Algorithmic study of Kernel Contraction in $\mathcal{EL}$

by

Amr Dawood

B.Sc. (Hons.), German University in Cairo, 2011

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2017

Approval

Name:	**Amr Dawood**
Degree:	**Master of Science (Computing Science)**
Title:	**Algorithmic study of Kernel Contraction in $\mathcal{EL}$**
Examining Committee:	**Chair:** Fred Popowich Professor
	James P. Delgrande Senior Supervisor Professor
	David G. Mitchell Supervisor Associate Professor
	Kewen Wang External Examiner Professor
Date Defended:	**September 13, 2017**

Abstract

Kernel contraction is an interesting problem that can be considered a step towards belief revision. Kernels were introduced as a tool to determine why a given belief is accepted by the knowledge base. The aim of using kernels is to invalidate the reasons why that given belief is accepted, and hence rejecting that belief. We use Description Logic $\mathcal{EL}$ for two reasons: it is used in some large knowledge base applications, and it has a polynomial-time reasoning algorithm. In this study we introduce an algorithm that performs kernel contraction by reduction to the network-flow problem. We evaluate the rationality of the algorithm by applying postulates that govern kernel contraction. We also explain two heuristics: localization and specificity, that can be used to arrive at more reasonable and common-sense solutions. We will also be focusing on the complexity of the algorithms as an indicator of their feasibility.

Keywords: Kernel Contraction; EL; Description Logic; Specificity; Localization

Table of Contents

List of Figures

Chapter 1

Introduction

Changing one's mind is a process that happens very frequently as part of one's daily routine. Pushing the power button on a very old television that is covered in dust and seeing it getting turned on, one would change their mind and believe that the television is working. They used to think it is broken because it hasn't been used for decades, in which case we say that they believed that "the television is broken". That belief was part of their state of mind. But seeing it work means that there is a contradicting belief "the television works" that needs to be adopted by the mind which implies removing the old belief that "the television is broken". Belief revision refers to the process of modifying the state of mind to adopt a new belief that contradicts existing beliefs.

Mind changing can be thought of as the manipulation of beliefs according to perception. When humans perceive any change in their world, they change their knowledge accordingly. So changing the knowledge (or beliefs[1]) of an agent[2] can be seen as what we informally call changing the mind of the agent. One example of belief change is **Revision**, which involves changing the knowledge base in order to add a conflicting belief. This can actually be broken down into two processes: changing the knowledge base to account for the conflict, and adding the new belief.

The first process involves removing the beliefs that are causing that conflict. This process is called **Belief Contraction**. The second process involves adding the new belief and expanding the knowledge base accordingly. This process is called **Belief Expansion**. Contraction is the type of change we are mainly concerned with in this study. Kernel contraction is one approach to contraction that works by computing all kernels and removing some statements from each of them.

[1]Throughout this study we will be using the terms "knowledge" and "beliefs" interchangeably, although they are not exactly the same. Knowledge is usually assumed to be a special kind of beliefs. However, for convenience, when we use the word "knowledge" we will be referring to "belief."

[2]The word "agent" here means humans, computers, or any thing that has knowledge and perception.

1.1 Kernel contraction by example

The following chapters will explain kernel contraction in more detail, but it is helpful to get a brief idea on what it is before moving on. A kernel is a minimal set of beliefs that imply a certain belief. If our knowledge contains the following beliefs:

- We are in Canada
- The temperature is 30 Celsius
- If temperature is 30 Celsius in Canada, then it is summer,

we can infer that it is indeed summer time. In this case, the three beliefs together form a kernel that implies "it is summer." That kernel can be used for contraction. If we want to give up the belief "it is summer," we can remove a belief from the kernel so that the remaining two are not enough to imply that it is summer. In some other cases we can have more than one kernel, and the same rules can still apply: remove a belief from each kernel in order to perform kernel contraction. The beliefs used in this example are quite subjective, but the aim is to give a brief and simple example. The following chapters will talk more formally about kernel contraction.

1.2 The language

The knowledge of an agent can be represented as a set of beliefs. An agent is assumed to believe in A if A is a member of its *belief set*[3]. In this study, the language used to represent belief sets is *Description Logic*. Description Logic is a family of formalisms that are used to represent knowledge. They use concepts to represent classes of individuals and roles to represent relationships between them. They have different expressive powers and different reasoning mechanisms with different complexities. They vary according to the set of logical operators they use. Here we use $\mathcal{EL}$, which is a member of the Description Logic family.

1.3 Scope of this study

We aim at studying implementations of kernel contraction for knowledge bases represented in the description logic $\mathcal{EL}$. We will look at some basic implementations as well as some advanced ones. Our focus is on investigating the algorithmic aspects of those different approaches. The time complexity will be taken into consideration, and more importantly the optimality of the solutions found. Optimality can be measured in different ways, among them is one that makes more sense as a solution to humans and that is closer to solutions humans generally prefer.

[3]Belief sets will be explained in the next chapter.

The starting point of optimality seeking in this study will be from a syntactic point of view based on the hitting set problem. The hitting set problem is simply the problem of finding the smallest set of elements that intersect with each set of a given set of sets of elements (this will be explained in more details later). We adopt the hitting set approach because what we have is a set of kernels (sets of elements) that we need to remove one element from each. By doing this we hope to achieve syntactical optimality that is only measured by the size of the solution: the smaller the better.

One of the significant contributions of this study is the investigation of algorithms that perform kernel contraction achieving semantic optimality based on $\mathcal{EL}$ semantics. These approaches exploit the structure of $\mathcal{EL}$ knowledge bases to find most reasonable solutions. Both approaches that seek semantic and syntactic optimality can be combined together to achieve better solutions. In contracting big knowledge bases, we can end up with kernels that include beliefs that are not all directly related to each other. In that case, we say that it makes more sense to select beliefs that are related to be removed rather than selecting the beliefs that are not. This idea motivates the heuristic that we call *localization*. Another heuristic we introduce is *specificity*, which is based on the idea that removing beliefs about specific matters is more reasonable than removing beliefs about general ones. We will study these two heuristics, which can be used to make decisions on which beliefs to remove based on the meaning of the logical operators and the structure of the language. These heuristics can be useful in cases where other approaches (such as the minimum hitting set algorithm) result in more than one solution with the same size, to select the solution that is more reasonable based on the meaning of the beliefs. Approaches that only consider the size of the solution do not always make a meaningful choice on which beliefs to remove. So we can then use the heuristics as a tie-breaker step when we get more than one solution.

We start the study by discussing the basic types of belief change, but before that, we build a ground for them by defining the framework that was introduced by Gardenfors. We define epistemic states and attitudes that will help in understanding the mechanics of belief change. We then explain some postulates introduced in the AGM framework; those postulates are considered rationality rules for belief change operations. Then we explain what description logic is and what logical operators are used. In that discussion, we focus on the most relevant variation to this study, which is $\mathcal{EL}$.

After that, we will go over some belief contraction techniques and show how syntactic optimality can be achieved, and what the cost can be (in terms of time complexity.) We will talk more formally about contraction using kernels. Our new approach to kernel contraction is by reducing the problem of contraction to a graph problem. We will give a brief introduction to this new technique and will describe it in detail using a few different examples. We will use the algorithm for network flow problems to produce a solution to kernel contraction that will be guaranteed to be smallest. This algorithm cannot be used for every $\mathcal{EL}$ knowledge base; it only works on knowledge bases in some certain settings.

This will be discussed in more detail later, and the examples will show the cases in which the graph technique is useful and the cases in which it is not. Then, we will turn to the approaches that consider the semantics of the language in finding best solutions to the kernel contraction problem. These heuristics are called *Localization* and *Specificity*. Later, we will see what they are and how useful they can be.

Chapter 2

Background

Learning is one of the most interesting functions of the human mind. If we think of the human memory as a storage of beliefs, we may think of learning as the changing of these beliefs. In that sense, learning something new would correspond to the addition of new beliefs. But learning is not only about new things. We sometimes learn things that contradict other things we learned before. In that sense, learning could be thought of as the update or revision of beliefs.

In this chapter, we aim at explaining the context of the ideas developed in this study, and giving some background of the main building blocks of our work. We will discuss three different ways of changing beliefs and we will use the AGM framework[13] to define and explain these changes. After explaining the AGM rules that govern our approaches to rational belief change, we will explain a Description Logic called $\mathcal{EL}$, which will be used in this study as the knowledge representation language.

2.1 Belief change

The AGM framework [13] is named after Alchourron, Gardenfors, and Makinson, the three scientists who invented it in the 1980s. Since that time, AGM has been the most adopted framework for belief change. In [7], Gardenfors provided some very useful notations to describe the change of beliefs, *epistemic states* and *attitudes*. We can think of the epistemic states as the states of mind (from the beliefs perspective), where epistemic states of an agent are defined by the epistemic attitudes agents have towards the concepts they know about. In this case, changing one attitude towards one concept implies a new epistemic state. Therefore, belief change is simply transitions between different epistemic states. Before moving on to explaining the AGM paradigm, it is useful to shed some light on the notions of epistemic states and attitudes.

2.1.1 Epistemic states

As the word "change" suggests, our concern is about transitions between epistemic states. One can think of an epistemic state as the state of beliefs of an agent (or a human), and of the change as the move from a given state to a new state. The beliefs of an agent can be modelled -as the epistemological theory suggested in [7]- as a set of propositions, with some assumptions on the attitudes towards each of the propositions (these will be discussed in more detail in 2.1.2). Those beliefs are not meant to be psychological propositions expressing beliefs in a human mind; they are epistemological *idealizations* of psychological propositions that a human mind can believe or reject. It is reasonable to consider that every agent should always seek an *equilibrium* state, where the epistemic state is consistent; and if some propositions are contradicting, the agent ought to revise its beliefs to reach a consistent state.

In this study, we use the words "belief sets" and "epistemic state", and they are not exactly the same, although they are related; A **Belief Set** is the set of all things that an agent believes in, while an **Epistemic State** is the complete state of the agent's epistemic attitudes towards all propositions, including the ones that the agent does not believe in. Depending on the epistemological model we follow, we might consider propositions that are not in the belief set to be rejected, or we might assume that the agent neither believes in them nor rejects them.

Another important notion is the *belief base*, which we use to refer to the (possibly incomplete) set of *explicit* beliefs of an agent. It is different from a belief set because we typically assume that belief sets are closed under logical consequence, while belief bases are not. In that sense, we say that the belief set includes some *implicit* beliefs that follow from some explicit ones.

The AGM paradigm will use the notion of belief sets. Later, in 2.2, we will discuss Description Logics (DLs) as formalisms to express knowledge, and start using DL *concepts* to represent belief bases. All following discussions about *contraction* algorithms will use such representation.

2.1.2 Epistemic attitudes

A belief of an agent can be interpreted according to the concepts in an epistemic state. Suppose the concept C is defined as:

$$C = \text{It's Monday}$$

If C exists in the belief set of an agent, we say that the agent believes in C (believes that today is Monday), or *accepts* C [7], which also means that accepting C is part of the epistemic state of the agent. However, *accepting* a belief is not the only attitude an agent can have towards a proposition. An agent can also *reject* a concept C if the negation of that concept is in the belief set (i.e. It is not Monday.) In that epistemic state, we say

that the agent rejects C. It is also possible that an agent stays *undetermined* (or ignorant) about a concept if neither the concept nor its negation is in the belief set (e.g. an agent has no idea whether today is Monday or not.) There can be more attitudes if we consider other models of beliefs such as the probabilistic model, where an agent might have different levels of beliefs. But for the scope of this study we are only interested in the three attitudes: *accept*, *reject* and *undetermined*.

2.1.3 Basic types of change

The AGM framework defines three types of belief change: *revision*, *expansion*, and *contraction*. If one sees a flying penguin, it introduces a new belief (penguins might fly), which contradicts another belief people these days have: penguins can't fly. In this case, just accepting the new belief will make the belief system of the agent (in this case, a human) inconsistent, since the two beliefs are contradicting. So revising the old beliefs, and possibly removing the belief that penguins can't fly, is the rational action humans tend to take. This is referred to as belief revision.

When we add a new belief without removing old beliefs, we call this process *belief expansion*. When a belief is removed, or given up, without accepting a new belief, we call this process *belief contraction*. A good example of belief contraction is to give up a belief for the sake of argument, to reach a common ground with an opponent, and argue based on this ground.

Before going further into details of those types of change of belief, some important notions need to be explained.

belief set We represent the beliefs of an agent by a set of sentences. The belief set is closed under deductive inference; if K is a belief set, then $K \models \alpha$ if and only if $\alpha \in K$.

Given a belief set K and a sentence A, we say $K \models A$ if and only if A is a consequence of the belief set K. We refer to the set of all consequences of K by $Cn(K)$. Since K is a belief set, then it is closed under deduction. Hence, $K = Cn(K)$, where $Cn(K)$ is the deductive closure of K. For any belief set K and a sentence A, we can represent the three aforementioned epistemic attitudes as follows [7]:

- If $A \in K$, A is *accepted* and $\neg A$ is *rejected*.
- If $\neg A \in K$, A is *rejected*, and $\neg A$ is *accepted*.
- $A \notin K$ and $\neg A \notin K$ means that A is *undetermined*.

We can think of belief change about A as a change of the attitude towards A. Because we have three attitudes, we have six possible changes. However, because of the symmetry of some pairs of change we only consider three types of change:

1. Change from being *undetermined* to either *accepting* or *rejecting* A.

2. Change from *accepting* to *rejecting* or vice versa (*revision*).

3. Change from *accepting* or *rejecting* A to being *undetermined* (*contraction*)[7].

Belief Expansion

As part of the learning process and gaining knowledge an agent expands its belief set by *accepting* new beliefs. If K is a belief set and A is a statement, then we denote by K_A^+ the new belief set resulting from accepting A. So the expansion function $+$ is a function that takes a belief set K and a statement A and returns another belief set K_A^+.

Belief Contraction

Contraction is different from expansion. Expansion by A is the change from the state of not accepting nor rejecting it to the state of accepting it, while contraction is the change from accepting or rejecting A to being undetermined of it. We denote the belief set resulting from contracting the belief set K with respect to sentence A by K_A^-. Since contraction is the most relevant change to the scope of this study, we need to discuss some of the rationality criteria (or postulates) of such change. We consider the contraction function $-$ as a function from a set of beliefs K and a sentence A to a new set of beliefs K_A^- as in [7] and [13].

In the AGM framework, the following eight postulates ((K$-$1) to (K$-$8)) are introduced as basis for ensuring the rationality of contraction functions (more details can be found in [13] and [7]):

(K$-$1) For any sentence A and any belief set K, K_A^- is a belief set.

K_A^- is obtained by removing some beliefs. So

(K$-$2) $K_A^- \subseteq K$.

If A is not already in the belief set, the contraction should not change the belief set.

(K$-$3) If $A \notin K$, then $K_A^- = K$.

Unless A is logically valid, A should not be in the belief set after contraction.

(K$-$4) If not $\vdash A$, then $A \notin K_A^-$.

The main idea of contraction is to give up some belief. If we need to give B up we can remove it from the belief set, and make sure it is not derivable from the remaining beliefs. To do that we need to look for statements that together entail B and remove at least one of them. This will be elaborated on more in chapter 3 when we discuss kernels. So we do not just remove A from K, but remove the minimum number of sentences that would entail

A. We remove the "minimum" number of sentences because it is better to give up as little as possible (minimum change will be explained in more detail in chapter 3). Also, while performing expansion with A we compute all possible consequences that can be reached after accepting A. Therefore, all beliefs are recoverable by expansion after contraction:

(K−5) If $A \in K$, then $K \subseteq (K^-_A)^+_A$.

If A and B are equivalent sentences, then contracting K for the belief A should result in the same belief set as contracting it for B.

(K−6) If $\vdash A \leftrightarrow B$, then $K^-_A = K^-_B$.

If we contract K by $A\&B$, we should contract K by either A or B, but nothing else.

(K−7) $K^-_A \cap K^-_B \subseteq K^-_{A\&B}$.

Motivated by the concept of minimal change, the last postulate states that if A was removed while contracting K by $A\&B$, then at least A was given up – possibly along with B.

(K−8) If $A \notin K^-_{A\&B}$, then $K^-_{A\&B} \subseteq K^-_A$.

These eight postulates (K−1) – (K−8) are defined for contracting belief sets. However, in this study we are investigating the contraction of belief bases.

In [8], Hansson declared that an operator for A is a kernel contraction (kernels will be explained later) if and only if it satisfies the postulates of *success*, *inclusion*, *core-retainment*, and *uniformity*. We are discussing these postulates because they are more applicable in our investigation of kernel contraction than the previous eight postulates. They are explained in [8] as follows (the symbol $\div$ denotes contraction, e.g. K^-_A is equivalent to $K \div A$):

Success If $\alpha \notin Cn(\emptyset)$, then $\alpha \notin Cn(A \div \alpha)$.

This is similar to **(K − 4)**.

Inclusion $A \div \alpha \subseteq A$.

This is similar to **(K − 2)**.

Core-retainment If $\beta \in A$ and $\beta \notin A \div \alpha$, then there is a set A' such that $A' \subseteq A$ and that $\alpha \notin Cn(A')$ but $\alpha \in Cn(A' \cup \{\beta\})$.

This implies that only sentences that contribute to making A imply α can be removed during contraction, and nothing else.

Uniformity If it holds for all subsets A' of A that $\alpha \in Cn(A')$ if and only if $\beta \in Cn(A')$, then $A \div \alpha = A \div \beta$.

This is similar to **(K − 6)**.

Belief Revision

Revision is considered to be *non-monotonic*. During revision, new beliefs are added with -possibly- some of the old beliefs removed. Revision takes place when the attitude is changed from accepting to rejecting or from rejecting to accepting. We can think of revision as a combination of both contraction and expansion. The revision function $*$ is a function from a belief set K and a sentence A to a belief set K^*_A. If $-A \in K$, then $K^*_A = (K^-_{-A})^+_A$.

In the next section we look at a class of languages called Description Logic that is being used in some applications to represent knowledge. We use it as a model and implement contraction in a DL-based knowledge bases.

2.2 Description logic

One of the earliest approaches to representing knowledge is using logic. Logic has been suitable for general-purpose applications. Another approach is what is so-called semantic networks, which is a graph (or a network)-based approach. Semantic networks use graphs to represent knowledge [4], where nodes represent *concepts* and edges represent *relationships* between them.

Description Logic (DL) comes as an evolution of semantic networks to a logic-based approach with some new flavours. DL is a class of logics based on -as the name suggests- describing sets of individuals using *concepts* and describing the relationships between them using *roles*. Usually, two types of knowledge need to be represented: *intensional* and *extensional* knowledge. Intensional knowledge is general knowledge about a problem or a domain, while extensional knowledge is knowledge about a specific problem instance. For this purpose, a knowledge base[1] is composed of two main components: Terminological Box, which we will refer to by *TBox*, and Assertions Box, which we refer to by *ABox*.

To get an idea about what a Description Logic looks like, we look at two categories of symbols: logical and non-logical symbols. **Non-logical** symbols include the following:

Concepts are similar to category nouns (e.g. *Human*, *Mother*, *Animal*, *College*, etc.). They are used to represent classes (or sets) of individuals. So we can use the concept *Animal* to refer to the set of animals, and the concept *Man* to represent the class of men.

Roles are like relational nouns (e.g. *MotherOf*, *HeightOf*, etc.). They can be used to specify attributes of concepts.

Constants are used to represent individuals, and they are similar to proper nouns, e.g. *Adam*, *Sally*, etc.

[1]By the word *Knowledge base* we refer to a belief base – a set of sentences that represent the belief of an agent.

Concepts and roles can be used with the help of some logical symbols to construct complex expressions. **Logical** symbols include the following:

- $\sqcap, \sqcup, \neg$ used as propositional constructors (conjunction, disjunction, and negation respectively).
- $\forall, \exists$ used for restriction and quantification.
- $\top, \bot$ ($\top$ represents the set of all individuals, while $\bot$ represents the empty set of individuals).

Logical and non-logical symbols can be used together to construct complex expressions. To get a sense of how to build complex expressions, suppose C and D are concepts, and R is a role:

- $C \sqcap D$, $C \sqcup D$, and $\neg C$ are concepts (also called concept expressions).
- $\forall R.C$, and $\exists R.\top C$ are concepts (or concept expressions).

2.2.1 ABox

As we said at the beginning of this section, to represent knowledge we usually look at intensional and extensional knowledge. DL knowledge bases usually consist of two main components: TBox and ABox. ABoxes are built from assertions (extensional knowledge) about a specific problem or domain, e.g.

Girl(Sally) may represent the fact that *Sally* is a *Girl* and,
FatherOf(Adam, Sally) may represent the fact that *Adam* is the father of *Sally*.

where *Sally* and *Adam* are constants, *Girl* is a concept name, and *FatherOf* is a role name. Using such representation, ABoxes can be built to represent assertions about a problem or a domain.

2.2.2 TBox

Suppose that *Human* is a concept that refers to all humans, and *Male* is the concept that refer to all male beings. We can use conjunction ($\sqcap$) in

$$Human \sqcap Male$$

to represent all individuals that are both humans and male beings. We can also define a new concept Man to represent those individuals by introducing the definition

$$Man \doteq Human \sqcap Male$$

So every man has to satisfy that definition. The $TBox$ is composed of concept definitions and General Concept Inclusion rules (GCIs). GCIs are weaker than definitions; the rule

$$Man \sqsubseteq Human$$

states that *Man* is subsumed by *Human*, which means that every man is a human[2]. Every definition can be safely broken down into two GCIs. For example, the definition

$$Man \doteq Human \sqcap Male$$

can be broken down into the two GCIs:

$$Man \sqsubseteq Human \sqcap Male$$
$$Human \sqcap Male \sqsubseteq Man$$

Tboxes are used to represent general knowledge (intensional knowledge) about a class of problems or domains, using axioms (or terminologies). A typical TBox is composed of definitions and GCIs – sometimes for convenience all definitions are broken down into GCIs. The following is an example of a -somewhat incomplete- DL TBox:

1. $Man \doteq Human \sqcap Male.$
2. $Woman \sqsubseteq Human \sqcap \neg Man.$
3. $Father \doteq Man \sqcap \exists ParentOf.\top$
4. $Mother \doteq Woman \sqcap \exists ParentOf.\top$
5. $FatherWithoutSon \doteq Father \sqcap \forall ParentOf.\neg Man.$
6. $Parent \doteq Father \sqcup Mother$
7. $GrandFather \sqsubseteq Father \sqcap \exists ParentOf.Parent$

They can be interpreted such that #1 defines Man to be a human male, #2 states that every $Woman$ is a human and not a man, #3 defines $Father$ to be a man that is a parent of something (since $\top$ includes everything), #4 defines $Mother$ similarly, #5 defines a $FatherWithoutSon$ to be a father which every individual that is in a "$ParentOf$" relationship with is not a man, #6 defines a $Parent$ to be a father or a mother, and #7 states that every $GrandFather$ is a father and in a "$ParentOf$" relation ship with a parent.

Along with some other symbols and constructors, these symbols are the building blocks of Description Logics. There are many members of the DL family, that vary in their expressive power and the complexity of their reasoning algorithms. Each member of the family includes a subset of the DL symbols, and is uniquely identified by the containment of those symbols. In the following section we discuss a very famous member of the DL family, $\mathcal{EL}$, which we use as a knowledge representation language for the rest of the study.

[2]or more precisely, every member of the set represented by *Man* is also a member of the set represented by *Human*.

2.2.3 $\mathcal{EL}$ language

One of the DLs that has recently attracted much attention is $\mathcal{EL}$. $\mathcal{EL}$ is a light-weight (contains a small set of logical operators) Description Logic, though it is used in some well-known ontologies such as SNOMED CT [3], which is a medical ontology that contains around 380000 concepts. $\mathcal{EL}$ only contains a subset of the concept constructors that we discussed in the previous section, and they are:

- The conjunction symbol $\sqcap$
- Existential restriction $\exists$
- The top concept $\top$

One of the advantages of $\mathcal{EL}$, besides being simple and easy to use, is its polynomial-time subsumption problem. The subsumption problem, which is the most important problem in $\mathcal{EL}$, is actually a classification problem. The subsumption algorithm classifies the $TBox$ depending on the subsumption relation expressed by $\sqsubseteq$. The problem is checking whether a specific subsumption relation holds or not (e.g. whether $C \sqsubseteq D$ holds or not) in a given knowledge base. Checking whether a subsumption relation holds or not is also checking whether the concept (or concept expression) on the left-hand side of the subsumption relation is subsumed by the concept on the right-hand side.

It is helpful to sketch the subsumption algorithm before moving on to the core of this study, as it will be used later on. But we will use an example to see how the algorithm can be applied. Let's assume we have a knowledge base TBox $\mathcal{T}$ that consists of only two subsumption relations:

$$\mathcal{T} = \{\ \text{Haddock} \sqsubseteq \text{Fish},\ \text{Fish} \sqsubseteq \text{Animal}.\ \}$$

The first one states that every Haddock is a Fish, and the second one states that every Fish is an Animal. Now, given that $\mathcal{T}$, we need to check that every Haddock is an Animal:

$$\text{Haddock} \sqsubseteq \text{Animal}.$$

This is not explicitly stated in $\mathcal{T}$, but we can infer it using the subsumption algorithm. The algorithm proceeds in four steps:

1. Normalize the TBox.
2. Translate the normalized TBox into a graph.
3. Complete the graph using completion rules.
4. Read off the subsumption relationships from the normalized graph.

We can follow the four steps and apply them one by one to our example and get the solution that we need.

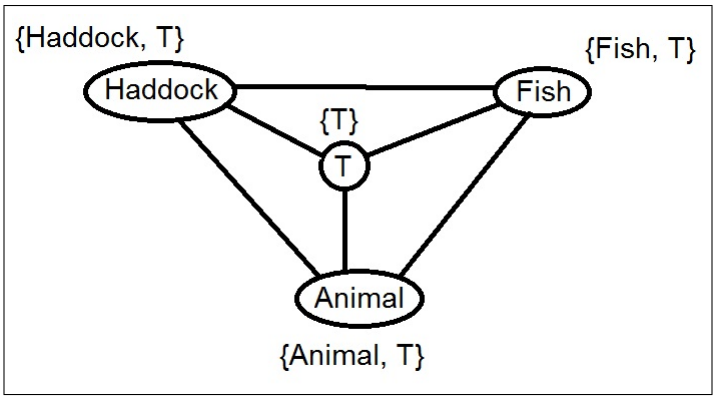

Figure 2.1: Graph transformation of the TBox

Normalize the TBox

We call a TBox normalized if all the GCIs it contains are of one of the following forms:

- $A_1 \sqcap A_2 \sqsubseteq B$
- $A \sqsubseteq \exists r.B$
- $\exists r.A \sqsubseteq B$

where A, A_1, A_2, and B are concept names, and r is a role name. A TBox can be normalized in polynomial time with respect to its size. $A \sqcap \top$ is equivalent to A. So, the normalized version of the $\mathcal{T}$ look like:

$$\{ \text{ Haddock} \sqcap \top \sqsubseteq \text{Fish, Fish} \sqcap \top \sqsubseteq \text{Animal } \}.$$

Translate the normalized TBox into graph

The second step is to translate the TBox into a graph. This is done by creating a node that corresponds to each concept in the TBox (including $\top$), and having an edge between every pair of nodes. We use S to denote the set of nodes' labels, and R to denote the set of roles' labels. S will contain the subsumers of a node, e.g. $S(A)$ contains B if the TBox contains $A \sqsubseteq B$. Likewise, $R(A, B)$ contains r if $A \sqsubseteq \exists r.B$ is in the TBox. Initially, the set S starts by containing the nodes and $\top$, i.e. $S(Haddock) = \{Haddock, \top\}$, $S(Fish) = \{Fish, \top\}$, and $S(Animal) = \{Animal, \top\}$. The set R will initially be empty. The graph can be visualised in Figure 2.1. Translating the TBox into a graph can be done in polynomial time.

Complete the graph using completion rules

The following three rules are then used to extend the sets S and R:

- If $(A_1 \sqcap A_2 \sqsubseteq B) \in \mathcal{T}$ and $A_1, A_2 \in S(A)$ then add B to $S(A)$

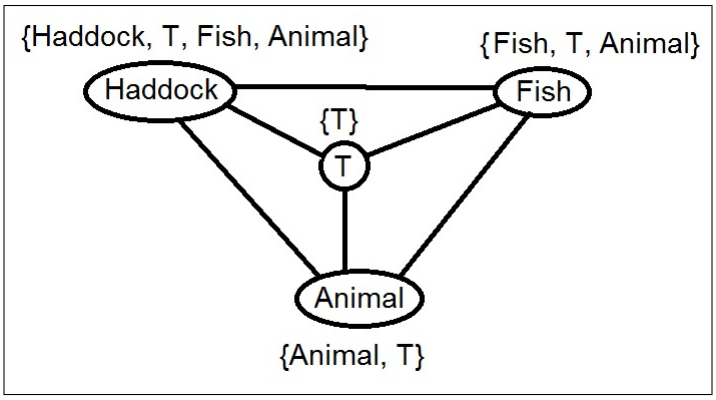

Figure 2.2: Completed graph of the TBox

- If $(A_1 \sqsubseteq \exists r.B) \in \mathcal{T}$ and $A_1 \in S(A)$ then add r to $R(A, B)$
- If $(\exists r.B_1 \sqsubseteq A_1) \in \mathcal{T}$ and $B_1 \in S(B), r \in S(A, B)$ then add A_1 to $S(A)$

If we apply the completion rules to $\mathcal{T}$, S will become as follows:

- $S(Animal) = \{Animal, \top\}$
- $S(Fish) = \{Fish, \top, Animal\}$
- $S(Haddock) = \{Haddock, \top, Fish, Animal\}$

The new graph (after the completion rules are applied) is shown in Figure 2.2.

Read off the subsumption relationships

Now that the graph is complete, we can look at the sets S and R and determine whether the subsumption relationship in question holds or not. Since that subsumption relationship was $Haddock \sqsubseteq Animal$, we can look at $S(Haddock)$ and see if it contains $Animal$ or not, since the set $S(Haddock)$ contains all subsumers of $Haddock$. So, indeed, $Haddock \sqsubseteq Animal$ holds. This brief example hopefully shows a simple application of the subsumption algorithm of $\mathcal{EL}$. We didn't discuss the complexity of the algorithm in details, but it is enough to say it is polynomial in the size of the TBox. More details on the algorithm can be found in [5] and [3].

Now that we have had a quick introduction to the basic types of belief change, the framework of AGM, and the $\mathcal{EL}$ language and reasoning algorithm, we can start talking about the core of this study, which is kernel contraction. Throughout this study, we only consider DL knowledge bases represented in $\mathcal{EL}$. In the next chapter we look at an implementation of belief contraction using *kernels* as introduced in [12]. We will discuss some basic and general approaches as well as some more sophisticated ones. Then we will consider a language-specific approach to exploit the structure of $\mathcal{EL}$ and discuss heuristics that will provide more meaningful solutions.

Chapter 3

Kernel contraction

Belief contraction is the process of removing beliefs from belief bases. In Chapter 2, we discussed the difference between belief bases and belief sets. In this study, we perform kernel contraction on belief bases. It can be done in either of two ways. The first way is computing subsets of the belief base that do not logically imply the belief to be removed. This is called the *remainder set* approach. It works by finding the maximal subsets of the belief base that do not imply the removed belief. For example, the knowledge base:

$$\{a, a \to b, b, c\}$$

has two maximal subsets that do not logically imply b:

$$\{a, c\}, \{a \to b, c\}$$

So we can choose one of them (or, in the general case, the intersection of more than one[1]) to be the new (contracted) belief base.

The other approach is done by incising all the minimal subsets that logically imply the removed belief; by "incising" we mean removing a member from the set. This second approach is the one we are going to focus on in this study. If a set $\mathbf{K}$ of beliefs implies a belief α, and $\mathbf{K}$ is the minimal such set (no proper subset of $\mathbf{K}$ implies α), removing one element of $\mathbf{K}$ produces a set that does not imply α. Contraction is done here by finding all such minimum subsets and removing an element from each of them. We call such minimal sets *kernels*. Kernels will be discussed in more detail in section 3.1. We will look at an example first.

Given the same set:

$$\{a, a \to b, b, c\}$$

To contract the set by b, it is not enough to remove b from the set. If we do so, we get:

$$\{a, a \to b, c\}$$

[1]This is called *partial meet contraction.*

which still implies b. We need to prevent the resulting set of beliefs from implying b. The minimal subset that implies b is

$$\{a, a \rightarrow b\}$$

We call it a b-kernel. Since the kernel is a minimal subset that implies b, removing only one element of that kernel is enough to make it not imply b. The only other b-kernel of the original set is $\{b\}$; we can only break this kernel by removing b. Breaking the other kernel can be done either by removing a, or by removing $a \rightarrow b$. So the two possible resulting belief bases after contraction are:

$$\{a, c\} \text{ and } \{a \rightarrow b, c\}$$

This is how kernel contraction works. Every b-kernel is one way to infer b. To contract b, we pick an element from each kernel and remove the picked elements from the original belief set.

Kernel contraction can be composed of two main steps: computing kernels, and cutting them. In this chapter, we will discuss kernels and see how they can be defined formally. We will discuss briefly how to compute kernels using an algorithm called *pinpointing*. Assuming we have a set of kernels that we computed, we will discuss previous work done on cutting kernels that will give us a good basis for introducing the new approaches afterwards. Then we will discuss the hitting-set approach to contraction in the context of the *minimal change* requirement. The last topic to be discussed in this chapter will be our new graph approach to kernel contraction. The algorithm will combine the two steps of kernel contraction (computing and cutting). The algorithm description will be followed by some analysis and comparisons between different examples using it.

3.1 What are kernels?

Kernel contraction was introduced by Hansson [9] as a variant of an older approach called "safe contraction"[2]. In both approaches, contracting a knowledge base $\mathbf{K}$ by α is done by discarding beliefs that contribute to making $\mathbf{K}$ imply α. Beliefs that contribute to making $\mathbf{K}$ imply α are members of some α-kernel of $\mathbf{K}$, and all members of every α-kernel contribute to making $\mathbf{K}$ imply α. In the previous example, there were two b-kernels: $\{b\}$ and $\{a, a \rightarrow b\}$. We need to remove at least one element from each kernel to contract the original set by b. We denote the set of α-kernels of $\mathbf{K}$ by $\mathbf{K} \perp \alpha$.

Definition 3.1. (Kernel Set [8]) For a belief base $\mathbf{K}$ and a belief α, $\mathbf{K} \perp \alpha$ is the set such that $X \in \mathbf{K} \perp \alpha$ if and only if:

1. $X \subseteq \mathbf{K}$,
2. $X \vdash \alpha$, and

3. if $Y \subset X$ then $Y \nvdash \alpha$.

Because of the minimality of kernels, every belief in a kernel is significant to implying the belief that we want to give up (α). And that's why removing one belief from a kernel is sufficient to make that kernel not imply α. So, we use a function that cuts every kernel in the kernel set. We call such function an *incision function*; it takes a set of kernels and selects an element from each kernel to be removed.

Definition 3.2. (Incision function [9]) An incision function σ for A is a function such that for all beliefs α:

1. $\sigma(A \perp \alpha) \subseteq \bigcup(A \perp \alpha)$
2. If $\phi \neq X \in A \perp \alpha$, then $X \cap \sigma(A \perp \alpha) \neq \phi$

Contraction is done by removing the beliefs that are selected by the incision function from the original knowledge base. Contraction $\approx_\sigma$ using the incision function σ can be defined as:

Definition 3.3. [8] (Contraction)

$$A \approx_\sigma \alpha = A \setminus \sigma(A \perp \alpha)$$

3.2 Computing kernels

Kernel contraction is about contracting knowledge bases using kernels. Kernels are the minimal subsets of the knowledge base that imply a certain consequence. Contraction can be performed by using an incision function to cut through all kernels of a specific belief. So, the first step in kernel contraction is computing all kernels that imply the belief that needs to be removed. For that purpose, we use the pinpointing algorithm introduced in [6].

To show how the algorithm works, we use the example introduced in [6]. Given the following $\mathcal{EL}$ *TBox* $\mathcal{T}$:

$$\mathcal{T} = \{ax_1 : A \sqsubseteq \exists r.A, \quad ax_2 : A \sqsubseteq Y, \quad ax_3 : \exists r.Y \sqsubseteq B, \quad ax_4 : Y \sqsubseteq B\},$$

we can see that $A \sqsubseteq B$ holds according to $\mathcal{T}$, i.e. $A \sqsubseteq_\mathcal{T} B$. Let:

$$\alpha = A \sqsubseteq B.$$

According to the definition of kernel sets:

$$\mathcal{T} \perp \alpha = \{\{ax_2, ax_4\}, \{ax_1, ax_2, ax_3\}\}$$

The algorithm introduced in [6] computes all kernels using a modified version of the $\mathcal{EL}$ subsumption algorithm. It works by finding a monotone boolean formula called the *"pinpointing formula"*. The propositions of the pinpointing formula are GCIs of the *TBox*, and

a propositional *valuation* represents a kernel. In that sense, a valuation is a set of propositional variables that satisfy the formula, and these variables are the GCIs that constitute a kernel. So, by finding all valuations of the pinpointing formula we can get all kernels.

In the worst case, this approach takes exponential time (w.r.t the size of the *TBox*) to find all kernels. This is the case when there are exponentially many kernels. However, [6] also gives a polynomial-time algorithm that computes only one kernel. Because it is not part of the scope of this study, we are not going to discuss and analyze details of the pinpointing algorithm. All that matters is the worst-case time complexity, as it will affect the complexity of the kernel contraction algorithm that uses it.

3.3 Previous work

We now know we can use the pinpointing algorithm to get the set of all kernels. We then need to remove one element from each kernel to perform contraction. This is the role of the incision function; it picks an element from each kernel so that it cuts through all kernels. In this section, we look at some of the incision function implementations discussed in [12]. The goal of this study is to continue the work started in [12], and to revise some of what has already been done.

Given an $\mathcal{EL}$ *TBox* $\mathcal{T}$ and a belief α, $(\mathcal{T} \perp \alpha)$ is the set of α-kernels. The main contraction algorithm is shown in Algorithm 1.

Algorithm 1 Contraction algorithm

```
1: procedure CONTRACT(𝒯, A)
2:     kernelset = PINPOINT(𝒯, A)
3:     giveUpSet = CUT(kernelset)
4:     𝒯 = 𝒯 / giveUpSet
5: end procedure
```

Here, "kernelset" refers to $(\mathcal{T} \perp \alpha)$, and *giveUpSet* is the set of beliefs selected by the incision function (CUT) to be removed. We use the function called CUT as a placeholder for any implementation of the incision function. The algorithm is straightforward. It works by generating the set of kernels using the pinpoint formula as discussed in the previous section. Then it calls the incision function, which picks up beliefs from every kernel, ensuring that it hits all kernels. The beliefs are then removed from the knowledge base, i.e. from the *TBox*.

This general approach can be used with different incision functions. The call to the CUT function can be replaced with a call to another implementation of the incision function. The first implementation is the most straightforward one, where it removes a random belief from each kernel. This is given in Algorithm 2.

The time complexity of RandomRemove function is polynomial: $O(m)$, where m is the number of kernels (size of the kernelset), assuming that the random selection takes constant time. However, this is clearly not a good algorithm. In this example:

Algorithm 2 Random removal

```
1: function RandomRemove(kernelset)
2:     giveUpSet = {}
3:     for kernel ∈ kernelset do
4:         choose a random belief α from kernel
5:         giveUpSet = giveUpSet ∪ {α}
6:     end for
7:     return giveUpSet
8: end function
```

$$\text{kernelset} = \{\{a, c\}, \{b, c\}\},$$

one of the possible outcomes of the RandomRemove algorithm is:

$$giveUpSet = \{c, b\}$$

which unnecessarily removes b. This could happen when the algorithm selects c from the first kernel and b from the second kernel. This solution:

$$giveUpSet = \{c\}$$

seems more concise and removes fewer beliefs. The second solution could be obtained with an algorithm smart enough to check if the next kernel has already been incised.

The next algorithm (Algorithm 3) does this. Every time it selects a belief, it marks all kernels that contain that belief so that in the following iterations they are not incised.

Algorithm 3 Random removal with exclusion

```
1: function RandomRemoveAndExclude(kernelset)
2:     giveUpSet = {}
3:     for i=0 to size(kernelset)-1 do
4:         kernelset[i].valid = true
5:     end for
6:
7:     for i=0 to size(kernelset)-1 do
8:         if kernelset[i].valid then
9:             choose a random belief α from kernelset[i]
10:            giveUpSet = giveUpSet ∪ {α}
11:            for j=i to size(kernelset)-1 do
12:                if kernelset[j].contains(α) then
13:                    kernelset[j].valid = false
14:                end if
15:            end for
16:        end if
17:    end for
18:    return giveUpSet
19: end function
```

Algorithm 3 has a worst-case time complexity of $O(n^2)$, where n is the size of the kernel set, assuming that kernels are stored as hash tables so "kernelset[j].contains(α)" takes constant time. If the kernels are stored as ordered lists, the time complexity would increase to become $O(n^2 \cdot \ln m)$, where m is the size of the largest kernel. If kernels are unordered lists, the time complexity is $O(n^2 \cdot m)$. The second main loop has another nested loop inside it, and neither of them takes more than n steps to finish.

3.3.1 Greedy Contraction

The greedy approach works by selecting for removal the beliefs that appear in more kernels to be before those that appear in fewer kernels. At each step, the algorithm finds which belief appears the most in kernels and removes it, and then it forgets about those kernels (it does not consider those kernels that contain the removed belief for the following steps). This means that if a belief is selected for removal, all the kernels that contain it are already incised and no more beliefs need to be removed from these kernels.

Algorithm 4 explains how the greedy approach for contraction works. Given a set of kernels and the set of all beliefs (the knowledge base), it proceeds as follows: it computes the number of occurrences of each belief in the kernels given, then it removes them one by one starting from the beliefs with maximum number of occurrences, until all kernels are incised. It does not actually remove the beliefs, but it collects them in the giveUpSet that will be returned in line 3 of Algorithm 1, which in turn will remove them from the knowledge base. So the greedy approach can be embedded in the main algorithm as a way to choosing the beliefs that need to be removed.

The worst-case time complexity of the greedy algorithm is lower-bounded by the complexity of the "while" loop in the selectMaxBeliefs() function. If the size of beliefs is m, the size of kernelset is n, and the size of the biggest kernel is k, then the worst-case time complexity is $O(n^2 \cdot k \cdot m)$. This could be significantly improved by using more clever data structures to make the invalidation of the kernels (the step where we decrement the number of valid kernels and update the occurrences) faster.

3.4 Minimal change

According to the information economy principal (minimality), in every change of the epistemic state, loss of information should be minimum [14]. This means that a system should choose an epistemic change outcome that minimizes loss of information. To satisfy the requirement of the minimum change we need an algorithm that removes the least number of GCIs while hitting all the kernels; but the kernels are nothing but sets of beliefs (GCIs). Luckily, this is exactly the *minimal hitting set problem*, which already has some relatively efficient algorithms that we can use here.

Algorithm 4 Greedy Selection

```
1: function GreedyContract(kernelset, beliefs)
2:     initializeValidKernels(kernelset)
3:     initializeBeliefOccurrences(beliefs)
4:     computeOccurrences(kernelset, beliefs)
5:     return selectMaxBeliefs(kernelset, beliefs)
6: end function
1: // Mark all kernels as not cut to keep track of the remaining kernels to be cut
2: function initializeValidKernels(kernelset)
3:     validKernels = size(kernelset) //global variable
4:     for i = 1 to size(kernelset) do
5:         validKernel[i] = 1 //global array
6:     end for
7: end function
1: // Create an array to count the number of kernels that include each belief
2: function initializeBeliefOccurrences(beliefs)
3:     for i = 1 to size(beliefs) do
4:         occurs[i] = 0 //global array
5:     end for
6: end function
1: // For each belief, compute the number of kernels that include it
2: function computeOccurrences(kernelset, beliefs)
3:     for i = 1 to size(beliefs) do
4:         for j = 1 to size(kernelset) do
5:             if kernelset[j].contains(beliefs[i]) then
6:                 occurs[i] = occurs[i] + 1
7:             end if
8:         end for
9:     end for
10: end function
```

```
1: // Select the belief that appears in biggest number of kernels, and add to giveUpSet
2: function selectMaxBeliefs(kernelset, beliefs)
3:     giveUpSet = {}
4:     while validKernels > 0 do
5:         max = 0
6:         maxIndex = 0
7:         // search for the belief with max number of occurrences
8:         for i = 1 to size(beliefs) do
9:             if occurs[i] >max then
10:                max = occurs[i]
11:                maxIndex = i
12:            end if
13:        end for
14:        // add the selected belief from the containing kernels to giveUpSet and update
   count
15:        for i = 1 to size(kernelset) do
16:            if kernelset[i].contains(beliefs[maxIndex]) then
17:                validKernel[i] = 0
18:                validKernels = validKernels −1
19:                updateOccurrences(kernelset[i], beliefs)
20:            end if
21:        end for
22:        occurs[maxIndex] = 0
23:        giveUpSet = giveUpSet ∪ {beliefs[maxIndex]}
24:    end while
25:    return giveUpSet
26: end function
```

```
1: // disregard the given kernel from the kernels that include each belief
2: function updateOccurrences(kernel, beliefs)
3:     for i = 1 to size(kernel) do
4:         removeOccurances(beliefs, kernel[i])
5:     end for
6: end function
1: // decrement the number of kernels that include the given belief
2: function removeOccurrences(beliefs, belief)
3:     for i = 1 to size(beliefs) do
4:         if beliefs[i] == belief then
5:             occurs[i] = occurs[i] − 1
6:         end if
7:     end for
8: end function
```

3.4.1 Hitting set approach

Another (more efficient) approach to cutting kernels to perform contraction is the hitting set algorithm[2]. The minimal hitting set problem is defined as follows:

Definition 3.4. [1] Given a set $S = \{s_1, s_2, ..., s_n\}$ of n non-empty sets, a minimal hitting set d is a set such that:

$$\forall_{s_i \in S}[s_i \cap d \neq \emptyset] \wedge \nexists_{d' \subset d}[\forall_{s_i \in S}(s_i \cap d' \neq \emptyset)]$$

Thus, d is a minimal hitting set if and only if it contains at least an element from every set, and no proper subset of it is a hitting set. In the context of kernel contraction in $\mathcal{EL}$, we can define minimal hitting set contraction as:

Definition 3.5. (Minimal hitting set contraction) Given a kernelset $K = \{k_1, k_2, ..., k_n\}$ of n kernels, a minimal hitting set $giveUpSet \subset K$ is a set such that:

$$\forall_{k_i \in K}[k_i \cap giveUpSet \neq \emptyset] \wedge \nexists_{giveUpSet' \subset giveUpSet}[\forall_{k_i \in K}(k_i \cap giveUpSet' \neq \emptyset)]$$

Since kernels are subsets of the *TBox*, and our goal to find a *giveUpSet* that hits all kernels, we can use approaches to the minimal hitting set problem to solve it. Although the minimal hitting set is an NP-hard problem, there are some practically efficient algorithms, such as the one introduced in [1], for the minimal hitting set problem, that are feasible in terms of running time.

Minimal hitting sets are different from *minimum* hitting sets. The word minimal means there are no proper subsets that are hitting sets, while the word minimum means minimal and smallest in size. In other words, we call a hitting set minimum if there is no smaller-sized set that is a hitting set. However, the interpretation we use here is minimality. So there might exist different minimal hitting sets with different cardinalities, but none of them has a proper subset that is a hitting set [1]. But our goal was to satisfy the information economy principal by removing hitting sets with minimum cardinality.

For that reason, after getting all minimum hitting sets, we need to consider only the ones that are smallest in size. For the kernel set:

$$kernelset = \{\{a, b\}, \{a, c\}\},$$

there are two minimal hitting sets:

$$\{a\} \qquad \text{and} \qquad \{b, c\}$$

and they are of different sizes. The following step then is to find the smallest of them, which is the one to be used for contraction, which is $\{a\}$.

[2]We call an algorithm that solves the hitting set problem a hitting set algorithm.

We can now use one of the minimal hitting set algorithms combined with the cardinality selection step to implement contraction for a *TBox*, by implementing the minimal incision function that adopts them, to achieve minimum change (which we can call then, a *minimum incision function*). We are not going to implement such a function in this study, but for now, we will assume that there is a function

```
min-hit-CUT(kernelSet)
```

that takes a set of kernels, and returns a minimum hitting set of sentences to give up. Given that the minimum hitting set problem is NP-hard, the min-hit-cut function's complexity is exponential. We can use this function, as if it is implemented, and may implement it in some future work.

3.4.2 Graph approach

Now, we introduce another technique for kernel contraction in $\mathcal{EL}$ that is based on graphs. The reason following graph approach is useful in solving the contraction problem is that we are performing contraction of *TBoxes*, and they have an implicit graph-like hierarchy defined by the subsumption relationship.

An $\mathcal{EL}$ *TBox* consists of GCIs (we can transform all definition formulas to GCIs), that can be seen as nodes and edges. A GCI can be thought of as a directed edge between two nodes representing concepts on the two sides of the subsumption symbol ($\sqsubseteq$). So reasoning with *TBoxes* is similar to reasoning with directed graphs. That motivated the idea of reducing the problem of kernel contraction of a *TBox* to a graph problem and using a graph algorithm to solve it. The subsumption relationship that forms a *TBox* seem to have an intuitive interpretation as a directed graph; and that's why we use the word "hierarchy" to denote the relationship between concepts in the *TBox*.

The $\mathcal{EL}$ subsumption algorithm described in [5] uses a graph approach, perhaps because it is intuitive to think of *TBoxes* as graphs. A lot of work has been done in the area of Graph Theory, which makes it easy to use the already existing algorithms to solve some graph problems. Also, graphs are easy to imagine and work with.

Our goal is to reduce the kernel contraction problem to a graph problem, and use some efficient graph algorithms to perform kernel contraction. The algorithm we build here starts by transforming the *TBox* into a graph. Then, it generates all kernels by finding all paths that imply the formula we are contracting. After that, it removes one formula from each kernel by removing an edge from each path, since paths here represent kernels (this will all be explained shortly). Finally it transforms the graph back into a *TBox*.

We will start by describing the algorithm in detail, and then explain each of its main steps.

Main algorithm

Given an $\mathcal{EL}$ *TBox* $\mathcal{T}$, and a GCI $\mathbb{A}$ (where $\mathbb{A}$ is in the form of $C \sqsubseteq D$, such that C and D are arbitrary concept expressions), we contract $\mathcal{T}$ by A using Algorithm 5:

Algorithm 5 Contraction using graph approach

```
1: function GRAPHCONTRACT(𝒯, 𝔸)
2:     complete(𝒯)
3:     G = transform(𝒯)
4:     C = arg_left(𝔸) // gets the concept expression on the left side of the GCI
5:     D = arg_right(𝔸) // gets the right-side concept expression
6:     paths = getPaths(G, C, D)
7:     cutPaths(G, paths)
8:     𝒯 = de-transform(G)
9: end function
```

The complete() function in step 2 uses the $\mathcal{EL}$ subsumption algorithm to compute all subsumptions of $\mathcal{T}$. The subsumption algorithm implicitly computes all subsumptions in the process of checking whether a subsumption holds or not. So, we use that algorithm to compute all subsumptions, and then add an extra step of explicitly adding all subsumptions to $\mathcal{T}$. The subsumption algorithm proceeds in four steps:[3]

1. Normalize the *TBox*.
2. Translate the normalized *TBox* into a graph.
3. Complete the graph using completion rules.
4. Read off the subsumption relationship from the normalized graph.

The fourth step is originally meant to examine the graph to determine whether a given subsumption holds or not. We use this step now to add each subsumption on the graph to the *TBox* $\mathcal{T}$. Here is an example of how this step works. Given the *TBox*:

$$\mathcal{T} = \{Human \sqsubseteq Mammal, Mammal \sqsubseteq Animal\},$$

the new $\mathcal{T}$ after adding all subsumptions would be:

$$\{Human \sqsubseteq Human,\ Human \sqsubseteq Mammal,\ Human \sqsubseteq Animal,\ Mammal \sqsubseteq Mammal,\\ Mammal \sqsubseteq Animal,\ Animal \sqsubseteq Animal\}.$$

The new *TBox* contains a lot of unneeded GCIs. This can be avoided by adding a small step to the subsumption algorithm in [5], such that after completing the subsumption graph using the completion rules, we can remove all subsumptions of the form $X \sqsubseteq X$ (e.g. $Human \sqsubseteq Human$).

The subsumption algorithm is explained in full details in [3]. We now need to explain the transformation of the *TBox* $\mathcal{T}$ into a graph.

Transforming the *TBox* into a graph

Assuming the *TBox* $\mathcal{T}$ contains GCIs of the form $C \sqsubseteq D$ (where D is an arbitrary concept, and C is a concept expression of the form $c_1 \sqcap ... \sqcap c_n$ such that $n \geq 1$) , we construct a graph $graph = (V, E)$, where V is a set of nodes and E is a set of directed edges. Starting with an empty V and E, for every GCI $C \sqsubseteq D$, add C and D to V, and add (C, D) to E. This way every $v \in V$ represents a concept expression, and every pair $(C, D) \in E$ represents the subsumption relation $C \sqsubseteq D$.

Algorithm 6 Transforming a *TBox* to a graph

```
1: function TRANSFORM(T)
2:     result = new Graph(V, E)
3:     for every C ⊑ D in T do
4:         V = V ∪ {C, D}
5:         E = E ∪ {(C, D)}
6:     end for
7:     return result
8: end function
```

Now we have a graph *graph* that represents the *TBox* $\mathcal{T}$. The next step is to compute the paths (using getPaths() function) from C to D (where C and D are graph nodes) using depth-first search. Obviously, computing the paths using depth-first search can take polynomial time. For simplicity, we assume the *TBox* is acyclic. This means we don't allow the following situation:

$$\{C \sqsubseteq D, D \sqsubseteq E, E \sqsubseteq C\}.$$

Thus, the graph must be acyclic too. The algorithm can be generalized to account for cycles. Computing all paths can be done as in Algorithm 7.

This can also be done if the graph contains cycles. It would require using a more complicated algorithm that keeps track of the number of edges and the nodes visited during execution. Finding paths in cyclic graphs is explained in detail in [11].

Graph kernel contraction

The function cutPath() is the incision function. Every path from C to D is actually a subsumption path that entails $C \sqsubseteq D$. Removing an edge from such path would break the subsumption between C and D through this path. So, in order to contract $C \sqsubseteq D$ it is enough to remove one edge from each of the paths from C to D; as each path represents a *kernel* of $C \sqsubseteq D$ and breaking them is sufficient to give up the subsumption.

So, a simple and very easy implementation (though inefficient) for the contraction function is to go over the set of paths, and remove a random edge from every path. Suppose that the graph in Figure 3.1 is interpreted as C is the most specific concept that is subsumed

Algorithm 7 Computing all paths between two nodes

```
1: function getPaths(graph, C, D)
2:     result = {}
3:     Stack path = new empty Stack
4:     computeAllPaths(graph, C, D, path, result)
5:     return result
6: end function
1: function computeAllPaths(graph, C, D, path, paths)
2:     graph = (V, E)
3:     for every (C, X) ∈ E do
4:         if X = D then
5:             Stack temp = new empty Stack
6:             temp.pushAll(path) // adds all edges without changing path
7:             temp.push((C, X))
8:             paths = paths ∪ {temp} // adding the path to the list of paths
9:         else
10:            path.push((C, X))
11:            computeAllPaths(graph, C, X, path, paths)
12:            path.pop()
13:        end if
14:    end for
15: end function
```

by A ($C \sqsubseteq A$), and as we climb the graph up, the concepts get more general. Here there are four C-D paths: (C, X, Z, A, E, D), (C, X, Z, A, F, D), (C, Y, Z, A, E, D), and (C, Y, Z, A, F, D). A function that removes random edges from each path might remove (C, X), (Y, Z), (E, D), and (F, D), which will actually remove the subsumption between C and D. However, it would be better (in terms of minimal change) to remove only (Z, A), which will also guarantee that subsumption between C and D is removed.

Sometimes we prefer to remove GCIs that involve most specific concepts during contraction (this will be discussed in Chapter 4). To contract such GCIs, we can just remove, from each $C - D$ path, the edges going into C. In Figure 3.1, it would mean removing two edges: (C, X) and (C, Y). But removing such edges does not guarantee the minimum change. So we might end up having to choose which strategy is more preferred: minimal change or change with most specific concepts. The minimal hitting set approach that was mentioned earlier might not always satisfy specificity. So the user might have to choose which one to apply first, and which one to use as a tie breaker.

Removing the edges that involve nodes representing most specific concepts is straightforward; remove the edges that go into the most specific concept's node (C in our example). But it is not clear if one chooses to remove the least number of edges, instead, how this can be done. For this, we introduce an approach that uses the Minimum Cut algorithm to determine the minimum number of edges that need to be removed and identify them.

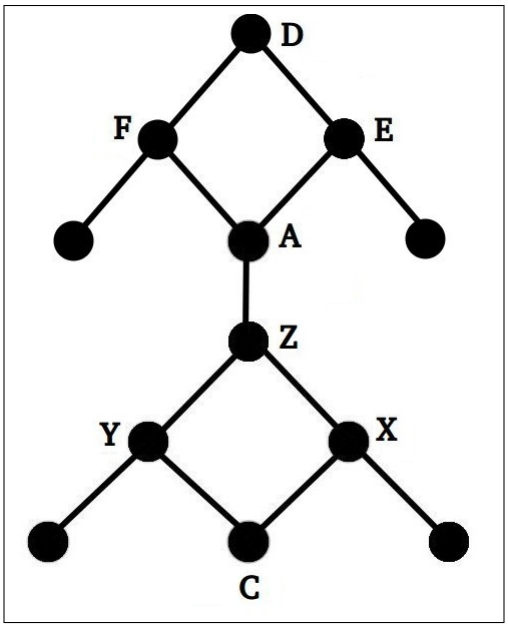

Figure 3.1: Four different paths between C and D, that share one edge.

Reduction to network flow problem

As explained in [11], the network flow problem is the problem of computing the maximum flow possible in a network (represented as a graph) by finding the minimum cut of the network. The input to the Ford-Fulkerson algorithm, that solves this problem, would be:

- A graph $G = (V, E)$.
- A source $s \in V$.
- A sink $t \in V$.
- Capacity function $C : E \to \mathbb{N}$ representing the maximum capacity of each edge.

To contract $C \sqsubseteq D$ given the *TBox* graph G, we choose C to be the source, D to be the sink, and we assume that the capacities of all edges are the same, equal to 1. The algorithm will find the maximum flow from C to D, which is equal to the capacity of the minimum cut (the sum of capacities of the cut edges); we can then extract the edges that form that minimum cut and remove them.

The approach of removing the minimum cut edges of the graph is equivalent to the approach of removing the minimum hitting set formulas of the kernels. The minimum hitting set is the smallest set containing the minimum number of elements that hit all sets, while the minimum cut of the graph is the minimum number of edges (since they all have the same capacity) that cover all paths from C to D (where edges represent GCIs of the *TBox* and paths represent kernels.) So using the minimum cut approach should guarantee the minimum change for kernel contraction, as the minimum hitting set approach does.

Assuming we have a function "Ford-Fulkerson(graph, s, t)" that computes the maximum flow in the network (or graph) from source node "s" to a sink "t" with edge-capacities "1", and returns the set of edges of the minimum cut, we can modify the contraction algorithm to adopt the minimum cut approach as in Algorithm 8.

Algorithm 8 Another version of contraction algorithm

```
1: function GRAPHCONTRACTUSINGMINCUT(𝒯, 𝔸)
2:     complete(𝒯)
3:     graph = transform(𝒯)
4:     C = arg_left(𝔸) //arg_left gets the concept expression on the left side of the GCI
5:     D = arg_right(𝔸) //arg_right gets the right-side concept expression
6:     min-cut = Ford-Fulkerson(graph, C, D)
7:     remove min-cut edges from graph
8:     𝒯 = de-transform(graph)
9: end function
```

For special cases such as contracting $C \sqsubseteq D1 \sqcap D2$, it is sufficient to contract $C \sqsubseteq D1$ first, and then contract $C \sqsubseteq D2$. But since the graph is already normalized (using complete() function), rules of the form $C \sqsubseteq D1 \sqcap D2$ are already broken down into two: $C \sqsubseteq D1$, and $C \sqsubseteq D2$. Therefore, the conjunction $\sqcap$ would only appear on the left hand side of a GCI in a normalized *TBox* (e.g. $A1 \sqcap A2 \sqsubseteq B$). In that case, for contracting $A1 \sqcap A2 \sqsubseteq B$, there will be a node $A1 \sqcap A2$, which we will use as a source node; the sink would be the node representing B.

As in [11], the minimum cut algorithm runs in polynomial time. So using it in the contraction algorithm will not have a significant effect on the complexity of the main algorithm (will not elevate the complexity from being polynomial to being exponential).

In some applications, the decision about which strategy to follow for choosing the edges to remove might vary depending on the situation. So the user can be asked in such case about which strategy to follow – specificity or minimality.

The last step of the algorithm is to transform the graph back to $\mathcal{EL}$. This can be done as follows: starting with an empty *TBox* $\mathcal{T}'$, for every edge (X, Y), add $X \sqsubseteq Y$ to $\mathcal{T}'$. The resulting *TBox* would be the result of contracting $\mathcal{T}$ by $\mathbb{A}$. This is shown in Algorithm 9.

Algorithm 9 Transforming a graph back to a *TBox*

```
1: function DE-TRANSFORM(graph)
2:     result = {}
3:     graph = (V, E)
4:     for every (X, Y) ∈ E do
5:         result = result ∪ {X ⊑ Y}
6:     end for
7:     return result
8: end function
```

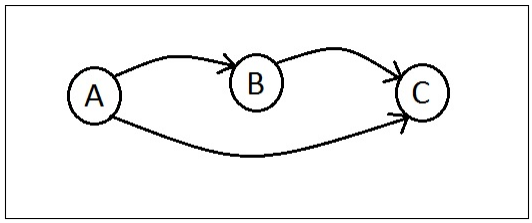

Figure 3.2: Graph representing TBox $\mathcal{T} = \{A \sqsubseteq B, B \sqsubseteq C, A \sqsubseteq C\}$

Since the running time of every step of the contraction algorithm starting from "transform($\mathcal{T}$)" until the last step is polynomial in the size of the *TBox*, the complexity of the algorithm will depend on the complexity of the first step (the complete() function). If building the subsumption hierarchy by generating all subsumptions of an $\mathcal{EL}$ *TBox* can be done in polynomial time, then the contraction algorithm will in turn take polynomial time. But if generating all subsumptions takes exponential time, then the algorithm will take exponential running time as well.

A Sample Run

Suppose we have a TBox $\mathcal{T}$:

$$\mathcal{T} = \{A \sqsubseteq B, B \sqsubseteq C\}$$

which implies $A \sqsubseteq C$. Trying to contract $\mathcal{T}$ by $A \sqsubseteq C$ using the network flow approach would work as follows:

1. complete($\mathcal{T}$). The TBox is already normalized. Translating it into a graph and applying the completion rules will introduce $A \sqsubseteq C$ and will be added explicitly to the TBox.

2. transform($\mathcal{T}$). After transforming the TBox into a graph, it will look like the graph in Figure 3.2 (Assuming we remove the redundant subsumptions, such as $A \sqsubseteq A$).

3. getPaths(graph, A, C) will get all paths between A and C. There are only two paths: A-B-C and A-C.

4. Cutting the two paths A-B-C and A-C will be done by removing the edge between A and C, as well as one of the two edges A-B and B-C. So the resulting graph will only have one edge: A-B or B-C.

5. After transforming the graph again into a TBox, the result will be:

   $$\mathcal{T} = \{A \sqsubseteq B\}$$

   or

   $$\mathcal{T} = \{B \sqsubseteq C\}.$$

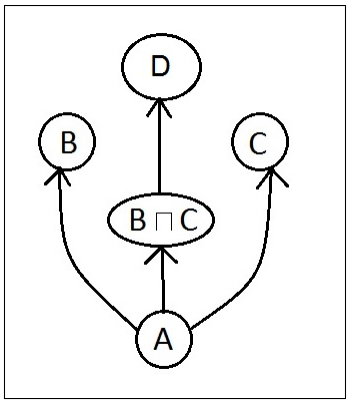

Figure 3.3: Graph representing TBox $\mathcal{T} = \{A \sqsubseteq B, A \sqsubseteq C, A \sqsubseteq B \sqcap C, B \sqcap C \sqsubseteq D\}$

The limitations of the algorithm

The previous example shows how the algorithm succeeds in finding the set of kernels by finding the paths from the source to the sink (where source and sink represent the two sides of the GCI that we need to remove). The example we will discuss now shows how the conjunction symbol ($\sqcap$) might introduce further complexity that the algorithm will not overcome. Given the TBox $\mathcal{T}$:

$$\mathcal{T} = \{A \sqsubseteq B, A \sqsubseteq C, B \sqcap C \sqsubseteq D\}$$

we can see that it entails

$$A \sqsubseteq D.$$

If we want to contract the TBox by $A \sqsubseteq D$, we would look for its kernels and remove a statement from each. In this example we have only one kernel:

$$\{A \sqsubseteq B, A \sqsubseteq C, B \sqcap C \sqsubseteq D\}.$$

So removing one of the three GCIs is enough to give up $A \sqsubseteq D$. Contracting it using our graph approach works as follows:

1. complete($\mathcal{T}$). The TBox is already normalized. Translating it into a graph and applying the completion rules will introduce $A \sqsubseteq B \sqcap C$ and will be added explicitly to the TBox.

2. transform($\mathcal{T}$). After transforming the TBox into a graph, it will look like the graph in Figure 3.3.

3. getPaths(graph, A, D) will get all paths between A and D. There is only one such path, which is $A - B \sqcap C - D$. So removing one of these two edges is the only possible outcome of the algorithm.

This last step is where the algorithm fails. It finds only one of the kernels because it does not recognize that the two parallel edges from A to B and C actually form another kernel. This is because a kernel in our graph approach can only be represented as a path. The conjunction symbol $\sqcap$ has a different meaning, and is not analogous to the path notion. However, the algorithm worked with the previous example because the GCI we tried to remove was the result of applying the transitivity property of the subsumption symbol ($\sqsubseteq$) to two other GCIs and they together were represented by a path in the graph representation.

Similarly, we can argue that the algorithm also fails when the existential quantification symbol ($\exists$) is used. For example, the following TBox:

$$\mathcal{T} = \{A \sqsubseteq \exists r.B, B \sqsubseteq C\}$$

implies the following expression:

$$A \sqsubseteq \exists r.C$$

The graph approach will not get the kernels of that expression using the getPaths() function. So, it will also fail. But this is only because of the subsumption of B by C, where B is included in the role expression $\exists r.B$. If the existential quantification symbol is used without changing the symbols involved in the roles (such as B), the algorithm will work fine, as it will only be depending on the subsumption relations between concept expressions.

So it seems that the limitations of this algorithm are only due to the difference between the inference using paths and the inference using conjunction or existential quantification. However, the algorithm does not always fail when $\sqcap$ or $\exists$ is involved. In Figure 3.2, if we replace B by (X $\sqcap$ Y), or by ($\exists r$.B), the algorithm will succeed. That is because the nodes do not change after the inference; all that is added by the subsumption algorithm are new edges.

Analysis of the graph algorithm

The analysis we refer to here is with respect to rationality. It is important to be able to measure the correctness of the solution we get from running the algorithm. Correctness could be measured by the rationality postulates that are expected to govern kernel contraction. We will examine each of the four postulates that Hansson mentioned in [8] and see if the algorithm actually satisfies all of them.

Success If $\alpha \notin Cn(\emptyset)$, then $\alpha \notin Cn(A \div \alpha)$.

Since according to the definitions of kernels, each kernel is one way to imply α, and since we are incising all kernels (all paths), then the new set cannot imply α (except in the cases that the algorithm cannot solve). So this postulate is satisfied.

Inclusion $A \div \alpha \subseteq A$.

Since we are not adding any extra beliefs, and assuming A is the normalized version of the TBox, then this postulate is also satisfied. It is safe to assume that A is the normalized TBox because normalization does not change the belief set (or the epistemic state) of the agent, but it only changes the belief base by putting the beliefs in a form easier to contract.

Core-retainment If $\beta \in A$ and $\beta \notin A \div \alpha$, then there is a set A' such that $A' \subseteq A$ and that $\alpha \notin Cn(A')$ but $\alpha \in Cn(A' \cup \{\beta\})$.

This postulate is also satisfied because we are only removing beliefs that contribute to the contracted belief, as the minimum cut actually represents the smallest set of sentences that can be removed to contract the knowledge by α (*minimum change*). So, if a belief β doesn't contribute to making A imply α, then β will not be part of any kernel (path), and hence will not be possible to remove.

Uniformity If it holds for all subsets A' of A that $\alpha \in Cn(A')$ if and only if $\beta \in Cn(A')$, then $A \div \alpha = A \div \beta$.

Given the semantics of $\mathcal{EL}$, two GCIs can be equivalent if and only if they are exactly the same. So this requirement is trivially achieved.

3.4.3 How to generate kernels?

So far, we have discussed two ways for generating all kernels:

1. Using the pinpointing algorithm.
2. Using the graph approach.

Computing kernels using the graph approach finds all kernels by performing depth-first search on the graph representation of the TBox. It finds all kernels in $O(E)$, where E is the number of edges in the graph (or the number of GCIs in the TBox). However, it only works with subsumptions that do not include the conjunction symbol ($\sqcap$). So, in cases where the conjunction is not used in the TBox, the graph approach will be at least as efficient as the pinpointing approach, if not better (the axiom pinpointing algorithm introduced in [6] finds one kernel in polynomial time).

3.4.4 What incision function to use?

If we care about selecting the minimum number of beliefs to remove, then there are two incision approaches we discussed so far:

1. The minimum hitting set algorithm.
2. The minimum cut algorithm for the network flow problem.

The minimum hitting set problem is NP-hard. However, the algorithm we discussed in this study is a relatively efficient algorithm. However, in cases where the minimum cut approach can be used, the minimum cut algorithm is probably the best and most efficient. The algorithm works by finding the max flow possible through the network build from the TBox, and then uses it to find the minimum cut.

The Ford-Fulkerson maximum flow algorithm takes $O(E \cdot F)$ steps to find the maximum flow, where E is the number of edges and and F is the value of the maximum flow. The value of F is upper bounded by the the sum of all capacities, i.e. the maximum flow can never be greater than the sum of capacities of all edges. Since all edges in our approach has capacity 1, F cannot exceed E. So the complexity of the algorithm can be reformulated as $O(E^2)$. After finding the maximum flow, we can find the minimum cut in $O(E)$. So the overall complexity of the algorithm is $O(E^2)$. The Ford-Fulkerson algorithm is not polynomial-time, in general, because it depends on the value of the maximum flow, which is dependent on the capacities. There are other algorithms that are proved to be polynomial-time in general, such as the one introduced in [10]. However, the application of Ford-Fulkerson algorithm here is always polynomial-time, because the capacities are all 1 which makes the maximum flow a polynomial function of the number of edges.

In other cases where the graph approach is not applicable, the minimum hitting set algorithm is a good candidate. In the next chapter, we will also give some techniques for implementing the incision function. Some of them might be even more suitable than the minimum hitting set approach, depending on the domain and the preferences of the user.

Chapter 4

Heuristics for contraction

In this study we care about approaching optimal contraction. In the context of contraction, we can define optimality to be removing the smallest number of beliefs, or removing beliefs that make more sense to be removed together. In the last chapter, we discussed contraction algorithms that attempt to achieve optimality by trying to remove the smallest number of beliefs in an attempt to adopt the concept of minimum change. However, we haven't considered the significance of the DL language or the subsumption hierarchy of $\mathcal{EL}$ in choosing the beliefs to be removed from a $TBox$. In this chapter, we look at heuristics that are motivated by the subsumption relationship between formulas. We will discuss ways of determining a preferred set of beliefs to be removed, based on the subsumption relationship of $\mathcal{EL}$ represented by the symbol $\sqsubseteq$.

According to *localization* and *specificity* heuristics[1], the preference of removing a certain set of beliefs is not based on their count. We can say that it is "semantically" better to remove beliefs that are related than to remove unrelated beliefs; by "semantically" we mean according to the meaning of the language's logical connectives. The semantics of DL language's logical connectives imply certain meanings to the knowledge expressed using it. Keeping such meaning in mind while contracting helps make a reasonable decision on what beliefs to be removed. For example, it is arguably better to remove beliefs about specific concepts than general ones. Localization and specificity, can be used as tie-breakers when we have multiple *giveUpSets* of the same size and we need to select only one. They can also be used on their own regardless of the size of the *giveUpSets*.

4.1 Localization

Giving up beliefs about a certain concept means being in doubt of this concept. The more beliefs we remove, the more concepts we become in doubt of. This can lead us to considering removing beliefs about a certain concept rather than beliefs about many different concepts.

[1]These heuristics will be explained shortly.

As discussed in [12], it is more efficient to remove beliefs that share concepts or roles in a TBox as it means being in doubt of less number of concepts.

This seems a reasonable approach as humans tend to deal with related beliefs than unrelated ones. If we try to revise our knowledge, we will probably find ourselves leaning towards focusing on beliefs that share common concepts.

The following $\mathcal{EL}$ example:

$$(1)\ AspergillusFumigatus \sqsubseteq Multicellular$$
$$(2)\ Multicellular \sqsubseteq NotBacteria$$
$$(3)\ AspergillusFumigatus \sqsubseteq Eukaryotes$$
$$(4)\ Eukaryotes \sqsubseteq NotBacteria$$

implies:

$$(5)\ AspergillusFumigatus \sqsubseteq NotBacteria,$$

and we can see that there are two (5)-kernels:

$$\{1,\ 2\} \text{ and } \{3,\ 4\}$$

So, to contract the knowledge base by (5), we need to remove a belief from each kernel. However, removing (1) and (4) seems unreasonable because it means that we are in doubt of our knowledge of four concepts: $AspergillusFumigatus$, $Multicellular$ beings, $Eukaryotes$, and beings that are $NotBacteria$, while removing (1) and (3), on the other hand, makes us in doubt only of our knowledge of only three concepts: $AspergillusFumigatus$, $Multicellular$ beings, and $Eukaryotes$. Not only that, but removing (1) and (3) actually means that our knowledge of $AspergillusFumigatus$ is wrong; Our knowledge of $Multicellular$ beings and $Eukaryotes$ is not necessarily false. Same with removing (2) and (4), we suspect the soundness of our beliefs about beings that are $NotBacteria$.

Localization is implemented in [12] using a graph-like approach. The algorithm operates by defining edge-like relationship between formulas and using graph connectivity and graph clustering rules to determine localized sets.

For every two GCIs $\mathbb{X}$ and $\mathbb{Y}$, we say that there is an edge between them if they share a concept or role. After applying that, we will end up with a graph-like structure, where some GCIs of the knowledge base are connected by edges. The most basic path in a graph is an edge, but a path can be composed of more than one edge. Generally, a path is a sequence of edges.

Definition 4.1. (Conncected GCIs) Two GCIs $\mathbb{A}$ and $\mathbb{B}$ are connected if and only if there is a path from one of them to the other, i.e. if there is a sequence of edges and GCIs $\mathbb{A}e_1G_1e_2...G_{n-1}e_n\mathbb{B}$, where G_i is a GCI, e_i is an edge, every triple $G_ie_{i+1}G_{i+1}$ means G_i and G_{i+1} are connected by an edge, and $n \geq 0$.

In the last $\mathcal{EL}$ example, we can remove one of the following sets to contract (5):

- {1, 3} (a)
- {1, 4}. (b)

There are more possible *giveUpSets*, but (a) and (b) are sufficient to show how the definition of connected GCIs can be applied.

For the set (a), there is an edge between (1) and (3) because they share the same concept *AspergillusFumigatus*. So, according to the definition, (1) and (3) are connected.

For the set (b), there is no edge between (1) and (4) because they don't share any concept or role. So according to the definition, (1) and (4) are not connected.

Definition 4.2. (Clusters) A GCI $\mathbb{X}$ belongs to a cluster $\mathfrak{C}$ if and only if $\mathbb{X}$ is connected to every GCI in $\mathfrak{C}$.

Thus, according to the definitions, the set (a) has only one cluster, as (1) and (3) are connected and belong to the same cluster. However, the set (b) has two clusters, because (1) and (4) are not connected in (b), so they belong to two different clusters.

The algorithm uses the notion of clusters to identify the best *giveUpSet*. A GCI belong to a cluster if it shares concepts or roles with other GCIs in the *giveUpSet*. If a *giveUpSet* contains a lower number of clusters than another set, then it is better to remove it during contraction. This is because the more clusters there is in a set the more sparse the graph is, hence the less localized the contraction is.

The algorithm works on each possible *giveUpSet* separately. It builds the graph-like relation between formulas in the set according to the definitions we gave. It is important that the algorithm computes the connectivity in each set separately, because two formulas might have a path between them in one set but not the other; so they would be considered connected in a set and disconnected in the other. The algorithm then counts the number of clusters. It does that for all the sets, then chooses the set with least number of clusters to be removed.

Algorithm 10 selects the giveUpSet with least number of clusters by first computing the number of clusters in each giveUpSet and then choosing the one with the smallest number. The time complexity of the algorithm is $O(n^3 \cdot m)$, where m is the number of kernels, n is the size of the biggest kernel. The loop in the main function goes over the whole list of kernels, which is of size m, and the second function has a 3-level nested loop that goes over the kernel whose size is at most n in two levels. The third level checks the connectivity of two nodes, which can take at most n if it has to go over all the elements in the kernel.

4.2 Specificity

The subsumption hierarchy enforced by the subsumption relation in $\mathcal{EL}$ categorizes concepts into different levels of generality and specificity. Consider the kernel:

Algorithm 10 Computing localized hit

```
1: // Compute the number of clusters in each set and choose the least
2: function getLocalizedHit(giveUpSets)
3:     setWithLeastClusters = null
4:     smallestNumber = ∞
5:     for giveUpSet ∈ giveUpSets do
6:         numOfClusters = getNumberOfClusters(giveUpSet)
7:         if numOfClusters < smallestNumber then
8:             setWithLeastClusters = giveUpSet
9:             smallestNumber = numOfClusters
10:        end if
11:    end for
12:    return setWithLeastClusters
13: end function
1: // Compute the number of clusters
2: function getNumberOfClusters(giveUpSet)
3:     for i = 1 to size(giveUpSet) do
4:         label[i] = i
5:     end for
6:     for i = 1 to size(giveUpSet) do
7:         cluster[i] = 0
8:     end for
9:     // Label connected nodes (beliefs) similarly
10:    for i = 1 to size(giveUpSet) do
11:        for j = i + 1 to size(giveUpSet) do
12:            if giveUpSet[i] and giveUpSet[j] are connected then
13:                label[j] = label[i]
14:            end if
15:        end for
16:    end for
17:    // Mark clusters based on used labels. Each label now represents a cluster
18:    for i = 1 to size(giveUpSet) do
19:        cluster[label[i]] = 1
20:    end for
21:    // Calculate the number of clusters by counting the used labels
22:    for i = 1 to size(giveUpSet) do
23:        numberOfClusters = numberOfClusters + cluster[i]
24:    end for
25:    return numberOfClusters
26: end function
```

$$Lion \sqsubseteq Mammal$$
$$Mammal \sqsubseteq Vertebrate$$
$$Vertebrate \sqsubseteq Animal$$

Figure 4.1: Simple Animal kernel

which implicitly entails

$$Lion \sqsubseteq Animal$$

And suppose we would like to contract the $TBox$ by $Lion \sqsubseteq Animal$. We have three options: we can remove $Lion \sqsubseteq Mammal$, $Mammal \sqsubseteq Vertebrate$, or $Vertebrate \sqsubseteq Animal$. Removing any of the three GCIs would guarantee minimum change. We can say that $Lion$ is more specific than $Mammal$ and $Mammal$ is more specific than $Vertebrate$. Removing $Lion \sqsubseteq Mammal$, in this case, is more reasonable than removing $Vertebrate \sqsubseteq Animal$, because $Lion \sqsubseteq Mammal$ involves concepts that are more specific than the ones involved in $Vertebrate \sqsubseteq Animal$. Removing $Vertebrate \sqsubseteq Animal$ may affect more concepts in the subsumption hierarchy (in worst case) than those affected by removing $Lion \sqsubseteq Mammal$. This means that removing $Vertebrate \sqsubseteq Animal$ means that all individuals that are $Vertebrates$ are no longer believed to be $Animals$, while removing $Lion \sqsubseteq Mammal$ only affects a subset of $Vertebrates$ which is $Lions$.

So, it is preferable to remove GCIs that involve specific concepts rather than removing GCIs that involve more general ones. To account for specificity, we assign a label to each of the concepts representing its level in the subsumption hierarchy, where levels increase with generality. And during contraction, we consider contracting GCIs that involve concepts at lower level (more specific) before considering GCIs that involve concepts at higher level.

Before explaining how the algorithm works, we need to explain the notion of *children* that we use in the context of subsumption hierarchy. We use this notion to define the relationship between concepts that reflects a tree-like relationship between concepts. This will help us determine and compute specificity for each GCI.

Similar to the hierarchy of the subsumption graph of an $\mathcal{EL}$ TBox, we give a definition to *parent* and *child* concepts as follows:

Definition 4.3. For the GCI $A \sqsubseteq B$:

- A is a *child* concept relative to B, and
- B is a *parent* concept relative to A.

So by looking again at Figure 4.1, we can now say that:

- Mammal is a *parent* of Lion, and Lion is a *child* of Mammal

- Vertebrate is a *parent* of Mammal, and Mammal is a *child* of Vertebrate
- Animal is a *parent* of Vertebrate, and Vertebrate is a *child* of Animal
- Animal is the most *general* concept, i.e. Animal subsumes all other concepts
- Lion is the most *specific* concept (least *general*), i.e. it is subsumed by all other concepts

The approach we follow in this study for adopting the preference of removing sentences that contain more specific concepts is achieved by assigning weights to every GCI that reflects its generality. Then we select the kernels with minimal overall weight. Every GCI gets a numeric weight depending on the level of generality of the concepts involved in it (depending on their position in the subsumption graph). A GCI $A \sqsubseteq B$ gets weight '0' if A has no children, i.e. if there is no rule of the form $X \sqsubseteq A$, where X is an arbitrary concept; and the GCI involving its parent (A GCI of the form $B \sqsubseteq C$) will have weight '1' in that case. So the weight somehow represents the level in the subsumption graph, starting from level '0' at the trees' leaves (which is A in this example.) For simplicity, we assume that the $TBox$ is acyclic – so we avoid the problems that will be caused by loops.

Now we can compute the weights for the last TBox:

- $Lion \sqsubseteq Mammal$ has weight = 0
- $Mammal \sqsubseteq Vertebrate$ has weight = 1
- $Vertebrate \sqsubseteq Animal$ has weight = 2

The algorithm we introduce now removes kernels that contain GCIs involving most specific concepts by first computing the weights of every kernel based on the weights of the GCIs inside it. Following Definition 4.3, we start the algorithm by building the *children* labels for all the concepts in the $TBox$ that we will use to assign weights to the GCIs. Given a $TBox$ $\mathcal{T}$, the algorithm proceeds as in Algorithm 11.

Algorithm 11 Building the children graph

```
1: function GRAPHBUILD(T)
2:     children(X) = ∅, for every concept X in T
3:     for every A ⊑ B ∈ T do
4:         children(B) = children(B) ∪ {A}
5:     end for
6: end function
```

Now we have a graph composed of nodes and their *child* relationship; the nodes represent concepts and the edges (or the child relationship) are the subsumption relationship between them. So GCIs are represented by edges in the graph. The roots are now nodes that are not

children of any node, and the leaves are nodes with no children. The edges involving leaves are assigned weight '0' and it increases by '1' every step towards the roots. So, the roots are most general concepts, and the leaves are most specific ones. If a GCI contains concepts $\alpha \sqsubseteq \beta$, where α has no children and β has no parents, then it gets weight '0' according to Algorithm 12. The weights are assigned to edges, not nodes. Given an edge e that connects node a to its child b, if b has more than one child, the weight of e is determined by the longest sequence of edges to a leaf starting at any of the children of b. So the GetMaxWeight function traverses a tree to all of its leaves computing the weight of each edge and choosing the weight according to the longest path to a leaf.

Algorithm 12 Assigning Weights

```
1: function assignWeights(T)
2:     for every A ⊑ B ∈ T do
3:         weight(A ⊑ B) = GetWeight(A ⊑ B)
4:     end for
5: end function
1: function getWeight(X ⊑ Y)
2:     if weight(X ⊑ Y) ≠ NULL then
3:         return weight(X ⊑ Y)
4:     end if
5:     if children(X) = ∅ then
6:         return 0
7:     else
8:         return 1 + GetMaxWeight(X)
9:     end if
10: end function
1: function getMaxWeight(X)
2:     max = – 1
3:     for every Z in children(X) do
4:         w = GetWeight(Z ⊑ X)
5:         if  w > max then
6:             max = w
7:         end if
8:     end for
9:     return max
10: end function
```

Now we have every GCI in $\mathcal{T}$ assigned a weight relative to its level of generality. So, in every kernel we have a preference level of what to give up. Based on the weights, we will choose the sentences with less weight over the ones with more weight to remove. This preference can be used as a tie-breaker after we apply the minimal hitting set algorithm and end up with more than one minimal set. If we arrive at two minimal hitting sets for the kernels, we can compute the overall weight of each of the hitting sets and remove the GCIs in the set with less overall weight. This is shown in Algorithm 13. It uses the min-hit-CUT

algorithm that gets the minimum hitting sets given a kernel set. There could be more than one minimum hitting set.

Algorithm 13 Removing specific hitting set

```
1: function getMostSpecificHit(kernelset)
2:     min-hit-sets = min-hit-CUT(kernelSet)
3:     min = ∞
4:     hit = NULL
5:     w(s) = 0, for every s ∈ min-hit-sets
6:     for every s in min-hit-sets do
7:         for every A ⊑ B in s do
8:             w(s) = w(s) + weight(A ⊑ B)
9:         end for
10:        if w(s) < min then
11:            min = w(s)
12:            hit = s
13:        end if
14:    end for
15:    return hit
16: end function
```

Now the CUT method in Algorithm 1 can be re-implemented using the algorithms introduced here to account for specificity heuristics. Given a TBox $\mathcal{T}$ and a GCI $\mathbb{A}$, the contraction algorithm is given in Algorithm 14.

Algorithm 14 Contraction algorithm – modified

```
1: procedure contract(T, A)
2:     kernelset = pinpoint(T, A)
3:     graphBuild(T)
4:     assignWeights(T)
5:     giveUpSet = getMostSpecificHit(kernelset)
6:     T=T/ giveUpSet
7: end procedure
```

The contraction algorithms in this version guarantees the minimality, then specificity, of the removed beliefs. It guarantees the minimality by reducing the selection of beliefs from the kernelset to the minimal hitting set problem. The better the performance and the optimality of the hitting set algorithms, the efficient and minimal the contraction algorithm is. If the minimal set algorithm produces more than one solution of the same size, the specificity heuristic is used as a tie-breaker. It selects the solution with more specific beliefs to be removed.

The time complexity of Algorithm 12 is polynomial. The worst-case time complexity is $O(n \cdot l)$, where n is the size of the TBox $\mathcal{T}$ and l is the depth of the subsumption graph. Since the largest value for l is n, we can consider the worst-case time complexity

to be $O(n^2)$. The algorithm works as depth-first search because of the recursive call in $getWeight$ function. The $graphBuild$ function takes $O(n)$ steps to build the parent-child relationship graph, where n is the size of the TBox $\mathcal{T}$. Finally, the $getMostSpecificHit$ function takes $O(n \cdot m)$ steps in the worst case, where n is the size of the TBox $\mathcal{T}$, and m is the number of kernel sets. So the contraction algorithm takes polynomial time if the minimum hitting set algorithm runs in polynomial time. But, if the minimum hitting set algorithm is not polynomial-time, then the contraction algorithm will not be. Thus, it is safe to say that the time complexity of the contraction algorithm is lower-bounded by the time complexity of the minimum hitting set algorithm's time complexity.

Chapter 5

Conclusion

We have discussed that the AGM framework introduces a reasonable set of rules that can be followed to reach rationality in building belief systems. However, we used a different formalism to express knowledge and perform contraction throughout this study than the one the AGM framework uses. This formalism, which is $\mathcal{EL}$, belongs to a family called Description Logic (DL). $\mathcal{EL}$ is one of the simple members of the DL family, and, as the rest of the DLs, uses concepts and roles as the building blocks of the knowledge base. We only showed how to perform contraction on TBoxes, and not ABoxes. We then discussed some basic approaches to contraction and introduced a general algorithm that can accommodate the use of different heuristics to seek optimality.

We introduced a restricted contraction algorithm that uses graphs and solves contraction as a network flow problem. The graph approach is restricted in the sense that it can be used only in certain cases where inference in the TBox involves only the transitive property of the subsumption relation. We showed how the approach is sound and complete (only in the restricted cases) by showing that it follows most of the AGM postulates for contraction. We also discussed three heuristics, two of which are based on the semantics of the subsumption hierarchy of $\mathcal{EL}$, which are Localization and Specificity. We showed also how the greedy approach for contraction can be used.

The bottleneck for the contraction algorithms we discussed is actually generating all the kernels of a specific belief. The pinpointing algorithm we use for this purpose can sometimes take exponential number of steps with respect to the size of the TBox. Other than that, all the algorithms we discussed take polynomial time to run.

TBox kernel contraction can be implemented to find the smallest set of beliefs to be removed in polynomial time using the graph approach. The graph approach works only when the inference involves only the subsumption relationship, not the existential quantification. The complexity of the algorithm is the same as the network flow algorithm complexity as it uses the concept of minimum cut to compute the kernels.

Specificity is also one of the main contributions of this study. The $\mathcal{EL}$ description logic sets up a hierarchical relationship between concepts that reflects a specificity relationship.

A concept can be said to be more (or less) specific compared to another concept based on the subsumption relationship in the TBox. This inspires the solution we adopted in removing knowledge about more specific concepts before considering more general ones.

Two things are worth mentioning and could be addressed in some future work. First, contracting a TBox by $A \sqsubseteq B \sqcap A \sqsubseteq C$, which is equivalent to contracting by $A \sqsubseteq B \sqcap C$, can be done by contracting the TBox by only one of the two GCIs. However, the result of contracting by $A \sqsubseteq B$ will probably be different from the result of contracting by $A \sqsubseteq C$. So how can we choose which GCI of the two to contract by? It could be based on the side effects that each of them will cause to the TBox.

Second, our graph approach works by finding the minimum cut after computing the maximum flow. What if there are more than one cut? It would be useful if we use some algorithms that find all minimum cuts and apply heuristics to select the most appropriate cut to be removed. The current version of the algorithm we introduced is agnostic about the semantics of the minimum cut produced. But if we can combine it with some heuristics (e.g. specificity), we might get a more reasonable and more rational solution.

One addition that could be done in the future is to give an algorithm that contracts explicit knowledge in the ABox, not just the TBox. Also, the use of $\mathcal{EL}$ was very important to finding polynomial time contraction algorithms. One other suggestion is to attempt to use other, more expressive, versions of DL and investigate if this results in higher order contraction than polynomial. Another intriguing question is: what advantage would using a more expressive DL give, and will it be worth it if contraction algorithms get harder and more complicated?

Bibliography

[1] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In Vadim Bulitko and J. Christopher Beck, editors, *SARA*. AAAI, 2009.

[2] Carlos E. Alchourrón and David Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, 44(4):405–422, 1985.

[3] Franz Baader. What's new in description logics. *Informatik-Spektrum*, 34(5):434–442, Oct 2011.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[5] Franz Baader, Carsten Lutz, and Anni-Yasmin Turhan. Small is again beautiful in description logics. *KI - Künstliche Intelligenz*, 24(1):25–33, Apr 2010.

[6] Franz Baader, Rafael Peñaloza, and Boontawee Suntisrivaraporn. Pinpointing in the description logic $\mathcal{EL}$. In *Proceedings of the 2007 International Workshop on Description Logics (DL2007)*, volume 250 of *CEUR-WS*, pages 171–178, Brixen/Bressanone, Italy, June 2007. Bozen-Bolzano University Press.

[7] P. Gärdenfors. *Knowledge in Flux. Modelling the Dymanics of Epistemic States*. Mit Press, 1988.

[8] S.O. Hansson. *A Textbook of Belief Dynamics: Theory Change and Database Updating*. Number v. 2 in Applied logic series. Kluwer Academic Publishers, 1999.

[9] Sven Ove Hansson. Kernel contraction. *J. Symb. Log.*, 59(3):845–859, 1994.

[10] A.V. KARZANOV. Determining the maximal flow in a network by the method of preflows. volume 15, pages 434–437, 1974.

[11] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[12] Zhiwei Liao. Kernel contraction in $\mathcal{EL}$. Master's thesis, Simon Fraser University, 2014.

[13] Pavlos Peppas. *Handbook of Knowledge Representation*, chapter 8, pages 317–359. Foundations of ARTIFICIAL INTELLIGENCE. Elsevier B.V., first edition, 2008.

[14] Rafael R. Testa. The cost of consistency: information economy in paraconsistent belief revision. *South American Journal of Logic*, 1(2):461–480, December 2015.