More on kernel contraction in $\mathcal{EL}$

by

Amr Dawood

B.Sc., German University in Cairo, 2011

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

in the
Department of Computing Science
Faculty of Applied Sciences

Approval

Name:	**Amr Dawood**
Degree:	**Master of Science (Computing Science)**
Title:	***More on kernel contraction in $\mathcal{EL}$***
Examining Committee:	**Chair:** name Professor

James P. Delgrande
Senior Supervisor
Professor

David G. Mitchell
Supervisor
Associate Professor

Date Defended: 1 September 2015

Abstract

This is a blank document from which you can start writing your thesis.

Keywords: Masters; Thesis; Simon Fraser University; Kernel Contraction; EL; Description Logic; Specificity; Localization

Dedication

Acknowledgements

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Learning is one of the most intuitive functions of the human mind. It involves translating human's perception into knowledge, and adapting the state of mind according to that knowledge. Seeing the sunrise in the morning, one would intuitively change his mind and believe that it is not dark anymore; that would logically imply that one gives up the current belief that it is dawn time and replace it with a new belief that it is sunrise time. This is an example of belief revision. Revision is a kind of change in which the new belief ("it is sunrise") is conflicting with the current state of mind (believing that "it is dawn").

Learning 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 the reality of learning. 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 caused 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. In the following chapter, we will discuss in some level of detail these types of change with more focus on contraction.

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*. 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

[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 base and perception.

reasoning mechanism 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.

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 discuss what description logic is, what logical operators are used and what it is composed of. In that discussion, we focus on the most relevant variation to this study, which is $\mathcal{EL}$. And we explain the most important reasoning algorithm for $\mathcal{EL}$.

Chapter 2

Background

2.1 Belief change

2.1.1 Epistemic states

2.1.2 Epistemic attitudes

2.1.3 Basic types of change

2.1.4 AGM framework

2.2 Description logic

2.2.1 ABox

2.2.2 TBox

2.2.3 $\mathcal{EL}$ language

Chapter 3

Kernel contraction

Belief contraction is the process of removing beliefs from belief bases. It can be done either by selecting one of the largest subsets of the belief base that do not logically imply the abandoned belief, or by incising all the minimal sets that logically imply that belief; by "incising" we mean removing a member from the set. The latter approach is the one we are going to focus on in this study. If a set K of beliefs imply a belief α, and K is the minimal such set (no proper subset of K implies α), removing one element of K will guarantee that it does not imply α. Contraction is done here by finding all such minimum sets and removing an element from each of them. We call such minimal sets *kernels*. Kernels will be discussed in more details in the coming section. But we need to go over some definitions first.

Consider the following 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:

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

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

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

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 $a \to b$. So the two possible resulting belief sets after contraction are:

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

This is how kernel contraction works. Every b-kernel is one way to infer b. To contract b every kernel should be broken by removing at least one belief.

3.1 What are kernels?

Kernel contraction was introduced by Hansson[7] 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 make $\mathbf{K}$ imply α. Beliefs that contribute to make $\mathbf{K}$ imply α are members of some α-kernels of $\mathbf{K}$, and all members of every α-kernel contribute to make $\mathbf{K}$ 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 refer to the set of α-kernels of $\mathbf{K}$ by $\mathbf{K} \perp \alpha$.

Definition 3.1. (Kernel Set) According to [6], for a belief set $\mathbf{K}$ and a belief α, $\mathbf{K} \perp \alpha$ is a 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 $Y \nvdash \alpha$.

$\mathbf{K} \perp \alpha$ is a kernel set that contains all α-kernels of $\mathbf{K}$.

Kernels are the smallest subsets of the knowledge base that imply a specific belief. Therefore, in order to give up that belief, every one of those kernels needs to be incised. If those kernels are not "minimal", they might contain some beliefs that do not contribute to make the kernel imply that specific belief. For example, in $\{\alpha, \beta\}$, β is a belief that does not contribute to make the set imply α. In that case, removing β only would not help in giving up α.

So, because of the minimality of the kernels, every belief in the 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 removes an element from each kernel.

The incision function is defined in [6] as follows:

Definition 3.2. (Incision function) An incision function σ 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. [6] (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. The incision function can perform the contraction by cutting through all kernels of a specific belief. So, the first step in contraction is computing all kernels that imply the belief that needs to be removed. For that purpose, we use the pinpointing algorithm introduced in [5].

To show how the algorithm works, we use the example introduced in [5]. Given an $\mathcal{EL}$ *TBox* $\mathcal{T}$ consisting of the following four GCI axioms:

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

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 [5] computes all kernels using a modified version of the $\mathcal{EL}$ subsumption algorithm. It works by finding a monotone boolean formula called *"pinpointing formula"*. The propositions of the pinpointing formula are GCIs of the *TBox*, and a propositional *valuation* represents the kernels. 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, [5] 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 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 [9]. The goal of this study is to continue the work done in [9], and to discuss 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 look like the following:

Algorithm 1 Contraction algorithm

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

"kernelset" refers to ($\mathcal{T} \perp \alpha$), and *giveUpSet* is the set of beliefs selected by the incision function (CUT) to be removed. 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 that picks up beliefs from every kernel, ensuring that it cuts 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 every kernel. This is given in Algorithm 2.

Algorithm 2 Random removal

```
1: function RANDOMREMOVE(kernelset)
2:     giveUpSet = {}
3:     for $kernel \in kernelset$ do
4:         choose a random belief $\alpha$ from $kernel$
5:         giveUpSet = giveUpSet $\cup$ {$\alpha$}
6:     end for
7:     return giveUpSet
8: end function
```

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

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

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 less beliefs. And it could be done easily if the algorithm is smart enough to check if the second kernel has already been incised or not.

The next algorithm (Algorithm 3) does this. It goes over all kernels, and randomly selects a formula from every kernel to be removed. Every time it selects a belief, it marks all kernels that contain that belief so that they are not considered for belief removal in the following iterations.

Algorithm 3 Random removal with exclusion

```
function RandomRemoveAndExclude(kernelset)
    giveUpSet = {}
    for i=0 to size(kernelset)-1 do
        kernelset[i].valid = true
    end for

    for i=0 to size(kernelset)-1 do
        if kernelset[i].valid == true then
            choose a random belief $\alpha$ from kernelset[i]
            giveUpSet = giveUpSet $\cup$ {$\alpha$}
            for j=i to size(kernelset)-1 do
                if kernelset[j].contains($\alpha$) then
                    kernelset[j].valid = false
                end if
            end for
        end if
    end for
    return giveUpSet
end function
```

Algorithm 3 has a worst-case time complexity of $O(n^2)$, where n is the size of the kernel set. This is because the second main loop has another nested loop inside it, and neither of them takes more than n steps to finish.

3.4 Minimal change

According to the information economy principal (minimality), in every change of the epistemic state, loss of information should be minimum[10]. 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*.

3.4.1 Hitting set approach

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 the 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* 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. There are good approximation algorithms, such as the one introduced in [1], for the minimal hitting set problem, that are feasible in terms of running time.

The notion of *minimality* can be interpreted in terms of set-containment or cardinality. We call a hitting set minimal in terms of caridnality if there is no smaller-sized set that is a hitting set. However, the interpretation we use here is minimality accoring to set-containment; this means a minimal hitting set is a hitting set that has no proper subset that is a hitting set. So there might exist different minimal hitting sets with different cardinalities, but none of them is subsumed by 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 determine that the smallest of them should be selected for contraction, which is $\{a\}$.

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 *minimal 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 minimal hitting set (minimal in both set-containment and cardinality) of sentences to give up. 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 why following graph approaches is useful in solving contraction problem is that we are performing contraction of *TBoxes*, and they have an implicit hierarchy of a graph.

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. We can reduce *TBox* reasoning problems to graph problems and use graph algorithms to solve them. The subsumption relationship that forms a *TBox* seem to have an intuitive interpretation as a directed graph; and that's why we might use the word "subsumption hierarchy" later in this study to denote the relationship between concepts in the *TBox*.

The $\mathcal{EL}$ subsumption algorithm described in [4] 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, using some efficient graph algorithms, perform kernel contraction. The algorithm starts by transforming the *TBox* into a graph. Then, it generates all kernels by finding all paths of the graph 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 details, and then explain each of its main steps.

Main algorithm

Given an $\mathcal{EL}$ *TBox* T, and a GCI $\mathbb{A}$ (where $\mathbb{A}$ is of the form $C \sqsubseteq D$, such that C and D are arbitrary concept expressions), we contract $\mathcal{T}$ by A using Algorithm 4:

The complete function in step 2 uses the $\mathcal{EL}$ subsumption algorithm to compute all subsumptions of $\mathcal{T}$. All the subsumptions computed are added to $\mathcal{T}$. The subsumption algorithm proceeds in four steps:[3]

1. Normalize the *TBox*.

2. Translate the normalized *TBox* into a graph.

Algorithm 4 Contraction using graph approach

```
1: function graphContract($\mathcal{T}$, $\mathbb{A}$)
2:     complete($\mathcal{T}$)
3:     graph = transform($\mathcal{T}$)
4:     C = $arg_{left}(\mathbb{A})$
5:     D = $arg_{right}(\mathbb{A})$
6:     paths = getPaths(graph, C, D)
7:     cutPaths(graph, paths)
8:     $\mathcal{T}$ = de-transform(graph)
9: end function
```

3. Complete the graph using completion rules.

4. Read off the subsumption relationship from the normalized graph.

The 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 (or a conjunction of concepts), and every pair $(C, D) \in E$ represents the subsumption relation $C \sqsubseteq D$.

Algorithm 5 Transforming a *TBox* to a graph

```
1: function transform($\mathcal{T}$)
2:     result = new Graph(V, E)
3:     for every $C \sqsubseteq D$ in $\mathcal{T}$ do
4:         $V = V \cup \{C, D\}$
5:         $E = E \cup \{(C, D)\}$
6:     end for
7:     return result
8: end function
```

Transformation analysis

It is worth mentioning that even though subsumption can be decided in polynomial time with respect to the size of the *TBox* (using the algorithm in [4]), computing all subsumptions can be exponential in the number of concepts. For example, with $\mathcal{T} = \{Human \sqsubseteq Mammal, Mammal \sqsubseteq Animal\}$, 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 [4], such that after completing the subsumption graph using the completion rules, we can remove all subsumptions of the form $X \sqsubseteq X$ (e.g. $Human \sqsubseteq Human$); also we need to remove all subsumptions $C \sqsubseteq D$ if $C \sqsubseteq X$ and $X \sqsubseteq D$ (e.g. exclude $Human \sqsubseteq Animal$ if $Human \sqsubseteq Mammal$, and $Mammal \sqsubseteq Animal$ are in the $TBox$) are already there.

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 still be generalized to account for cycles. Computing all paths can be done as in Algorithm 6.

Algorithm 6 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 $(C, X) \in E$ 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:             $paths = paths \cup \{temp\}$ // 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
```

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 in explained in details in [8].

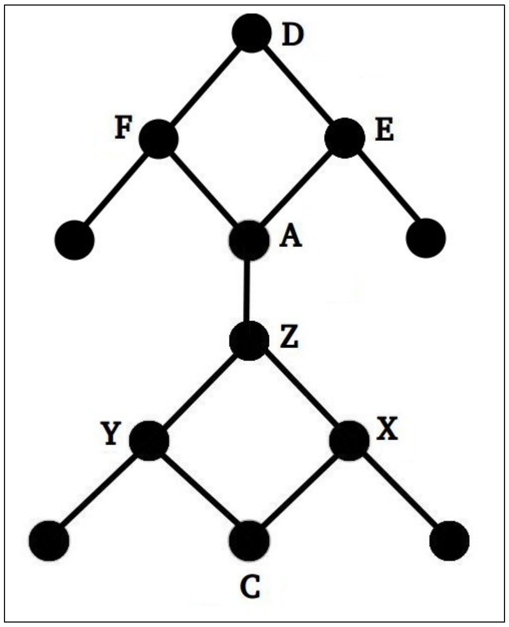

Figure 3.1: Four different paths between C and D, that share one edge.

Graph kernel contraction

The function cutPath is the contraction 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, and C would no longer be subsumed by D – unless there is another subsumption line. 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 remove it.

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 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 the coming chapter). 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 edges that need to be removed and identify them.

Reduction to network flow problem

As explained in [8], 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 capacity is not important here, as it is normally used to determine the bottleneck edges of the network to optimize the flow, while in our case, they are only needed to determine which edges are being used; so, an edge is used in the flow if the flow passing through it is 1, and it can never be greater than 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 minimal hitting set formulas of the kernels. The minimal hitting set is the 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 minimal hitting set approach does.

Assuming we have a function "Ford-Fulkerson(graph, s, t)" that computes the maximum flow in the network "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 7.

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

Algorithm 7 Another version of contraction algorithm

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

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 [8], 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 discussed in Algorithm 8.

Algorithm 8 Transforming a graph back to a *TBox*

```
1: function de-transform(graph)
2:     result = {}
3:     graph = (V, E)
4:     for every $(X, Y) \in E$ do
5:         $result = result \cup \{X \sqsubseteq Y\}$
6:     end for
7:     return result
8: end function
```

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

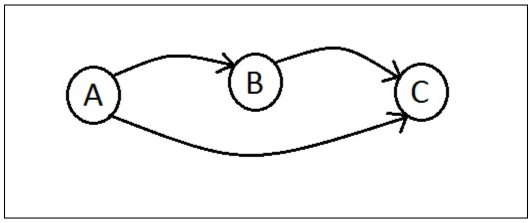

Figure 3.2: Graph representing TBox $\mathcal{T} = \{A \sqsubseteq B, B \sqsubseteq C, A \sqsubseteq C\}$

generating all subsumptions takes exponential time, then the algorithm will take exponential running time as well.

A working example

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.

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

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

This example shows how the graph approach works as expected and produces similar results to the approach of generating kernels using pinpointing algorithm then incising them using the minimal hitting set algorithms. However, this approach does not always work. The next example will show how the conjunction ($\sqcap$) introduces more complexity that the algorithm cannot solve.

Chapter 4

Heuristics for contraction

Considering algorithms that guarantee contraction with minimum change is not always enough for optimal contraction. So far, we have considered minimal contraction by trying to find the smallest *giveUpSet*. However, we haven't considered the significance of the subsumption hierarchy of $\mathcal{EL}$ in contracting 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, we can prefer to remove a certain set of beliefs to another, even if it is bigger or equal in size. 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. Also, it is better to remove beliefs about specific concepts than general ones. These two selection heuristics, 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*.

Before discussing localization and specificity, there is another approach that is worth mentioning. That approach is the *Greedy* approach that was introduced in [9]. It does not account for the subsumption relationship between $\mathcal{EL}$ concepts (it is language independent). It only cares about the intersection between kernels. After explaining how it works, we will turn to the two language-dependent heuristics and see how they work.

4.1 Greedy Contraction

The greedy approach works by selecting the beliefs that appear in more kernels to be removed before those that appear in less kernels. At each step, the algorithm finds which belief appears 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 step). 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 9 explains how the greedy approach for contraction works. Given a set of kernels and the set of all beliefs it proceeds as follows: it computes the number of occurrences of every 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 to the main contraction algorithm to remove them from the knowledge base. So the greedy approach can be embedded in the main algorithm as a way to choosing the beliefs 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 and the size of $kernelset$ is n, then the worst-case time complexity is $O(n^2 + nm)$ (or $O(n(n + m))$).

4.2 Localization

When giving up beliefs, we are not confident about our knowledge of what the beliefs are related to. Removing related beliefs, or beliefs about one common concept, seems better as it only questions the knowledge about that concept. If we remove beliefs about more concepts, then we would be questioning our knowledge about those concepts. According to [9], it is more efficient to remove knowledge that shares concepts or roles because it means there are less areas of knowledge we are questioning.

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:

$$\begin{gathered}
(1)\ AspergillusFumigatus \sqsubseteq Multicellular \\
(2)\ Multicellular \sqsubseteq NotBacteria \\
(3)\ AspergillusFumigatus \sqsubseteq Eukaryotes \\
(4)\ Eukaryotes \sqsubseteq NotBacteria
\end{gathered}$$

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 question

Algorithm 9 Greedy Selection

```
function GreedyContract($kernelset, beliefs$)
    $initializeValidKernels(kernelset)$
    $initializeBeliefOccurances(beliefs)$
    $computeOccurances(kernelset, beliefs)$
    return $selectMaxBeliefs$(kernelset, beliefs)
end function
function initializeValidKernels($kernelset$)
    $validKernels = 0$ //global variable
    for $i = 1$ to $size(kernelset)$ do
        $validKernel[i] = 1$ //global array
        $validKernels = validKernels + 1$
    end for
end function
function initializeBeliefOccurances($beliefs$)
    for $i = 1$ to $size(beliefs)$ do
        $occurs[i] = 0$ //global array
    end for
end function
function computeOccurances($kernelset, beliefs$)
    for $i = 1$ to $size(beliefs)$ do
        for $j = 1$ to $size(kernelset)$ do
            if $kernelset[j].contains(beliefs[i])$ then
                $occurs[i] = occurs[i] + 1$
            end if
        end for
    end for
end function
function selectMaxBeliefs($kernelset, beliefs$)
    $giveUpSet = \{\}$
    while $validKernels > 0$ do
        $max = 0$
        $maxIndex = 0$
        for $i = 1$ to $size(beliefs)$ do
            if $occurs[i] > max$ then
                $max = occurs[i]$
                $maxIndex = i$
            end if
        end for
        for $i = 1$ to $size(kernelset)$ do
            if $kernelset[i].contains(beliefs[maxIndex])$ then
                $validKernel[i] = 0$
                $validKernels = validKernels - 1$
            end if
        end for
        $occurs[beliefIndex] = 0$
        $giveUpSet = giveUpSet \cup \{beliefs[maxIndex]\}$
    end while
    return $giveUpSet$
end function
```

our knowledge of $AspergillusFumigatus$, $Multicellular$ beings, $Eukaryotes$, and beings that are $NotBacteria$, while removing (1) and (3), on the other hand, only questions out knowledge of $AspergillusFumigatus$, $Multicellular$ 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 [9] 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.

Definition 4.1. For every two formulas $\mathbb{X}$ 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 4.2. (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 formulas $Ae_1F_1e_2...F_{n-1}e_nB$, where F_i is a formula, e_i is an edge, every triple $F_ie_{i+1}F_{i+1}$ means F_i and F_{i+1} are connected by an edge, and $n \geq 0$.

So, in the previous $\mathcal{EL}$ 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 4.3. (Clusters) A node X belongs to a cluster $\mathfrak{C}$ if and only if X is connected to every node 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 higher number of clusters in a set the more sparse the graph is, hence the less localized the contraction is.

The algorithms works on each possible *giveUpSet* separately. It builds the graph-like edges 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.

4.3 Specificity

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

$$\begin{gathered} Lion \sqsubseteq Mammal \\ Mammal \sqsubseteq Vertebrate \\ Vertebrate \sqsubseteq Animal \end{gathered}$$

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. However removing $Lion \sqsubseteq Mammal$ sounds much more reasonable than removing $Vertebrate \sqsubseteq Animal$: $Lion \sqsubseteq Mammal$ 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 $Lion \sqsubseteq Mammal$.

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 before considering GCIs that involve concepts at higher level.

The approach we follow in this study for adopting the preference of removing sentences that contain more specific concepts is by assigning weights to every GCI that reflects its

Algorithm 10 Computing localized hit

```
function GETLOCALIZEDHIT(giveUpSets)
    $setWithLeastClusters = null$
    $smallestNumber = \text{inf}$
    for $giveUpSet \in giveUpSets$ do
        $numOfClusters = getNumberOfClusters(giveUpSet)$
        if $numOfClusters < smallestNumber$ then
            $setWithLeastClusters = giveUpSet$
            $smallestNumber = numOfClusters$
        end if
    end for
    return $setWithLeastClusters$
end function
function GETNUMBEROFCLUSTERS(giveUpSet)
    for $i = 1$ to $size(giveUpSet)$ do
        $label[i] = i$
    end for
    for $i = 1$ to $size(giveUpSet)$ do
        $cluster[i] = 0$
    end for
    for $i = 1$ to $size(giveUpSet)$ do
        for $j = i + 1$ to $size(giveUpSet)$ do
            if giveUpSet[i] and giveUpSet[j] are connected then
                $label[j] = label[i]$
            end if
        end for
    end for
    for $i = 1$ to $size(giveUpSet)$ do
        $cluster[label[i]] = 1$
    end for
    for $i = 1$ to $size(giveUpSet)$ do
        $numberOfClusters = numberOfClusters + cluster[i]$
    end for
    return $numberOfClusters$
end function
```

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 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' roots. For simplicity, we assume that the $TBox$ is acyclic – so we avoid the problems that will be caused by loops.

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 4.4. 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
- $Lion \sqsubseteq Mammal$ 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 4.4, 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 $TBox$ T, the algorithm proceeds as in Algorithm 11.

Now we have a graph composed of concepts and their *children* relation; the nodes are concepts and the edges are the child-parent relationship between them. The roots are now concepts with no parents, and the leaves are concepts with no children. The GCIs involving leaves are assigned weight '0' and it increases by '1' every step towards the roots. So, the

Algorithm 11 Building the children graph

```
 1: function graphBuild(T)
 2:     $children(X) = \emptyset$, for every concept X in T
 3:     for every $A \sqsubseteq B \in T$ do
 4:         $children(B) = children(B) \cup \{A\}$
 5:     end for
 6: end function
```

roots are most general concepts, and the leaves are most specific. If a GCI contains concepts $\alpha \sqsubseteq \beta$, where α has no parents and β has no children, then it gets weight '0' according to Algorithm 12.

Algorithm 12 Assigning Weights

```
 1: function assignWeights(T)
 2:     for every $A \sqsubseteq B \in T$ do
 3:         $weight(A \sqsubseteq B) = GetWeight(A \sqsubseteq B)$
 4:     end for
 5: end function
 1: function getWeight($X \sqsubseteq Y$)
 2:     if $weight(X \sqsubseteq Y) \neq NULL$ then
 3:         return $weight(X \sqsubseteq Y)$
 4:     end if
 5:     if $children(X) = \emptyset$ 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($Z \sqsubseteq X$)
 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 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 sentences in the one with lower 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

```
function GETMOSTSPECIFICHIT(kernelset)
    min-hit-sets = min-hit-CUT(kernelSet)
    $min = \infty$
    hit = NULL
    w(s) = 0, for every $s \in$ min-hit-sets
    for every s in min-hit-sets do
        for every $A \sqsubseteq B$ in s do
            $w(s) = w(s) + weight(A \sqsubseteq B)$
        end for
        if w(s) < min then
            min = w(s)
            hit = s
        end if
    end for
    return hit
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 T and a GCI A, the contraction algorithm is given in Algorithm 14.

Algorithm 14 Contraction algorithm – modified

```
procedure CONTRACT(T, A)
    kernelset = PINPOINT(T, A)
    graphBuild(T)
    assignWeights(T)
    giveUpSet = GETMOSTSPECIFICHIT(kernelset)
    T = T / giveUpSet
end procedure
```

The contraction algorithms in this version guarantees the minimality and 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 more minimal and efficient 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 $O(nl)$, where n is the size of the TBox T and l is the depth of the subsumption graph. 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 T. Finally, the $getMostSpecificHit$ function takes $O(nm)$ steps in the worst case, where n is the size of the TBox 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

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. WhatâĂŹs new in Description Logics. *Informatik-Spektrum*, 34(5):434–442, 2011.

[4] Franz Baader, Carsten Lutz, and Anni-Yasmin Turhan. Small is Again Beautiful in Description Logics. *KI - KÃijnstliche Intelligenz*, 24(1):25–33, 2010.

[5] 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.

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

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

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

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

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

Appendix A

Code