Finally, the developed techniques can be applied to every tableau-based OWL reasoner that can reason about super sets of $\mathcal{ALC}$. Moreover, we yet have to explore what other sources that require optimal decision-making can be enhanced by machine learning in JFact and other similar OWL reasoners.

Bibliography

[1] Alaya, N., Yahia, S. B., and Lamolle, M. (2015a). Towards unveiling the ontology key features altering reasoner performances. *arXiv preprint arXiv:1509.08717*.

[2] Alaya, N., Yahia, S. B., and Lamolle, M. (2015b). What makes ontology reasoning so arduous?: Unveiling the key ontological features. In *International Conference on Web Intelligence, Mining and Semantics*, pages 4:1–4:12.

[3] Baader, F., Buchheit, M., and Hollander, B. (1996). Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1-2):195–213.

[4] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors (2007). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2nd edition.

[5] Baader, F., Hollunder, B., Nebel, B., Profitlich, H.-J., and Franconi, E. (1994). Am empirical analysis of optimization techniques for terminological representation systems. *Applied Intelligence*, 4(2):109–132.

[6] Baker, A. B. (1994). The hazards of fancy backtracking. In *AAAI National Conference on Artificial Intelligence*, pages 288–293.

[7] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. (2013). The MathSAT5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107.

[8] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[9] Cover, T. M. and Thomas, J. A. (1991). Entropy, relative entropy and mutual information. *Elements of information theory*, 2:1–55.

[10] Donini, F. M. and Massacci, F. (2000). EXPtime tableaux for $\mathcal{ALC}$. *Artificial Intelligence*, 124(1):87–138.

[11] Faddoul, J. (2011). *Reasoning Algebraically with Description Logics*. PhD thesis, Concordia University, Montréal, Québec, Canada.

[12] Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E., and Srinivas, K. (2006). The summary ABox: Cutting ontologies down to size. In *International Semantic Web Conference*, pages 343–356.

[13] Freeman, J. W. (1995). *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania.

[14] Gasse, F. and Haarslev, V. (2009). Expressive description logics via SAT: The story so far. In *International Workshop on Satisfiability Modulo Theories*, pages 30–34.

[15] Haarslev, V. and Möller, R. (1999). An empirical evaluation of optimization strategies for ABox reasoning in expressive description logics. In *International Workshop on Description Logics*, pages 115–119.

[16] Haarslev, V. and Möller, R. (2001). High performance reasoning with very large knowledge bases: A practical case study. In *International Joint Conference on Artificial Intelligence*, pages 161–168.

[17] Haarslev, V. and Möller, R. (2008). On the scalability of description logic instance retrieval. *Journal of Automated Reasoning*, 41(2):99–142.

[18] Haarslev, V., Möller, R., and Turhan, A.-Y. (2001). Exploiting pseudo-models for TBox and ABox reasoning in expressive description logics. In *International Joint Conference on Automated Reasoning*, pages 61–75.

[19] Horrocks, I. and Patel-Schneider, P. F. (1998). DL systems comparison. In *International Workshop on Description Logic*, pages 55–57.

[20] Horrocks, I. and Patel-Schneider, P. F. (1999). Optimizing description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293.

[21] Horrocks, I. R. (1997). *Optimising tableaux decision procedures for description logics*. PhD thesis, , University of Manchester Manchester, UK; .

[22] Kang, Y.-B., Krishnaswamy, S., and Li, Y.-F. (2015). r_2 o_2: An efficient ranking-based reasoner for OWL ontologies. In *International Semantic Web Conference*, pages 322–338.

[23] Kang, Y.-B., Li, Y.-F., and Krishnaswamy, S. (2012). Predicting reasoning performance using ontology metrics. In *International Semantic Web Conference*, pages 198–214.

[24] Knublauch, H., Fergerson, R. W., Noy, N. F., and Musen, M. A. (2004). The Protégé OWL plugin: An open development environment for semantic web applications. In *International Semantic Web Conference*, pages 229–243.

[25] Kolb, S., Teso, S., Passerini, A., and De Raedt, L. (2018). Learning SMT (LRA) constraints using SMT solvers. In *International Joint Conference on Artificial Intelligence*, pages 2333–2340.

[26] Kühlwein, D., Schulz, S., and Urban, J. (2013). E-MaLeS 1.1. In *International Conference on Automated Deduction*, pages 407–413.

[27] Lagoudakis, M. G. and Littman, M. L. (2001). Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359.

[28] Li, Z. (2008). *Efficient and generic reasoning for modal logics*. PhD thesis, , The University of Manchester, UK; .

[29] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016). Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140.

[30] Maandag, P., Barendregt, H., and Silva, A. (2012). Solving 3-SAT. Bachelor's Thesis; , Radboud University Nijmegen; .

[31] Maniraj, V. and Sivakumar, R. (2010). Ontology languages-a review. *International Journal of Computer Theory and Engineering*, 2(6):887–891.

[32] Marques-Silva, J. (1999). The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74.

[33] Mehri, R. and Haarslev, V. (2017). Applying machine learning to enhance optimization techniques for OWL reasoning. In *International Workshop on Description Logics*, volume 1879 of CEUR-WS.

[34] Mehri, R., Haarslev, V., and Chinaei, H. (2018). Optimizing heuristics for tableau-based OWL reasoners. arXiv:1810.06617; .

[35] Mehri, R., Haarslev, V., and Chinaei, H. (2019). Learning the right expansion-ordering heuristics for satisfiability testing in OWL reasoners. arXiv:1904.09443; .

[36] Pan, J. Z., Bobed, C., Guclu, I., Bobillo, F., Kollingbaum, M. J., Mena, E., and Li, Y.-F. (2018). Predicting reasoner performance on ABox intensive OWL 2 EL ontologies. *International Journal on Semantic Web and Information Systems*, 14(1):1–30.

[37] Patel-Schneider, P. F. (1998). DLP system description. In *International Workshop on Description Logics*, pages 78–79.

[38] Pham, T. A. L., Le-Thanh, N., and Sander, P. (2008). Decomposition-based reasoning for large knowledge bases in description logics. *Integrated Computer-Aided Engineering*, 15(1):53–70.

[39] Samulowitz, H. and Memisevic, R. (2007). Learning to solve QBF. In *National Conference on Artificial Intelligence*, pages 255–260.

[40] Sazonau, V., Sattler, U., and Brown, G. (2014). Predicting performance of OWL reasoners: Locally or globally? In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 661–664.

[41] Schmidt-Schauß, M. and Smolka, G. (1991). Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26.

[42] Shalizi, C. (2013). *Advanced data analysis from an elementary point of view*. Cambridge University Press.

[43] Sharma, A. and Goolsbey, K. (2017). Identifying useful inference paths in large commonsense knowledge bases by retrograde analysis. In *AAAI Conference on Artificial Intelligence*.

[44] Sirin, E., Grau, B. C., and Parsia, B. (2006). From wine to water: Optimizing description logic reasoning for nominals. In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 90–99.

[45] Tartir, S., Arpinar, I. B., Moore, M., Sheth, A. P., and Aleman-Meza, B. (2006). OntoQA: Metric-based ontology quality analysis. Technical report, Wright State University.

[46] Tsarkov, D. and Horrocks, I. (2005). Ordering heuristics for description logic reasoning. In *IJCAI*, pages 609–614.

[47] Tsarkov, D. and Horrocks, I. (2006). FaCT++ description logic reasoner: System description. In *International Joint Conference on Automated Reasoning*, pages 292–297.

[48] Wandelt, S. and Möller, R. (2012). Towards ABox modularization of semi-expressive description logics. *Applied Ontology*, 7(2):133–167.

[49] Wang, J. (2005). Large ABox store (LAS): database support for TBox queries. Master's thesis, Concordia University.

[50] Wold, S., Esbensen, K., and Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52.