A Tree Decomposition Approach to Automated Reasoning

MSc Thesis in Computer Science

Hilverd Reker
August 2007

Supervisors:
Prof. dr. Gerard R. Renardel de Lavalette
Dr. Rineke Verbrugge

Rijksuniversiteit Groningen
# Contents

## 1 Introduction
1.1 Decompositions of Logical Theories .................................................. 2
1.2 Thesis Overview ................................................................................. 3

## 2 Theory Graphs
2.1 Preliminaries .................................................................................... 4
2.2 Reasoning with Theory Graphs .............................................................. 5
2.3 Theory Trees ..................................................................................... 8
2.3.1 Lyndon’s interpolation property .................................................. 11

## 3 Decomposing Logical Theories
3.1 Tree Decompositions ......................................................................... 13
3.2 Answering Queries .......................................................................... 15

## 4 Adaptation to Semantic Tableaux
4.1 Semantic Tableaux ........................................................................... 17
4.1.1 Specification of a typical tableau algorithm .................................. 18
4.1.2 Soundness, termination, and completeness .................................. 18
4.1.3 Optimizations .............................................................................. 20
4.2 Semantic Tableaux for Theory Trees ................................................ 22
4.2.1 Specification of the tableau algorithm for theory trees ............... 25
4.2.2 Miscellaneous considerations ..................................................... 26

## 5 Application to Description Logics
5.1 Description Logics ............................................................................ 28
5.2 The Logic $\mathcal{ALC}$ ........................................................................ 28
5.2.1 Relation to other logics ............................................................... 29
5.3 Semantic Tableaux for $\mathcal{ALC}$ Theories ....................................... 30
5.3.1 Subset blocking ......................................................................... 33
5.3.2 Optimizations ............................................................................ 33
5.4 Semantic Tableaux for $\mathcal{ALC}$ Theory Trees ................................ 33
5.5 Combination with Existing Optimizations ....................................... 36

## 6 Experimental Setup and Results
6.1 Setup ............................................................................................... 38
6.1.1 Implementation .......................................................................... 38
6.1.2 Knowledge bases ...................................................................... 39
6.1.3 Tests ......................................................................................... 40
6.2 Results ............................................................................................ 40
6.2.1 Comparison with results obtained by Amir et al. ....................... 43
6.2.2 Results using Lyndon’s interpolation property ............................ 43
7 Conclusions

7.1 Summary of Contributions ............................................. 44
7.2 Related Work .............................................................. 45
7.3 Future Directions ......................................................... 45

A Example Tree Decompositions ........................................ 46
Abstract

Common-sense knowledge bases typically possess certain structural characteristics which randomly generated ones do not. It has been conjectured that we can exploit these characteristics for improving the efficiency of automated reasoning about such KBs. In a nutshell, the idea is to automatically decompose a KB into a particular kind of tree, and apply a restricted variant of a reasoning procedure such as resolution to that tree. Completeness of reasoning is ensured by the fact that resolution enjoys Craig’s interpolation property. A recent study by Amir et al. has experimentally applied a resolution-based version of this approach to first-order logic KBs, yielding promising results.

In this thesis, an adaptation of this method to semantic tableaux is proposed, and then specialized to a reasoning algorithm for the description logic $\mathcal{ALC}$. This algorithm was implemented, as well as a “plain” $\mathcal{ALC}$ reasoning procedure. By comparing the number of transformation steps taken by both algorithms for various queries, a first attempt at measuring their relative performance was made.

Since only few of the optimizations used in existing description logic reasoners were added to either implementation, the results are not yet very conclusive. However, they do appear encouraging enough to warrant further investigation of this method.
Acknowledgements

I am grateful to my first MSc thesis supervisor, Gerard Renardel, for having faith in me and being willing to consult with me on my thesis so many times. I also thank my second supervisor, Rineke Verbrugge, for her helpful comments on earlier versions of this thesis.
Chapter 1

Introduction

This thesis is concerned with automated reasoning, which might be summarized in one sentence as follows: we present a set of formulas expressed in some logical language as input to the computer, and then ask it to reason about those formulas. Such a set of formulas will be called a theory or a knowledge base (KB). For instance, we might instruct the computer to find out whether the formula \( p \) follows from the following theory:

\[
p \lor \neg q \quad s \lor q \quad \neg s \lor q \quad r \lor t \lor s.
\]

The computer then attempts to answer this question using some kind of reasoning procedure. Many popular reasoning procedures for computer use are based on proof systems such as resolution or semantic tableaux. Since resolution is an especially simple proof system, we will use it to clarify some ideas in this section.

Recall that resolution has only a single inference rule:

\[
\frac{a \lor \varphi \quad \neg a \lor \chi}{\varphi \lor \chi},
\]

where \( a \) is an atomic formula and \( \varphi, \chi \) are formulas. To reason about a theory, we first translate its formulas into clausal form (which the above theory is already in), and then recursively use the inference rule to derive new facts from already known facts. Such a derivation can be depicted as a proof tree, in which every leaf is an initially assumed formula, and every internal vertex is a formula inferred from its children by a single proof step. It turns out that there is a simple derivation of \( p \) from the theory given above, depicted in Figure 1.1(i).

![Figure 1.1: Derivations of \( p \).](image)

(i) Proof tree for \( p \).  
(ii) Another proof tree for \( p \).

However, by making inferences differently we could also have constructed the bigger proof shown in Figure 1.1(ii), which unfortunately would have been more work. How to reason efficiently (i.e. what strategy to employ) is, in general, a very complicated question. For the
case of resolution, all kinds of variants have been proposed to improve the performance of automated reasoning. A well-known example is set-of-support resolution, which restricts the formulas that may be used as premises when applying an inference rule, while still maintaining completeness\(^1\).

### 1.1 Decompositions of Logical Theories

This thesis is based on a recently proposed method of reasoning [Ami02] which also (basically) works by restricting the premises allowed in proof steps. The method is oriented towards common-sense KBs, i.e. human-written theories such as, say, medical ontologies. It relies on certain structural characteristics that these KBs typically possess, as opposed to randomly generated KBs.

Consider the following tiny common-sense KB, describing a few university-related concepts expressed in first-order predicate logic (formulas are implicitly universally quantified):

\[
\begin{align*}
\text{student}(x) & \rightarrow \text{undergradStudent}(x) \lor \text{gradStudent}(x), \\
\text{student}(x) \land \text{hasBSc}(x) & \rightarrow \text{gradStudent}(x), \\
\text{student}(x) & \rightarrow \exists y (\text{course}(y) \land \text{enrolledIn}(x, y)), \\
\text{course}(x) & \rightarrow \exists y \text{ teaches}(y, x), \\
\text{cs_course}(x) & \rightarrow \text{course}(x), \\
\text{math_course}(x) & \rightarrow \text{course}(x).
\end{align*}
\]

One may observe that this KB can be decomposed roughly into two parts: the first consists of student-related statements, whereas the second describes course-oriented ones. We might depict such a decomposition as the following graph:

The edge label “course” indicates which symbols are shared between both subtheories. More generally, we can decompose large theories into theory graphs in the manner suggested above. As proposed in [Ami02, Ch. 4], we then reason using such theory graphs by repeatedly (and nondeterministically) performing the following steps:

- make inferences locally in a vertex \(v\), adding the conclusions to \(v\),
- copy an (inferred) formula \(\varphi\) from a vertex \(x\) to an adjacent vertex \(y\), but only if every symbol in \(\varphi\) also occurs in \(y\).

Provided the theory graph satisfies certain properties, one can show that this proof procedure is still refutation-complete. Furthermore, for reasons explained later, we are interested in theory graphs which have many, small vertices (i.e. subtheories), and small links (i.e. shared vocabularies) between adjacent vertices. Typically, these requirements can be met for common-sense KBs, as those are often made up of different “modules” of knowledge that are more or

---

\(^1\)Or actually refutation-completeness: if a theory is unsatisfiable, we can prove this using resolution.
less separate. This means e.g. that many combinations of symbols will never occur together in a single formula. By contrast, randomly generated KBs, especially those representing “hard” satisfiability problems, tend to be much denser in this respect.

A specific kind of theory graph, to be called a theory tree in this thesis, can automatically be generated using a tree decomposition algorithm. Although computing a tree decomposition may be relatively costly, in practice we often have a fixed theory which we want to answer many queries about. This means we can compute an initial decomposition for this theory, and then when given a query, merely add it (or rather its negation) to a suitable vertex.

In [MMAU03], promising results are reported for experiments using the method outlined above. Those experiments were done on first-order logic KBs using different variants of resolution; it turned out that reasoning was only more efficient when combined with other methods such as set-of-support resolution or ordered resolution. The focus in [Ami02] is also on first-order logic and resolution.

In this thesis, an adaptation of this method to semantic tableaux is proposed. This adaptation is then specialized to a tableau algorithm for the description logic $\mathcal{ALC}$. The performance of this new algorithm is compared empirically to that of a plain $\mathcal{ALC}$ tableau algorithm, to measure whether the efficiency of reasoning is indeed improved. The results appear to confirm this, but are very preliminary: real-world reasoners for description logics such as $\mathcal{ALC}$ incorporate a lot of optimizations. Almost none of these were taken into account. However, it can be argued that many of these optimizations may benefit the “plain” algorithm as well as the new one.

1.2 Thesis Overview

First, Chapters 2 and 3 present an account of the relevant parts of Amir’s PhD thesis [Ami02] and of [MMAU03]. Almost all of the material in these chapters is not new (cf. § 7.1 for an overview of what is), but much of it has been extensively reformulated. The aim of this was to make things more precise and easier to grasp.

Chapter 4 then proposes an adaptation of Amir’s reasoning method to semantic tableaux instead of resolution. This adaptation is given on a general level, and should be applicable to any logic for which tableau algorithms exist.

In Chapter 5, the description logic $\mathcal{ALC}$ is introduced, and a tableau algorithm for $\mathcal{ALC}$ is outlined which is based on the newly suggested reasoning method. There is also a brief discussion of which optimizations in existing description logic reasoners might be added to the new $\mathcal{ALC}$ algorithm.

The subject of Chapter 6 is the setup and results of some experiments with the new $\mathcal{ALC}$ reasoning procedure, compared to a plain $\mathcal{ALC}$ algorithm. Finally, Chapter 7 summarizes the contributions of this thesis and mentions some future directions.
Chapter 2

Theory Graphs

This chapter presents my own account of the approach described by Amir in [Ami02, § 4.2]. The main reason for this is to try and formulate certain concepts more clearly.

2.1 Preliminaries

First of all, we work in some logic with language $\mathcal{L}$ — that is, $\mathcal{L}$ is the set of syntactically valid formulas of the logic. For now, we will limit ourselves to propositional logic.

Definition 2.1.1 (Theory, signature, language). Let $\mathcal{L}$ be a logical language.

(i) A theory (in $\mathcal{L}$) is a set of $\mathcal{L}$-formulas (not necessarily closed under deduction). We will denote theories by $\Pi, \Gamma$. Formulas will be denoted by $\alpha, \beta, \gamma, \varphi, \chi, \psi$.

(ii) A signature (in $\mathcal{L}$) is a set of non-logical symbols in $\mathcal{L}$. We will denote non-logical symbols by $a, p, q, r, s$ and signatures by $\Sigma, \Omega$. The signature of $\Pi$, denoted $\text{sig}(\Pi)$, is the set of non-logical symbols in $\Pi$. For formulas, we will write $\text{sig}(\varphi)$ instead of $\text{sig} \{ \varphi \}$. The formulas $\top$ and $\bot$ are considered to have an empty signature.

(iii) The language of $\Sigma$, denoted $\text{lang}(\Sigma)$, is the set of all formulas in $\mathcal{L}$ constructible from $\Sigma$. The language of a theory $\Pi$ is $\text{lang}(\text{sig}(\Pi))$, and the language of a formula $\varphi$ is $\text{lang}(\text{sig}(\varphi))$.

For instance, the signature of the example theory given on page 1 is $\{ p, q, r, s, t \}$. In this work, we will consider only finite signatures and finite theories. The latter implies that we may view a theory as the conjunction of its elements (i.e. a single formula) rather than set, and we will do so when convenient.

Definition 2.1.2 (Derivability relation). A derivability relation for a logic with language $\mathcal{L}$ is a relation $\vdash \subseteq \wp(\mathcal{L}) \times \wp(\mathcal{L})$, where $\wp(\mathcal{L})$ is the power set of $\mathcal{L}$.

For instance, we can define $\vdash$ to be the resolution derivability relation as follows. Let $\Pi \vdash^1 \Gamma$ iff each formula $\varphi \in \Gamma$ is either in $\Pi$, or it is of the form $a \lor \varphi$ such that $a \in \Pi$ and $a \lor \varphi \not\in \Pi$, for some atomic formula $a$. Then $\vdash$ is the transitive closure of $\vdash^1$.

When using resolution, we first translate the given propositional theory $\Pi$ into conjunctive normal form (or clausal form for first-order theories).

We prefer to write $a \vdash \beta$ instead of $\{ a \} \vdash \{ \beta \}$, and $\Pi \vdash \varphi$ instead of $\Pi \vdash \{ \varphi \}$. Furthermore, we usually write $\Pi, \varphi \vdash \chi$ instead of $\Pi \cup \{ \varphi \} \vdash \chi$. As one would expect, we assume that whatever derivability relation we use is reflexive and transitive. The following are two more key properties of the derivability relations we are interested in.
Definition 2.1.3 (Craig’s interpolation property). A derivability relation $\vdash$ satisfies Craig’s interpolation property \([\text{Cra57a, Cra57b}]\) if, whenever $\alpha \vdash \beta$, there is an interpolant $\gamma$ with $\alpha \vdash \gamma$, $\gamma \vdash \beta$, and $\gamma \in \text{lang}(\alpha) \cap \text{lang}(\beta)$.

Definition 2.1.4 (Deduction property). A derivability relation $\vdash$ satisfies the deduction property if, for all formulas $\varphi, \chi, \psi$, we have $\varphi \wedge \chi \vdash \psi$ iff $\varphi \vdash \chi \rightarrow \psi$.

One often considers a derivability relation $\vdash$ in conjunction with a validity relation $|=\,$ for which we have $\Pi |= \varphi$ iff $\Pi \vdash \varphi$. When the “if” direction holds, we say that $\vdash$ is sound w.r.t $|=\,$, whereas the “only if” direction is called completeness of $\vdash$ w.r.t $|=\,$. It should be noted that resolution is not complete in this sense, but rather refutation-complete. The latter means that whenever a theory $\Pi$ is unsatisfiable (i.e. $\Pi |= \bot$), then this can be shown using resolution: $\Pi \vdash \bot$. To see that resolution is not complete, consider that it cannot derive $p \lor q$ from $p$.

As indicated before, in what follows we will be considering graphs of theories, i.e. graphs in which every vertex is a theory.

Definition 2.1.5 (Graph). A graph is a pair $\langle V, E \rangle$, where $V$ is a set of vertices and $E \subseteq V^2$ a set of edges, with every edge a pair of distinct vertices. If $E$ is required to be symmetric, then the graph is undirected, otherwise it is directed.

We will denote edges by $c, d, e$ and vertices by $v, w, x, y$. Sets of vertices will be denoted by $V, W, X, Y$, and sets of edges by $E$.

Definition 2.1.6 (Path). Let $G = \langle V, E \rangle$ be a graph. A path (from $v_1$ to $v_n$) in $G$ is a finite sequence $v_1, \ldots, v_n$ ($n \geq 1$) of vertices in $V$ such that every consecutive pair of vertices is an edge in $E$.

Definition 2.1.7 (Theory graph). A theory graph is a labeled graph $G = \langle V, E, \pi \rangle$ where $V \neq \emptyset$, $E \subseteq V^2$, and $\pi : V \rightarrow \wp(L)$ assigns a theory to each vertex of $G$.

We often identify a vertex $v$ with its theory $\pi(v)$. This convention also applies when drawing theory graphs, as exemplified in Figure 2.1.

```
s \lor q
-\neg s \lor q

r \lor t \lor s

p \lor -q
```

Figure 2.1: A theory graph.

Observe that the union of the above vertices gives us the example theory given on page 1. Note also that formulas may occur in (the labels of) multiple vertices: whether or not we do this in practice is elaborated on in § 6.1.1.

### 2.2 Reasoning with Theory Graphs

The basic reasoning algorithm for theory graphs can be described roughly as follows. We repeatedly, nondeterministically take either of the following actions:

1. we reason locally within a vertex $v$, adding the conclusions to $\pi(v)$,
(2) under certain circumstances, we may copy a formula from a vertex to an adjacent one.

This process is repeated until either a desired conclusion has been found (e.g. $\bot$ was added to some vertex), or until no more new steps can be taken. To express under what circumstances a formula may be copied between adjacent vertices, an edge labeling of the theory graph is used.

**Definition 2.2.1 (Edge labeling).** An edge labeling for a theory graph $G$ is a function $\sigma$ labeling each edge $(x, y)$ of $G$ with a signature $\sigma(x, y)$.

We will usually combine a theory graph and an edge labeling for it in the same diagram. Basically, an edge labeling will be used to restrict the kinds of formulas that may be sent from one vertex to another. Exactly how this is done will become apparent in a moment; for convenience, we first introduce a notation for adding a formula to a vertex.

**Definition 2.2.2 (Adding formulas to a theory graph).** If $G = (V, E, \pi)$ is a theory graph, then $G[x \leftarrow \varphi]$ is the theory graph $\langle V, E, \pi' \rangle$, where $\pi'$ is defined as

$$\pi'(v) = \begin{cases} \pi(v) \cup \{\varphi\} & \text{if } v = x, \\ \pi(v) & \text{otherwise} \end{cases}.$$ (Note that if $\sigma$ is an edge labeling for $G$, then it is also an edge labeling for $G[x \leftarrow \varphi]$.) The binary relation $\vdash_\sigma$ on the set of all theory graphs, given in Definition 2.2.4, establishes which steps may be taken by the reasoning algorithm for theory graphs. Before giving this definition, we need one more preliminary concept: as we will often need to refer to the “contents” of a theory graph $G = (V, E, \pi)$, we use the range $\text{rng}(\pi)$ of $\pi$ for this purpose.

**Definition 2.2.3 (Image, range).** Let $R$ be a relation from $A$ to $B$, and $A' \subseteq A$.

(i) The image of $A'$ under $R$, denoted $R[A']$, is the set $\{b \mid R(a, b) \text{ for some } a \in A'\}$.

(ii) The range of $R$, denoted $\text{rng}(R)$, is $R[A]$.

**Definition 2.2.4 (Relation $\vdash_\sigma$).** Suppose $\vdash$ is a derivability relation and $\sigma$ an edge labeling.

(i) The relation $\vdash_\sigma$ is the reflexive transitive closure of the binary relation $\vdash_1^\sigma$ defined as follows. If $\sigma$ is an edge labeling for a theory graph $G = (V, E, \pi)$, and $G' = G[y \leftarrow \varphi]$ for some $y \in V$ and $\varphi \notin \pi(y)$ such that at least one of the following holds:

(a) $\pi(y) \vdash \varphi$,

(b) for some $(x, y) \in E$, we have $\varphi \in \pi(x)$ and $\text{sig}(\varphi) \subseteq \sigma(x, y)$,

then $G \vdash_1^\sigma G'$.

(ii) We write $G \vdash_\sigma \varphi$ if there is a $G' = (V', E', \pi')$ with $G \vdash_\sigma G'$ and $\varphi \in \text{rng}(\pi')$.

We call $\vdash_\sigma$ a graph derivability relation.

This means that whereas $\vdash$ is a relation on theories, $\vdash_\sigma$ is a relation on theory graphs. The reasoning algorithm simply computes nondeterministically a maximal $\vdash_1^\sigma$-path. Unless noted otherwise, we shall take $\vdash$ to be the resolution derivability relation. For instance, starting with the theory graph shown in Figure 2.1, this algorithm can derive $p$ as depicted in Figure 2.2.

In the theory graph(s) depicted in this particular example, each edge $(x, y)$ has been labeled with $\text{sig}(\pi(x)) \cap \text{sig}(\pi(y))$. Added formulas are marked with a $\ast$ sign. Initially, in Figure 2.2(i), no formula can be copied to an adjacent vertex because the edge labeling $\sigma$ does not permit it (cf. Definition 2.2.4 (i)(b)). The only possible step is to infer $q$ from $s \lor q$ and $\neg s \lor q$, as these
formulas are located in the same vertex, and add \( q \) to that vertex. The result of this is shown in Figure 2.2(ii). Next, we are able to copy \( q \), which we just derived, to the lower right vertex; see Figure 2.2(iii). Finally, we resolve the clauses \( p \lor \neg q \) and \( q \) in the lower right vertex to produce \( p \).

Note that in any \( \vdash_\sigma \)-path for any theory graph, only the theories of the vertices are changed (namely, they are extended) but the structure of the graph remains the same. Naturally, in the case of resolution we are trying to derive \( \bot \), and the algorithm should stop as soon as \( \bot \) has been added to some vertex.

For convenience, we may write \( G \vdash \varphi \) instead of \( \text{rng}(\pi) \vdash \varphi \) for any theory graph \( G = \langle V, E, \pi \rangle \), by abuse of notation.

Obviously, \( \vdash_\sigma \) can be regarded as a restricted version of the underlying \( \vdash \) relation, as it does not allow just any formulas to be used as the premises in a derivation step, but only those that are in the same vertex. This means the number of possible proof trees, i.e. the search space, is reduced. By how much depends on the edge labeling, \( \sigma \).

As mentioned, we assume that the \( \vdash \) relation used is sound and (refutation-)complete w.r.t. the corresponding \( \models \)-relation. The question then becomes under what circumstances \( \vdash_\sigma \) is sound and complete w.r.t. \( \vdash \). The first, soundness, is easy to prove.

**Theorem 2.2.5 (Soundness).** Let \( \vdash \) be a derivability relation and \( \sigma \) an edge labeling for the theory graph \( G = \langle V, E, \pi \rangle \). If \( G \vdash_\sigma \varphi \), then \( G \vdash \varphi \).

**Proof.** For any theory graphs \( G_j = \langle V_j, E_j, \pi_j \rangle \) and \( G_k = \langle V_k, E_k, \pi_k \rangle \), if \( G_j \vdash_\sigma G_k \) then clearly \( \text{rng}(\pi_j) \vdash \text{rng}(\pi_k) \). As \( \vdash_\sigma \) is the reflexive transitive closure of \( \vdash_1 \) and, one may assume, \( \vdash \) is also reflexive and transitive, the previous statement holds for \( \vdash_\sigma \) instead of \( \vdash_1 \) as well. Now suppose \( G' = \langle V', E', \pi' \rangle \) is a theory graph with \( G \vdash_\sigma G' \) and \( \varphi \in \text{rng}(\pi') \). As we have just established, \( \text{rng}(\pi) \vdash \text{rng}(\pi') \), so since \( \text{rng}(\pi') \vdash \varphi \) we get \( \text{rng}(\pi) \vdash \varphi \).
Note that if $\vdash$ is the resolution derivability relation, then for any (finite) theory $\Pi$, there are only finitely many formulas $\varphi$ with $\Pi \vdash \varphi$. Informally speaking, this is because we cannot create arbitrarily long formulas by deriving $\varphi \lor \psi$ from $\varphi$. This property of resolution can be used to show that the reasoning algorithm must terminate.

**Theorem 2.2.6 (Termination).** Let $\sigma$ be an edge labeling for the theory graph $G = \langle V, E, \pi \rangle$, and suppose $\vdash$ is a derivability relation such that there are only finitely many formulas $\varphi$ with $\text{rng}(\pi) \vdash \varphi$. Then every maximal $\vdash^1_{\sigma}$-path beginning with $G$ is finite, with $\vdash^1_{\sigma}$ as in Definition 2.2.4.

**Proof.** In every step of a $\vdash^1_{\sigma}$-path, we add to some vertex $v$ a new consequence $\varphi$ of $\text{rng}(\pi)$, i.e. a $\varphi$ not yet “in $v$” with $\text{rng}(\pi) \vdash \varphi$ (cf. Theorem 2.2.5). As there are only finitely many vertices and finitely many consequences of $\text{rng}(\pi)$, eventually we will (at the most) have added all of these consequences to every vertex, at which point there are no more $\vdash^1_{\sigma}$-successors. ☐

The next section covers the more interesting question: which graph derivability relations $\vdash_{\sigma}$ are refutation-complete for which theory graphs $G$?

### 2.3 Theory Trees

We begin by defining what is meant by “completeness” of the reasoning algorithm.

**Definition 2.3.1 ($\Sigma$-complete for $G$).** Suppose $\vdash_{\sigma}$ is a graph derivability relation for a theory graph $G = \langle V, E, \pi \rangle$, and $\Sigma$ is a signature. We say that $\vdash_{\sigma}$ is $\Sigma$-complete for $G$ if for all $\varphi \in \text{lang}(\Sigma)$ with $G \vdash \varphi$, we have $G \vdash_{\sigma} \varphi$.

For most practical purposes, all we really need is refutation-completeness. That is, it suffices to be able to show unsatisfiability, as most interesting queries can be reduced to this question. For instance, to see whether $\varphi$ is a consequence of a theory $\Pi$, i.e. $\Pi \vdash \varphi$, we can test whether $\Pi \cup \{\neg \varphi\}$ is satisfiable.

**Definition 2.3.2 (Refutation-complete for $G$).** A graph derivability relation $\vdash_{\sigma}$ is refutation-complete for a theory graph $G$ if $\vdash_{\sigma}$ is $\emptyset$-complete for $G$.

Obviously, if we choose $\sigma$ such that all formulas are allowed to be copied between vertices, we have completeness, since then any $\vdash$-derivation is also a $\vdash_{\sigma}$-derivation. However, it turns out that a limited kind of $\sigma$ still yields refutation-completeness for certain kinds of graphs. Before discussing this further, we first look at some typical situations in which $\vdash_{\sigma}$ is not complete.

![Figure 2.3](image-url)

Figure 2.3: Some theory graphs for which $\vdash_{\sigma}$ is incomplete.

Figure 2.3(i) shows the simplest possible theory graph with an edge labeling $\sigma$ for which $\vdash_{\sigma}$ is not refutation-complete. Although the graph contains both $p$ and $\neg p$, we need to get them in
the same vertex in order to infer $\bot$, which cannot be achieved. A more complicated situation is depicted in Figure 2.3(ii). Initially, we cannot copy any formulas between any vertices. Resolving the clauses in the left or the right vertex is pointless, as this produces the tautologies $p \lor \neg p$ or $q \lor \neg q$. However, taken together the four formulas in the theory graph do prove $\bot$, but the reasoning algorithm cannot discover this.

Following the approach taken by Amir et al., we will now focus our attention on a special kind of theory graph, namely one which is a directed tree and whose edge labeling meets certain requirements.

**Definition 2.3.3 (Theory tree).** A theory tree is a theory graph $T$ with a special vertex, called the root, to which every vertex has exactly one path (i.e. $T$ is a rooted directed tree).

**Definition 2.3.4 (Proper labeling).** Let $T = \langle V, E, \pi \rangle$ be a theory tree and $\sigma$ an edge labeling for $T$ satisfying the following: if $e = \langle x, y \rangle \in E$ and $X$ (including $x$) resp. $Y$ are the sets of vertices into which $e$ partitions $V$, i.e. the respective sides of $e$ in $T$, then

$$\text{sig} \left( \bigcup_{v \in X} \pi(v) \right) \cap \text{sig} \left( \bigcup_{v \in Y} \pi(v) \right) \subseteq \sigma(x, y).$$

In that case, $\sigma$ is a proper labeling for $T$.

A few examples of properly labeled theory trees are shown in Figure 2.4. The graph in Figure 2.4(i) is a simple “linear” tree containing our example theory as used earlier. It is easy to see that the algorithm can use this tree to derive $p$: first we derive $q$ in the middle vertex, after which we copy $q$ to the top vertex, where $q$ may then be resolved with $p \lor \neg q$ to produce $p$. However, if we swap the top and bottom vertices and then (minimally) properly label the tree, as depicted in Figure 2.4(ii), we can no longer derive $p$. This is prevented by the directions of the edges, which further limit the derivations we may construct.

However, as we will see in the remainder of this section, any properly labeled theory tree is refutation-complete. For instance, if we take the tree from Figure 2.4(ii) and add $\neg p$ to the bottom vertex, we get the theory tree in Figure 2.4(iii). Here we can derive $\neg q$ in the bottom vertex, copy that to the middle one, and then obtain $\bot$ there. The proof of refutation-completeness uses the following small lemma.

**Lemma 2.3.5.** Let $T = \langle V, E, \pi \rangle$ be a theory tree and $\sigma$ a proper labeling for $T$. Suppose $\langle x, y \rangle \in E$ and

$$\varphi \in \text{lang} \left( \bigcup_{v \in X} \pi(v) \right) \cap \text{lang} \left( \bigcup_{v \in Y} \pi(v) \right),$$

Figure 2.4: Some properly labeled theory trees.
where \(X\) resp. \(Y\) are the sets of vertices into which \((x,y)\) partitions \(V\). Then \(\sigma\) is a proper labeling for \(T[x \leftarrow q][y \leftarrow q]\).

**Proof.** Obvious.

Rather than showing refutation-completeness, we prove something slightly different: if \(\sigma\) is a proper labeling for a theory tree \(T\), then \(\vdash_{\sigma}\) is \(\Sigma\)-complete for \(T\), where \(\Sigma\) is the signature of \(T\)’s root. This claim was hinted at by Figures 2.4(i) and 2.4(ii).

**Definition 2.3.6 (Restriction).** Let \(f : A \rightarrow B\) be a function and \(A' \subseteq A\). The restriction \(f|_{A'}\) of \(f\) to \(A'\) is the function \(f|_{A'} : A' \rightarrow B\) defined by \(f|_{A'}(a) = f(a)\) for all \(a \in A'\).

**Theorem 2.3.7 (Completeness).** Suppose \(\vdash\) is a derivability relation satisfying the deduction and interpolation properties. Let \(T = (V,E,\pi)\) be a theory tree with root \(v_0\), and \(\sigma\) a proper labeling for \(T\). Then \(\vdash_{\sigma}\) is \(\text{sig}(\pi(v_0))\)-complete for \(T\).

**Proof.** Suppose \(\varphi \in \text{lang}(\pi(v_0))\). We need to show that if \(\text{rng}(\pi) \vdash \varphi\), then \((V,E,\pi)\) \(\vdash_{\sigma}\) \(\varphi\). It suffices to show the following claim: if \(T' = (V,E,\pi')\) is a theory tree for which \(\sigma\) is a proper labeling, \(V' \subseteq V\) induces a subtree of \(T\), \(v_0 \in V'\), and \(\text{rng}(\pi'|_{V'}) \vdash \varphi\), then \((V,E,\pi)\) \(\vdash_{\sigma}\) \(\varphi\). Taking \(V' := V\) and \(\pi' := \pi\), this would prove the theorem. We proceed by induction on \(|V'|\). Note that if \(\varphi\) is already in \(\text{rng}(\pi'|_{V'})\) then \((V,E,\pi)\) \(\vdash_{\sigma}\) \(\varphi\) is trivial, so we assume this is not the case.

**Basis.** If \(V' = \{v_0\}\) then \(\pi'(v_0) \vdash \varphi\), so \((V,E,\pi)\) \(\vdash_{\sigma,1}\) \(\pi'[v_0 \leftarrow \varphi]\), with \(\vdash_{\sigma,1}\) as in Definition 2.2.4. Since \(\vdash_{\sigma}\) is the reflexive transitive closure of \(\vdash_{\sigma,1}\), it follows that \((V,E,\pi)\) \(\vdash_{\sigma}\) \(\varphi\).

**Step.** Suppose \(x\) is a leaf of the subtree induced by \(V'\). Then we have \(\pi'(x) \land \bigwedge_{v \in V'\setminus\{x\}} \pi'(v) \vdash \varphi\), which may be restated as \(\pi'(x) \vdash \bigwedge_{v \in V'\setminus\{x\}} \pi'(v) \rightarrow \varphi\) by the deduction property. According to Craig’s interpolation property, there is a \(\gamma\) with

(a) \(\gamma \in \text{lang}(\pi'(x)) \cap \text{lang}\left(\bigwedge_{v \in V'\setminus\{x\}} \pi'(v) \rightarrow \varphi\right)\),

(b) \(\pi'(x) \vdash \gamma\),

(c) \(\gamma \vdash \bigwedge_{v \in V'\setminus\{x\}} \pi'(v) \rightarrow \varphi\).

Since \(\varphi \in \text{lang}(\pi'(v_0))\) and \(v_0 \in V'\setminus\{x\}\), we can rewrite (a) to

\[
\gamma \in \text{lang}(\pi'(x)) \cap \text{lang}\left(\bigwedge_{v \in V'\setminus\{x\}} \pi'(v)\right). \tag{2.3.1}
\]

Furthermore, again using the deduction property we can restate (c) as

\[
\gamma \land \bigwedge_{v \in V'\setminus\{x\}} \pi'(v) \vdash \varphi. \tag{2.3.2}
\]

From (b) it follows that \((V',E,\pi)\) \(\vdash_{\sigma,1}\) \(T'[x \leftarrow \gamma]\). Now let \(y\) be the parent of \(x\). Because \(\sigma\) is a proper labeling for \(T'\), we have \(\sigma(x,y) \subseteq \text{sig}(\bigcup_{v \in X} \pi'(v)) \cap \text{sig}(\bigcup_{v \in Y} \pi'(v)) \subseteq \sigma(x,y)\), with \(X,Y\) certain sets of vertices (i.e. both “sides” of \((x,y)\)) such that \(x \in X\) and \(V'\setminus\{x\} \subseteq Y\). Hence,

\[
\text{sig}(\pi'(x)) \cap \text{sig}\left(\bigcup_{v \in V'\setminus\{x\}} \pi'(v)\right) \subseteq \sigma(x,y) .
\]

Considering (2.3.1), this means that \((V',E,\pi)\) \(\vdash_{\sigma,1}\) \(T'[x \leftarrow \gamma][y \leftarrow \gamma]\), since \(\text{sig}(\gamma) \subseteq \sigma(x,y)\). According to Lemma 2.3.5, \(\sigma\) is a proper labeling for \((V',E,\tilde{\pi})\). Now suppose \((V',E,\tilde{\pi})\) \(\vdash \varphi\), as \(\gamma \in \tilde{\pi}(y)\). Noting that \(V'\setminus\{x\}\) induces a subtree of \(T\), the induction hypothesis gives \((V,E,\tilde{\pi})\) \(\vdash_{\sigma}\) \(\varphi\). By transitivity of \(\vdash_{\sigma}\), this completes the proof.
This proof is somewhat simpler than Amir’s; his one shows completeness of the algorithm w.r.t. $\models$, whereas the above proof shows completeness of the algorithm w.r.t. $\vdash$ (recall that we assume $\vdash$ is complete w.r.t. $\models$).

So we certainly also have refutation-completeness for properly labeled theory trees. More generally, it follows that any theory graph with a properly labeled spanning tree is refutation-complete.

As mentioned before, the reasoning algorithm essentially does nothing more than compute a maximal $\vdash_\sigma$-path, which is a sequence of theory graphs. Although the contents of each successive theory graph are extended over the course of such a sequence, their structure remains the same. Therefore, we may also look at the reasoning algorithm as follows. Instead of a theory graph $G = \langle V, E, \pi \rangle$ together with an edge labeling $\sigma$, suppose we are given an edge-labeled graph $G = \langle V, E, \sigma \rangle$ and a “vertex-labeled theory”. By the latter we mean a set of pairs $\langle v, \varphi \rangle$ where $v \in V$. For the case of resolution, suppose there are two derivation rules:

\[
\begin{align*}
\langle v, a \lor \varphi \rangle & \quad \langle v, \neg a \lor \chi \rangle \\
\langle v, \varphi \lor \chi \rangle
\end{align*}
\]

(2.3.3)

In other words, we are considering a variant of resolution where clauses may be resolved only if they occur in the same vertex, and copying of clauses between adjacent vertices is also permitted (under certain conditions). The reasoning algorithm then constructs a derivation according to (2.3.3) and (2.3.4).

This way, we would basically view our reasoning method as a labelled deductive system $[\text{Gab96}]$. The presentation in this thesis does not take this point of view, but this is how things were implemented in practice (cf. §4.2.2).

If we use first-order logic instead of just propositional logic, a number of complications arise. For instance, one has to decide which constant and function symbols should be sent between vertices. Consider the following theory tree as an example, with formulas in clausal form:

\[
\begin{align*}
&\neg A(x) \\
&\quad A(c)
\end{align*}
\]

Obviously, the predicate symbol $A$ should be in the labeling of the edge. However, so should the constant symbol $c$, otherwise the algorithm cannot discover that we have a contradiction here. In general, a simple way to resolve this is to allow all constant and function symbols to be sent, but that does not seem very efficient. A solution suggested by Amir is to use unskolemization (a.k.a. reverse skolemization). In this case, unskolemization would translate $A(c)$ into $\exists y A(y)$, which can be sent to the top vertex.

We will not discuss the issues that arise when the algorithm has to deal with first-order logic any further here; cf. [Ami02, § 4.3.1].

### 2.3.1 Lyndon’s interpolation property

Lyndon’s interpolation theorem [Lyn59] is a refinement of Craig’s: instead of only atomic formulas, it also considers negations of atomic formulas. In other words, it considers both positive
and negative occurrences of symbols. Assume we call such occurrences in a formula \( \varphi \) the \textit{parameters} of \( \varphi \). For completeness, these are defined as follows:

\[
\begin{align*}
\text{par}(p) &= \{p^+\} \\
\text{par}(\neg \varphi) &= \{p^- \mid p^+ \in \text{par}(\varphi)\} \cup \{p^+ \mid p^- \in \text{par}(\varphi)\} \\
\text{par}(\varphi \boxdot \psi) &= \text{par}(\varphi) \cup \text{par}(\psi) \quad \text{with} \quad \boxdot \in \{\lor, \land, \rightarrow, \leftrightarrow\}.
\end{align*}
\]

We may then formulate Lyndon’s interpolation theorem as follows.

\textbf{Definition 2.3.8 (Lyndon’s interpolation property).} A derivability relation \( \vdash \) satisfies Lyndon’s interpolation property if, whenever \( \alpha \vdash \beta \), there is an interpolant \( \gamma \) with \( \alpha \vdash \gamma \), \( \gamma \vdash \beta \), and \( \text{par}(\gamma) \subseteq \text{par}(\alpha) \cap \text{par}(\beta) \).

It seems plausible that the performance of the reasoning algorithm could be further improved by employing this interpolation property instead of Craig’s. This is elaborated on in § 6.2.2.
Chapter 3

Decomposing Logical Theories

An analysis in [Ami02, § 4.4.2] shows that the “decomposition-based” reasoning procedure will likely work well for theory trees with many, small vertices (i.e. theories), and small “links” between vertices. In [MMAU03], results from experiments on first-order logic KBs are reported which appear to confirm this. As indicated in § 1.1, we are assuming that these properties are attainable when decomposing real-world, common-sense KBs. This section elaborates on how these decompositions are created.

3.1 Tree Decompositions

**Definition 3.1.1 (Symbols graph).** The symbols graph for a theory Π is the graph \( G = \langle \text{sig}(\Pi), E \rangle \) such that \( \langle p, q \rangle \in E \) iff there is a \( \varphi \in \Pi \) with \( p, q \in \text{sig}(\varphi) \).

For example, the theory \( \Pi = \{ p \lor \neg q, s \lor q, \neg s \lor q, r \lor t \lor s \} \) from our earlier examples has the symbols graph shown in Figure 3.1.

![Figure 3.1: A symbols graph.](image)

**Definition 3.1.2 (Tree decomposition).** For our purposes, a tree decomposition [Bod97] of a symbols graph \( G = \langle V_G, E_G \rangle \) is an (undirected) tree \( T = \langle V, E \rangle \) in which each vertex is a signature and

1. \( \bigcup_{v \in V} = V_G \),
2. for all \( \langle p, q \rangle \in E_G \), there exists a \( v \in V \) with \( p, q \in v \),
3. if a symbol \( p \) occurs in two vertices \( x, y \in V \), then it appears in all vertices on the path between \( x \) and \( y \) in \( T \); this is called the running intersection property.

A trivial tree decomposition for the symbols graph in Figure 3.1 is \( \langle \{ p, q, r, s, t, \varnothing \} \rangle \): a single vertex containing all of the symbols. Two more interesting tree decompositions are depicted in Figure 3.2.

13
Note that, e.g., the following is not a tree decomposition:

The symbol \( s \) occurs in the left vertex and in the right one, but not in the middle vertex which is on the path between the two. Hence the tree above does not satisfy the running intersection property.

From now on, we will work only with theory trees created in the following manner.

**Definition 3.1.3 (Theory tree for \( \Pi \)).** Let \( \Pi \) be a theory. If \( T = \langle V, E, \pi \rangle \) is a theory tree such that \( \langle V, E \rangle \) is a tree decomposition of the symbols graph for \( \Pi \), and

\[
\pi(v) := \{ \varphi \in \Pi \mid \text{sig}(\varphi) \subseteq v \},
\]

for all \( v \in V \), then \( T \) is a *theory tree* for \( \Pi \).

For instance, based on the resp. symbols graphs in Figure 3.2, we might decompose our example theory into the resp. (undirected) theory trees shown in Figure 3.3.

The next two lemmas show that constructing a theory tree for \( \Pi \) in this way indeed results in every \( \varphi \in \Pi \) being assigned to at least one \( \pi(v) \). As mentioned previously, this may also result in some formulas being assigned to multiple vertices (cf. § 6.1.1).

**Lemma 3.1.4.** Suppose \( \langle V, E \rangle \) is a tree decomposition of the symbols graph for \( \Pi \), and let \( \varphi \in \Pi \). Then there is a \( \Sigma \in V \) with \( \text{sig}(\varphi) \subseteq \Sigma \).

**Proof.** By induction on \( n = |\Sigma| \), we show that for every \( \Sigma \subseteq \text{sig}(\varphi) \), there is an \( \Omega \in V \) with \( \Sigma \subseteq \Omega \).

**Base.** The cases \( n = 0, 1, 2 \) are obvious.
Step. Assume \( p, q, r \) are distinct symbols from \( \text{sig}(\varphi) \) such that \( p \) and \( q \) are in \( \Sigma \) but \( r \) is not. Take
\[
\Sigma_{pq} = \Sigma, \quad \Sigma_{pr} = \{r\} \cup \Sigma \setminus \{q\}, \quad \Sigma_{qr} = \{r\} \cup \Sigma \setminus \{p\}.
\]
By the induction hypothesis, there must be \( \Omega_{pq}, \Omega_{pr}, \Omega_{qr} \in V \) with \( \Sigma_{pq} \subseteq \Omega_{pq}, \Sigma_{pr} \subseteq \Omega_{pr} \), and \( \Sigma_{qr} \subseteq \Omega_{qr} \). We may assume \( \Omega_{pq}, \Omega_{pr}, \Omega_{qr} \) are distinct, otherwise we are done. It is not hard to see that one of these vertices, say \( \Sigma_{pq} \), must be on the path between the other two. As \( \Sigma_{pr} \) and \( \Sigma_{qr} \) both contain \( r \), it follows that \( r \in \Sigma_{pq} \) as well, hence \( r \in \Omega_{pq} \). The other two cases are similar.

\[\text{Lemma 3.1.5.} \quad \text{Suppose} \ T = \langle V, E, \pi \rangle \text{ is a theory tree for} \ \Pi. \text{ Then} \ rng(\pi) = \Pi.\]

\[\text{Proof.} \]
\[\text{(\(\subseteq\)) Obvious.}\]
\[\text{(\(\supseteq\)) If} \ \varphi \in \Pi, \text{ then by Lemma 3.1.4, there is a} \ \Sigma \in V \text{ with} \ \text{sig}(\varphi) \subseteq \Sigma, \text{ so} \ \varphi \in \pi(\Sigma). \]

For theory trees which satisfy the running intersection property, there is a simpler way to formulate the notion of a proper labeling, as given in the next lemma.

\[\text{Lemma 3.1.6.} \quad \text{If} \ T = \langle V, E, \pi \rangle \text{ is a theory tree for} \ \Pi \text{ and we define an edge labeling} \ \sigma \text{ for} \ T \text{ by} \]
\[\sigma(x, y) := \text{sig}(\pi(x)) \cap \text{sig}(\pi(y))\]
\[\text{for all} \ \langle x, y \rangle \in E, \text{ then} \ \sigma \text{ is a proper labeling for} \ T.\]

\[\text{Proof.} \quad \text{Suppose} \ e = \langle x, y \rangle \in E \text{ and} \ X \ (\text{including} \ x) \text{ resp.} \ Y \text{ are the sets of vertices into which} \ e \text{ partitions} \ V, \text{ i.e. the respective sides of} \ e \text{ in} \ T. \text{ We need to show that} \]
\[\text{sig} \left( \bigcup_{v \in X} \pi(v) \right) \cap \text{sig} \left( \bigcup_{v \in Y} \pi(v) \right) \subseteq \sigma(x, y).\]

Assume \( p \in \text{sig}(\bigcup_{v \in X} \pi(v)) \) and \( p \in \text{sig}(\bigcup_{v \in Y} \pi(v)) \), for some symbol \( p \). Then there must be certain vertices \( w \in X \) and \( z \in Y \) such that \( p \in \text{sig}(\pi(w)) \) and \( p \in \text{sig}(\pi(z)) \). Obviously there is a (unique, undirected) path between \( w \) and \( z \) in \( T \) which passes through both \( x \) and \( y \). As \( \langle V, E \rangle \) is a tree decomposition, it satisfies property (3) of Definition 3.1.2. Therefore \( p \in \text{sig}(\pi(x)) \) and \( p \in \text{sig}(\pi(y)) \), hence \( p \in \text{sig}(\pi(x)) \cap \text{sig}(\pi(y)) = \sigma(x, y). \]

Henceforth, we are only interested in theory trees constructed as per Definition 3.1.3, so we may dispose of a separate edge labeling \( \sigma \) altogether. Instead, we simply copy only those formulas \( \varphi \) from a vertex \( x \) to its parent \( y \) for which \( \varphi \in \text{lang}(y) \). By Lemma 3.1.6 together with Theorem 2.3.7, this guarantees refutation-completeness for the reasoning algorithm.

This thesis does not go into how the tree decomposition algorithm itself operates. The tree decompositions for the experiments discussed later on (§ 6.1.1) were created using software made available by Amir et al. that was also used in their experiments [MMAU03, § 2.1].

### 3.2 Answering Queries

As computing a tree decomposition is usually relatively expensive, this approach to automated reasoning works best if we have a fixed theory \( \Pi \) which we want to answer many queries about. Fortunately, this is often the case in practice. We can then proceed as follows. First we create a theory tree \( T = \langle V, E, \pi \rangle \) for \( \Pi \). For each query \( \varphi \), we want to know whether \( \Pi \vdash \varphi \), which can be answered as follows:
(1) make a copy $T'$ of $T$,
(2) add $\neg \varphi$ to an appropriate vertex of $T'$, i.e. a vertex $v$ with $\text{sig}(\neg \varphi) \subseteq v$,
(3) direct $T'$ towards a suitable vertex, and
(4) test whether $T'$ is satisfiable.

Steps (2) and (3) require further explanation. For (3), the question is what vertices are “suitable”. A possibility Amir proposes in [Ami02, § 4.2.2], which was adopted for this thesis, is to direct $T'$ towards a vertex which is as far away from the query vertex, i.e. the one to which $\neg \varphi$ was added, as possible. This is discussed further in § 6.1.1.

As for (2): such a vertex may not exist. First, consider a simple situation in which a suitable query vertex does exist. Suppose we are dealing with the theory tree in Figure 3.3(ii), which is a tree decomposition of, say, $\Pi$. To find out whether $\Pi \vdash p$, we add $\neg p$ to the lower left vertex and direct the tree towards the lower right vertex. The result is depicted in Figure 3.4(i). One can easily see that the reasoning algorithm will deduce $\bot$ using this theory tree.

Now, assume we want to test whether $\varphi = p \rightarrow t$ is a consequence of $\Pi$. In that case, we select a vertex $v$ which shares as many symbols with $\neg \varphi = p \land \neg t$ as possible, and add $\neg \varphi$ to $\pi(v)$. This means we would add $p$ and $\neg t$ to the vertex containing $p \lor \neg q$, yielding the tree shown in Figure 3.4(ii). Unfortunately, the running intersection property no longer holds: the symbol $t$ should be present in the top vertex but it is not. To restore the property, we employ the following procedure.

Starting from the query vertex $v$, we “branch out” to every other vertex $w$, keeping track of the (unique) path from $v$ to $w$ as we go. Then if $w$ contains a symbol $p$ which is not in $v$, we simply add the symbol $p$ to every vertex $x$ on the path between $v$ and $w$. That is, we somehow attach the information to $x$ that $p$ is considered to be in the signature of $x$, which the reasoning algorithm will then take into account.

Provided there is a reasonably “fitting” vertex for $\varphi$, in most cases this restoration can be done in $O(n)$ time, where $n$ is the number of vertices. The resulting theory tree should still be suitable for efficient reasoning. In the experiments with $ALC$ theories discussed later (cf. Ch. 6), each query was (basically) of the form $p \rightarrow q$, which satisfies the above requirements. In fact, we could deal with such a query by adding $p$ to a suitable vertex (which is guaranteed to exist) and doing the same for $\neg q$.

To test merely whether a given theory is satisfiable as is, one would just direct the theory tree towards a random vertex. There may be a way to make a sensible choice as to which vertex that should be, but this is not considered in this thesis.
Adaptation to Semantic Tableaux

Our main motivation for using semantic tableaux is that they are widely used for modal logics, and almost exclusively for description logics.

4.1 Semantic Tableaux

First, we briefly review the method of semantic tableaux \cite{Bet55, Hin55, Smu68} in order to establish some notations, and to give a precise description of the basic tableau algorithm that we will modify later. For a more rigorous treatment, the reader is referred to the literature.

Semantic tableaux are a proof search method that works by systematically exploring all consequences of an assumption. To find out whether a formula $\phi$ is satisfiable, we (roughly speaking) use semantic tableaux to deconstruct $\phi$ into its constituent parts. A semantic tableau is usually represented as a rooted tree of formulas. The following informal description explains how semantic tableaux work for the case of propositional logic.

First of all, we assume that our initial assumption $\phi$ is expressed in negation normal form (NNF). This means that negations occur only directly in front of atomic formulas. For simplicity, suppose also that the only logical connectives present are $\neg$, $\land$, and $\lor$. (Using rules such as De Morgan’s, we can always translate a given formula into this form.) We start with a tree consisting of just a single vertex $\phi$, and then repeatedly apply the following transformation rules to the branches of $T$:

\[ \begin{align*}
\phi \land \psi & \quad \rightarrow \quad \phi \lor \psi \\
\phi \lor \psi & \quad \rightarrow \quad \phi \land \psi
\end{align*} \]

The rule for conjunctions states that if a branch contains $\phi \land \psi$, then we can extend it by adding vertices containing $\phi$ and $\psi$, unless already present in the branch. The rule for disjunctions states that if a branch contains $\phi \lor \psi$ but neither $\phi$ nor $\psi$, then we can split the branch into two subbranches, adding $\phi$ to the first and $\psi$ to the second.

A branch contains a clash whenever it contains two atomic formulas $p$ and $\neg p$. A branch is complete if no more transformation rules can be applied to it; similarly, a tableau is complete if all of its branches are. As soon as a branch contains a clash, we also consider it as complete. A tableau is closed if each of its branches contains a clash, and open otherwise.

In its simplest form, the tableau algorithm randomly applies transformation rules until the
tableau is complete. For example, below is a complete semantic tableau for \((p \lor q) \land (p \lor \neg q)\):

\[
(p \lor q) \land (p \lor \neg q)
\]

\[
\begin{array}{c}
p \lor q \\
p \lor \neg q \\
p \\
\neg q \\
\phi
\end{array}
\]

The only branch containing a clash has been marked with a \(\phi\) sign. All of the other branches represent countermodels to our original assumption, which has therefore been shown not to be a tautology. In practice, of course the algorithm would stop upon finding a countermodel, i.e. a complete non-clashing branch.

4.1.1 Specification of a typical tableau algorithm

The computer usually constructs a tableau depth-first, backtracking whenever the current branch leads to a clash. We will therefore specify our tableau algorithms as follows. The algorithm works with sets of formulas, i.e. theories, which are maintained in a stack \(S\). These theories represent the branches. Initially, \(S\) contains only our original assumption \(\Pi_0\) (recall that we may view a theory as a conjunction). We then repeatedly pop the top of \(S\), say \(\Pi\). If \(\Pi\) contains a clash, we backtrack by ignoring it and continuing to work on the rest of the stack. If \(\Pi\) is complete then we stop, having found a countermodel. Otherwise, we apply a transformation rule to \(\Pi\) and push the result(s) onto \(S\). This process continues until either we have found a model, or \(S\) becomes empty, in which case \(\Pi_0\) must be unsatisfiable.

There are two points at which the algorithm has a choice to make: (1) given a theory \(\Pi\), which \(\varphi \in \Pi\) to select for applying a transformation rule, and (2) when considering a disjunction \(\chi \lor \psi \in \Pi\), in what order to push \(\Pi \cup \{\chi\}\) and \(\Pi \cup \{\psi\}\) onto \(S\). (The latter corresponds to the question of what branch to explore first.) For now, we will assume these choices are made nondeterministically.

Figures 4.1 and 4.2 show a pseudocode specification of a typical tableau algorithm as just described. We will later modify this algorithm to operate on theory trees, and also adapt it to \(\mathcal{ALC}\) instead of propositional logic.

4.1.2 Soundness, termination, and completeness

Intuitively, it is easy to see that the tableau method is sound, terminating, and complete. For soundness, we have to show that every complete open branch in fact provides a countermodel. This can be done by defining a canonical interpretation corresponding to such a branch, and proving that this interpretation is indeed a countermodel. Termination, informally speaking, follows from the fact that there cannot be an infinite sequence of transformation rule applications starting from the initial tree (i.e. theory). Finally, completeness means that if the tableau algorithm cannot find a countermodel, then none exists. For detailed proofs of each of these assertions, the reader is referred to the literature.

Although the tableau algorithm is guaranteed to terminate when dealing with propositional logic, this claim does not hold for the first-order case. However, for the description logic \(\mathcal{ALC}\)
Procedure \texttt{Sat}(\Pi_0)

\textbf{Input}: \Pi_0 \text{ a theory in NNF.}

\textbf{Output}: \text{yes} if \Pi_0 \text{ is satisfiable, no if not.}

1. \texttt{S} ← the empty stack
2. \texttt{Push}(S, \Pi_0)
3. \textbf{while} \texttt{S} is nonempty \textbf{do}
   4. \texttt{Pi} ← \texttt{Pop}(S); (* i.e. remove the top of \texttt{S} and assign it to \texttt{Pi} *)
   5. \textbf{if} \texttt{Pi} \text{ does not contain a clash} \textbf{then}
   6. \texttt{Pi}' ← \{ \varphi \in \texttt{Pi} \mid \texttt{Transform}(S, \Pi, \varphi) \neq S \}
   7. \textbf{if} \texttt{Pi}' = \emptyset \textbf{then}
      (* no more transformation rules are applicable; we found a model *)
   8. \textbf{return} \text{yes}
   9. \textbf{else}
   10. \varphi ← \text{some element of} \texttt{Pi}'
   11. \texttt{S} ← \texttt{Transform}(S, \Pi, \varphi)
12. \textbf{end}
13. \textbf{return} \text{no}

Figure 4.1: A typical tableau algorithm.

Procedure \texttt{Transform}(S, \Pi, \varphi)

\textbf{Input}: S a stack, \Pi a theory in NNF, and \varphi \in \Pi.

\textbf{Output}: if \Pi can be transformed w.r.t. \varphi, then pushes the result(s) onto a copy of S and returns that; otherwise, returns S.

\texttt{S}' ← S

\textbf{switch} the structure of \varphi \textbf{do}

\textbf{case} \chi \land \varphi
   \textbf{if} \chi \notin \Pi \text{ or } \psi \notin \Pi \textbf{then} \texttt{Push}(S', \Pi \cup \{\chi, \psi\})

\textbf{case} \chi \lor \varphi
   \textbf{if} \chi \notin \Pi \text{ and } \psi \notin \Pi \textbf{then}
   \textbf{nondeterministically do}
   \texttt{Push}(S', \Pi \cup \{\psi\})
   \texttt{Push}(S', \Pi \cup \{\chi\})
   \textbf{or}
   \texttt{Push}(S', \Pi \cup \{\chi\})
   \texttt{Push}(S', \Pi \cup \{\psi\})
\textbf{end}

\textbf{return} \texttt{S}'

Figure 4.2: Transformation procedure for propositional logic.
considered later in this thesis (§ 5.2), we can regain termination by using subset blocking (§ 5.3.1).

4.1.3 Optimizations

As with resolution, the general question of how to reason efficiently is a difficult one. Below we discuss a few commonly used optimizations which attempt to address this issue. The reasoning method proposed in the next section (§ 4.2) of this thesis can be considered an optimization as well.

Prioritizing transformation rules

One of the more intuitive optimizations, at least for the case of propositional logic, is the prioritization of transformation rules. Most importantly, this typically means that we have the algorithm try to consider conjunctions before disjunctions. Informally speaking, this should be beneficial because disjunctions split up the tableau into several branches, in each of which we need to (separately) consider any disjunctions. By considering conjunctions first, this work can be avoided. Figure 4.3 illustrates what happens for the example theory \{ϕ₁ ∨ ··· ∨ ϕₙ, χ ∧ ψ\}.

In Figure 4.3(i) we transform the tableau according to the disjunction(s) first, and then treat the conjunctions, which requires \(n + n\) operations in total. As opposed to this, Figure 4.3(ii) shows that only \(n + 1\) operations are needed when preferring conjunctions over disjunctions.

Figure 4.3: Assigning priorities to transformation rules.

We can incorporate this optimization into the algorithm given in § 4.1.1 as follows. Suppose the following function \textit{priority} assigns an appropriate priority to a propositional logic formula (recall that we are only using \(\land, \lor, \text{and} \neg\)):

\[
priority(ϕ) := \begin{cases} 
2 & \text{if } ϕ = χ ∧ ψ \\
1 & \text{if } ϕ = χ ∨ ψ \\
0 & \text{otherwise} 
\end{cases}
\]

We could then replace line 10 of the algorithm in Figure 4.1 by the lines

\[
≲ ← \text{the total preorder on } Π' \text{ defined by: } ϕ ≲ ψ \text{ iff } priority(ϕ) ≤ priority(ψ) \\
ϕ ← \text{some } ≲\text{-maximal element of } Π'.
\]

Semantic branching

Normally, when the tableau algorithm encounters a disjunction \(ϕ ∨ ψ\), it splits up the tree into a branch containing \(ϕ\) and one containing \(ψ\). This can be called syntactic branching. By contrast, semantic branching splits up the tree into \(ϕ\) and \(¬ϕ ∧ ψ\).
Assuming the $\varphi$ branch is explored first, this is justified: if $\varphi$ (together with any assumptions already made) is satisfiable at all, we would never arrive at the $\neg\varphi \land \psi$ branch. Thus if we do, we may safely assert that $\varphi$ cannot be true.

Figure 4.4 shows the difference between both branching methods when applied to a theory $\{p \lor q, p \lor r, \ldots\}$. As depicted in Figure 4.4(i), suppose we start out with $p \lor q$, and the algorithm eventually (after a possibly large computation) finds out that $p$ cannot be satisfied. We then move to the $q$ branch and further split up the tree into a $p$ resp. $r$ branch w.r.t. the second formula. Obviously, we are bound to again find out that $p$ always leads to a clash — a fact that has not been memorized anywhere. On the other hand, Figure 4.4(ii) illustrates that had we added $\neg p$ to the $q$ branch, this re-computation would have been "short-circuited".

The M SOMS heuristic

The acronym M SOMS stands for Maximum number of Occurrences in disjunctions of Minimum Size. It is a heuristic concerning disjunctions: when dealing with a formula $\varphi \lor \psi$, the M SOMS heuristic helps decide which branch should be explored first. This heuristic is best explained by an example: see Figure 4.5.

In essence, if we are considering a disjunction $\varphi \lor \psi_1$ and there are also many other disjunctions $\varphi \lor \psi_2, \ldots, \varphi \lor \psi_n$, then we should explore $\varphi$ first. One would expect this to be a good
decision because $\varphi$ satisfies each of these other disjunctions, whereas $\psi_1$ satisfies only the first (assuming $\psi_1$ is distinct from $\psi_2, \ldots, \psi_n$).

**Other optimizations**

Another obvious optimization not yet mentioned is caching of intermediate results. Of course, one would need to carefully address the issue of what to cache and when. Note that the example in Figure 4.4 can be viewed as a situation in which caching would have been helpful.

In §§ 5.3.2 and 5.5, some optimizations for tableau reasoners are discussed further, and the question of whether they might be combined with the reasoning method proposed in this thesis is addressed.

### 4.2 Semantic Tableaux for Theory Trees

The approach we used for resolution, “copying” formulas between vertices, does not seem very suitable for semantic tableaux. When using resolution, we first transform the input theory into clausal form (or simply conjunctive normal form for propositional logic). The result is then regarded as a set of disjunctions, each of which is assumed to be true. We “distribute” such a set over a tree, an idea which is easy to grasp. By contrast, a semantic tableau should be regarded as a formula in disjunctive normal form: it is the disjunction of its branches, each branch being the conjunction of its vertices. We may try to think of each vertex in a theory tree as a separate tableau, but that would be awkward for a number of reasons. Most importantly, it would no longer be clear how and when a formula (representing a tableau) should be copied from one vertex to another.

Another problem is posed by the way in which quantifications (assuming we have those in our logic) are handled by tableau algorithms, as opposed to the method of resolution. To illustrate this, consider the following theory tree:

$$
\begin{align*}
A(c) \\
\neg C(c) \\
\quad A, C \\
A(x) & \rightarrow B(x) \\
B(x) & \rightarrow C(x)
\end{align*}
$$

When employing resolution, the subtheory in the bottom vertex would first be translated into $\{\neg A(x) \lor B(x), \neg B(x) \lor C(x)\}$. From this, the reasoning algorithm would then (using unification) infer $\neg A(x) \lor C(x)$, a formula which can be sent to the top vertex. Subsequently, the contradiction in the theory would be discovered.

On the other hand, the method of semantic tableau would not derive $A(x) \rightarrow C(x)$ locally. Instead, normally it would try instantiating $x$ with $c$ in the formulas of the bottom vertex, and then conclude $\neg A(c) \lor C(c)$. However, in our decomposition-based reasoning algorithm this would not be possible, as $c$ is not in the bottom vertex. Even more complicated problems will emerge if we have other universal or existential quantifiers in subformulas as well.

Below we will investigate how tree decompositions may still be used to try and speed up reasoning. As mentioned before, we construct theory trees in such a way that they satisfy the running intersection property (cf. Definition 3.1.2). Hence for any theory tree $T = (\mathcal{V}, \mathcal{E}, \pi)$, whenever $p \in \text{sig}(\pi(x))$ and $p \in \text{sig}(\pi(y))$, then $p \in \text{sig}(\pi(v))$ for all $v$ on the path between $x$ and $y$. However, this also means that for any $(x, y) \in \mathcal{E}$, if $p \in \text{sig}(\pi(x))$ but $p \notin \text{sig}(\pi(y))$, then
p \in \bigcup_{v \in X} \text{sig}(\pi(v))$ but $p \notin \bigcup_{v \in Y} \text{sig}(\pi(v))$, where $X$ resp. $Y$ are the sets of vertices into which $(x, y)$ partitions $T$. For instance, in the theory tree below, if $p \in \text{sig}(\Pi_5)$ but $p \notin \text{sig}(\Pi_2)$, then $p \notin \text{sig}(\Pi_i)$ for $i = 1, 2, 3, 4, 6, 7, 8$.

\begin{center}
\begin{tikzpicture}
  \node (T) at (0,0) {$T$};
  \node (P1) at (1,1) {$\Pi_1$};
  \node (P2) at (1,-1) {$\Pi_2$};
  \node (P3) at (3,1) {$\Pi_3$};
  \node (P4) at (1,-2) {$\Pi_4$};
  \node (P5) at (2,-2) {$\Pi_5$};
  \node (P6) at (3,-2) {$\Pi_6$};
  \node (P7) at (4,1) {$\Pi_7$};
  \node (P8) at (4,-1) {$\Pi_8$};
  \node (P9) at (1,-3) {$\Pi_9$};
  \node (P10) at (2,-3) {$\Pi_{10}$};

  \draw[->] (T) -- (P1);
  \draw[->] (T) -- (P2);
  \draw[->] (T) -- (P3);
  \draw[->] (P1) -- (P4);
  \draw[->] (P4) -- (P5);
  \draw[->] (P5) -- (P6);
  \draw[->] (P1) -- (P7);
  \draw[->] (P2) -- (P7);
  \draw[->] (P1) -- (P8);
  \draw[->] (P2) -- (P8);
  \draw[->] (P3) -- (P9);
  \draw[->] (P3) -- (P10);
\end{tikzpicture}
\end{center}

Based on this observation, we will use the following reasoning algorithm. Given a theory $\Pi$, we first construct a theory tree $T = (V, E, \pi)$ for $\Pi$. We then assign an index $\text{idx}(v)$ to each $v \in V$ such that

1. $\text{idx}(x) < \text{idx}(y)$ if $\text{height}(x) < \text{height}(y)$, and
2. $\text{idx}(x) = \text{idx}(y)$ iff $\text{parent}(x) = \text{parent}(y)$.

For instance, the theory tree above might be indexed as follows:

\begin{center}
\begin{tikzpicture}
  \node (T) at (0,0) {$T$};
  \node (P1) at (1,1) {$1 : \Pi_1$};
  \node (P2) at (1,-1) {$2 : \Pi_2$};
  \node (P3) at (3,1) {$2 : \Pi_3$};
  \node (P4) at (1,-2) {$3 : \Pi_4$};
  \node (P5) at (2,-2) {$3 : \Pi_5$};
  \node (P6) at (3,-2) {$3 : \Pi_6$};
  \node (P7) at (4,1) {$4 : \Pi_7$};
  \node (P8) at (4,-1) {$4 : \Pi_8$};
  \node (P9) at (1,-3) {$5 : \Pi_9$};
  \node (P10) at (2,-3) {$5 : \Pi_{10}$};

  \draw[->] (T) -- (P1);
  \draw[->] (T) -- (P2);
  \draw[->] (T) -- (P3);
  \draw[->] (P1) -- (P4);
  \draw[->] (P4) -- (P5);
  \draw[->] (P5) -- (P6);
  \draw[->] (P1) -- (P7);
  \draw[->] (P2) -- (P7);
  \draw[->] (P1) -- (P8);
  \draw[->] (P2) -- (P8);
  \draw[->] (P3) -- (P9);
  \draw[->] (P3) -- (P10);
\end{tikzpicture}
\end{center}

The tableau algorithm now works with theory trees rather than theories. We still (basically) use the ordinary transformation rules, but utilize the aforementioned properties of the theory tree to add two heuristics. For the case of propositional logic, the following informal description roughly illustrates how this is done; a more precise account of the algorithm will be given later.

Normally, the tableau algorithm randomly selects a formula $\varphi$ to which a transformation rule is to be applied (assuming we are not using any heuristics for this). Our algorithm, however, selects a $\varphi$ in a vertex $v$ such that $\text{idx}(v)$ is as high as possible. That is, we try to work upwards in the theory tree $T$. The formulas resulting from applying a transformation rule are added to $v$. For example, suppose $\chi \land \psi \in \pi(v)$, at least one of $\chi$ and $\psi$ is not in any vertex, and there is no $\varphi' \in \pi(w)$ with $\text{idx}(w) > \text{idx}(v)$ to which a transformation rule can be applied. Then we may transform $T$ into $T[v \leftarrow \{\chi, \psi\}]$.

The second heuristic concerns disjunctions. Normally, given $\chi \lor \psi \in \Pi$ with neither $\chi$ nor $\psi$ in $\Pi$, we would transform $\Pi$ into $\Pi' = \Pi \cup \{\chi\}$ and $\Pi'' = \Pi \cup \{\psi\}$, and continue the search nondeterministically with either $\Pi'$ or $\Pi''$ (eventually backtracking if needed). Our algorithm, however, tries to first choose that disjunct which is most likely to either (a) lead to a clash quickly, or (b) not lead to a clash at all. This is done as follows. Suppose $\chi \lor \psi \in \pi(x)$ and $y$ is \footnote{More generally, for convenience from now on we may write $T[v \leftarrow \{\varphi_1, \ldots, \varphi_n\}]$ instead of $T[v \leftarrow \varphi_1] \cdots [v \leftarrow \varphi_n]$.}
the parent of $x$ in $T$. We choose $\chi$ if $\text{sig}(\chi)$ shares less symbols with $\text{sig}(\pi(y))$ than $\text{sig}(\psi)$ shares with $\text{sig}(\pi(y))$; otherwise we choose $\psi$. Since any $p \not\in \text{sig}(\pi(v))$ can neither be in $\text{sig}(\pi(v))$ for any $v$ with $\text{idx}(v) < \text{idx}(y)$, the chances of discovering $\neg p$ later on should be small.

As a concrete example, suppose we are dealing with the theory

$$p \lor \neg q \quad s \lor q \quad \neg s \lor q \quad (r \land t) \lor s$$

for which the following (indexed) theory tree $T_0$ has been constructed:

```
1 : $(r \land t) \lor s$
   \uparrow
2 : $s \lor q$
   \uparrow
3 : $\neg s \lor q$
```

For convenience, call $v_1, v_2, v_3$ the vertices of $T_0$. There are four formulas to which a transformation rule can be applied, but as $v_3$ has the highest index, the algorithm starts by transforming $T_0$ into $T_1 = T_0[v_3 \leftarrow \neg q]$ and $T_2 = T_0[v_3 \leftarrow p]$. It then continues using $T_2$, because $|\{p\} \cap \{s, q\}| < |\{q\} \cap \{s, q\}|$. Next, it randomly chooses between $s \lor q$ and $\neg s \lor q$, since both have the same index. In either case (assume $\neg s \lor q$ is selected), it would continue using $T_2[v_2 \leftarrow q]$, which is depicted below:

```
1 : $(r \land t) \lor s$
   \uparrow
2 : $\neg s \lor q$
   \uparrow
3 : $p \lor \neg q$
```

Finally, the algorithm considers $(r \land t) \lor s$: as $v_1$ has no parent, a disjunct is randomly chosen, say $s$. At this point, a model has been found and we are done. This calculation corresponds to the following tableau:

```
\begin{array}{c}
(r \land t) \lor s \\
\quad s \lor q \\
\quad \neg s \lor q \\
\quad p \lor \neg q \\
\quad \neg q
\end{array}
```
Without these heuristics, we could have done far worse. For instance, we might have assumed \( \neg q \) early on:

\[
\begin{align*}
(r \land t) \lor s \\
 s \lor q \\
 \neg s \lor q \\
 p \lor \neg q \\
\end{align*}
\]

As one would expect, a theory tree \( T = \langle V, E, \pi \rangle \) contains a clash whenever there are certain vertices \( v \) and \( w \) with \( p \in \pi(v) \) and \( \neg p \in \pi(w) \); of course \( v \) need not be identical to \( w \).

4.2.1 Specification of the tableau algorithm for theory trees

As mentioned, when applying a transformation rule to a formula \( \varphi \) in a vertex \( v \), the resulting formulas “inherit” the index of \( v \). For the case of propositional logic, we can indicate this informally using the following straightforward transformation rules:

\[
\begin{align*}
\vdots \\
v : (\varphi \land \psi) & \quad \vdots \\
\vdots \\
v : (\varphi \lor \psi) & \\
\vdots \\
v : \varphi & \\
v : \psi & \\
\end{align*}
\]

A formal specification of the tableau algorithm for theory trees (in propositional logic) is given in Figures 4.6 and 4.7.

The question arises: which should be more important, the vertex indices or the priorities of the transformation rules (in other words, how should we integrate these two heuristics)? Preliminary tests suggested the latter. We will therefore impose an appropriate lexicographic ordering on the indices and priorities, by replacing line 11 in Figure 4.6 with

\[
\preceq \quad \text{the total preorder on } X \text{ defined by: } (x, \varphi) \preceq (y, \psi) \text{ iff either}
\]

1. \( \text{priority(} \varphi \text{)} < \text{priority(} \psi \text{)} \), or
2. \( \text{priority(} \varphi \text{)} = \text{priority(} \psi \text{)} \) and \( \text{idx}(x) \leq \text{idx}(y) \).
Procedure \textsc{Tree-Sat}(T_0)

\textbf{Input} : \( T_0 = \langle V, E, \pi_0 \rangle \) a theory tree for \( \Pi \) in NNF.
\textbf{Output} : yes if \( \Pi \) is satisfiable, no if not.

1. \( \text{idx} \leftarrow \) an arbitrary mapping from \( V \) to \( \mathbb{N} \) such that
   (i) \( \text{idx}(x) < \text{idx}(y) \) if \( \text{height}(x) < \text{height}(y) \), and
   (ii) \( \text{idx}(x) = \text{idx}(y) \) if \( \text{parent}(x) = \text{parent}(y) \)

2. \( S \leftarrow \) the empty stack
3. \( \text{Push}(S, T_0) \)
4. \textbf{while} \( S \) is nonempty \textbf{do}
5. \( \langle V, E, \pi \rangle \leftarrow \text{Pop}(S) \)
6. \textbf{if} \( \text{rng}(\pi) \) does not contain a clash \textbf{then}
7. \( X \leftarrow \{ \langle v, \varphi \rangle \mid \varphi \in \pi(v) \text{ and Tree-Transform}(S, T, \text{idx}, v, \varphi) \neq S \} \)
8. \textbf{if} \( X = \emptyset \) \textbf{then}
9. \textbf{return} yes
10. \textbf{else}
11. \( \preceq \leftarrow \) the total preorder on \( X \) defined by: \( \langle x, \varphi \rangle \preceq \langle y, \psi \rangle \) iff \( \text{idx}(x) \leq \text{idx}(y) \)
12. \( \langle v, \varphi \rangle \leftarrow \) some \( \preceq \)-maximal element of \( X \)
13. \( S \leftarrow \text{Tree-Transform}(S, T, \text{idx}, v, \varphi) \)
14. \textbf{end}
15. \textbf{return} no

Figure 4.6: Tableau algorithm for theory trees.

So for instance, if there is a conjunction in a vertex \( x \) and another one in \( y \), then we would prefer to apply a transformation rule to the one in \( y \) if \( \text{idx}(y) > \text{idx}(x) \). On the other hand, if there is a disjunction in \( y \) and a conjunction in \( x \), then we would prefer \( x \) no matter what their indices.

4.2.2 Miscellaneous considerations

The memory overhead is not nearly as bad as it may seem. The structure of the theory tree remains fixed during the course of the algorithm, so we can separate it into a “signature tree” and a vertex-labeled theory (i.e. a theory consisting of pairs \( \langle v, \varphi \rangle \) instead of only formulas). Such a point of view has already been considered on p. 11. The transformation rules then simply operate on these vertex-labeled formulas, consulting the signature tree where necessary. This is in fact how things were implemented.

Soundness, termination, and completeness are all obvious, assuming that the “underlying” tableau algorithm satisfies these requirements. This method is merely a heuristic and does not actually prune the search tree, although there may be a way to use theory trees for this.
Procedure Tree-Transform\((S, T, idx, v, \varphi)\)

**Input**: \(S\) a stack, \(T = (V, E, \pi)\) a theory tree in NNF, \(idx : V \to \mathbb{N}\), and \(\varphi \in \pi(v)\).

**Output**: if \(T\) can be transformed w.r.t. \(\varphi\), then pushes the result(s) onto a copy of \(S\) and returns that; otherwise, returns \(S\).

\[
S' \leftarrow S
\]

switch the structure of \(\varphi\) do

\[\text{case } \chi \land \psi\]

if \(\chi \not\in \text{rng}(\pi)\) or \(\psi \not\in \text{rng}(\pi)\) then

\[\text{Push}(S', T[v \leftarrow \{\chi, \psi}\])\]

\[\text{case } \chi \lor \psi\]

if \(\chi \not\in \text{rng}(\pi)\) and \(\psi \not\in \text{rng}(\pi)\) then

\[
\begin{align*}
&w \leftarrow v's \text{ parent, or } v \text{ itself if none} \\
&\chi\text{-dist} \leftarrow |\text{sig}(\chi) \cap w| \\
&\psi\text{-dist} \leftarrow |\text{sig}(\psi) \cap w|
\end{align*}
\]

if \(\chi\text{-dist} < \psi\text{-dist}\) then

\[\text{Push}(S', T[v \leftarrow \psi])\]

\[\text{Push}(S', T[v \leftarrow \chi])\]

else if \(\psi\text{-dist} < \chi\text{-dist}\) then

\[\text{Push}(S', T[v \leftarrow \chi])\]

\[\text{Push}(S', T[v \leftarrow \psi])\]

else

nondeterministically do

\[\text{Push}(S', T[v \leftarrow \psi])\]

\[\text{Push}(S', T[v \leftarrow \chi])\]

or

\[\text{Push}(S', T[v \leftarrow \chi])\]

\[\text{Push}(S', T[v \leftarrow \psi])\]

end

return \(S'\)

Figure 4.7: Transformation procedure for propositional logic theory trees.
Chapter 5

Application to Description Logics

5.1 Description Logics

Description logics (DLs) are a family of logics which, nowadays, are generally regarded as variants of first-order logic. The logic considered in this thesis, $\mathcal{ALC}$, is one of the most basic description logics, being a rather simple fragment of first-order logic. Other, more complicated description logics contain additional operators, some of which cannot be translated into FOL.

The main motivation for having such a variety of description logics is increased expressiveness on the one hand, and increased efficiency of reasoning on the other. In the last few decades, much research has been devoted to analyzing both of these properties for each description logic. Recently, description logics have been used as the basis for so-called web ontology languages as part of W3C’s Semantic Web initiative\(^1\).

For simplicity, this thesis limits itself to $\mathcal{ALC}$, which is introduced in the next section. After that, we will look at semantic tableaux for $\mathcal{ALC}$ theories as normally used, and subsequently modify them to incorporate the decomposition-based heuristics outlined in the previous chapter.

5.2 The Logic $\mathcal{ALC}$

Definition 5.2.1 (Syntax of $\mathcal{ALC}$). Assume that we are given a denumerable set of atomic concepts, atomic roles, and individuals. Atomic concepts will be denoted by $A$, $B$, atomic roles by $R$, $S$, and individuals by $a$, $b$, $c$.

(i) The set of $\mathcal{ALC}$ concepts is defined inductively as follows:

- Every atomic concept is a concept.
- If $C$ and $D$ are concepts, then the following expressions are also concepts:
  - $\top$ (the universal concept), $\bot$ (the bottom concept),
  - $\neg C$ (negation), $C \cap D$ (intersection), $C \cup D$ (union),
  - $\forall R.C$ (universal quantification), $\exists R.C$ (existential quantification).

(ii) The set of $\mathcal{ALC}$ formulas consists of

- concept inclusion axioms of the form $C \subseteq D$,
- concept equalities $C = E$,
- concept assertions $C(a)$, and

\(^1\text{http://www.w3.org/2001/sw/}\)
The interpretation function extends inductively to complex concepts as follows:

- role assertions $R(a, b)$.

More precisely, concept inclusions whose left-hand side (LHS) is atomic (i.e. $A \sqsubseteq C$) will be called primitive concept inclusions, whereas those whose LHS is complex are called general concept inclusions (GCIs).

As one before, formulas will be denoted by $\alpha, \beta, \gamma, \varphi, \chi, \psi$.

As one would expect, by an $\mathcal{ALC}$ theory we mean a set of $\mathcal{ALC}$ formulas. By contrast, in DL literature it is customary to distinguish between so-called TBoxes and ABoxes: a TBox is a set of concept inclusions, whereas an ABox contains concept and role assertions. (Sometimes one even separates out the role assertions, adding an RBox.) The combination of a TBox and an ABox is often called a knowledge base (KB), a term we also sometimes employ instead of the word “theory”. The name “ontology” is applied occasionally as well, although it can be argued that only the TBox part of a KB is an ontology.

**Definition 5.2.2 (Semantics of $\mathcal{ALC}$).** An $\mathcal{ALC}$ interpretation $I$ is a nonempty set $\Delta^I$ together with an interpretation function $\cdot^I$ which assigns to each atomic concept $A$ a set $A^I \subseteq \Delta^I$, to each atomic role $R$ a binary relation $R^I \subseteq \Delta^I \times \Delta^I$, and to each individual $a$ an element $a^I \in \Delta^I$.

1. The interpretation function extends inductively to complex concepts as follows:
   
   $\top^I = \Delta^I$
   
   $\bot^I = \emptyset$
   
   $(\neg C)^I = \Delta^I \setminus C^I$
   
   $(C \cap D)^I = C^I \cap D^I$
   
   $(C \cup D)^I = C^I \cup D^I$
   
   $(\forall R.C)^I = \{ a \in \Delta^I | \forall b \ (\langle a, b \rangle \in R^I \rightarrow b \in C^I) \}$
   
   $(\exists R.C)^I = \{ a \in \Delta^I | \exists b \ (\langle a, b \rangle \in R^I \land b \in C^I) \}$.

2. We write $I \models \varphi$ if the interpretation $I$ satisfies the formula $\varphi$, which is defined as follows:

   - $I \models C \sqsubseteq D$ iff $C^I \subseteq D^I$; we also say that $C$ is subsumed by $D$ w.r.t. $I$,
   - $I \models C \equiv D$ iff $C^I = D^I$,
   - $I \models C(a)$ iff $a^I \in C^I$,
   - $I \models R(a, b)$ iff $\langle a^I, b^I \rangle \in R^I$.

3. As usual, an interpretation $I$ is a model of a theory $\Pi$ if $I$ satisfies every formula in $\Pi$. Furthermore, it is customary to write $\Pi \models \varphi$ rather than $\Pi \models \{ \varphi \}$.

**5.2.1 Relation to other logics**

$\mathcal{ALC}$ is a fragment of first-order logic; roughly speaking, atomic concepts resp. roles can be regarded resp. as unary and binary predicates, and individuals as constants. A complete translation from $\mathcal{ALC}$ into first-order logic is given in [BCM+03, § 4.2].

The relation between $\mathcal{ALC}$ and the multimodal logic $\mathcal{K}_n$ is even closer [Sch91]: $\mathcal{ALC}$ without concept and role assertions can be seen as a notational variant of $\mathcal{K}_n$. In particular, $\mathcal{ALC}$’s $\forall$ and $\exists$ operators correspond to the modal operators $\square$ resp. $\Diamond$. Since concept and role assertions refer specifically to individuals, for those we need to go beyond standard modal logics. **Hybrid logics** extend modal logics with so-called nominals, which correspond to individuals as employed in description logics. For more on the connection between $\mathcal{ALC}$ and $\mathcal{K}_n$, see [BCM+03, § 4.2.2].
5.3 Semantic Tableaux for \(\mathcal{ALC}\) Theories

The tableau algorithm for \(\mathcal{ALC}\) is similar to the one for propositional logic treated in § 4.1. We will use the basic procedure discussed in [BS01, §3], but work with theories instead of ABoxes. It should be noted that most actual implementations of DL tableau algorithms represent the model which they attempt to construct as a certain type of tree. Without going into detail, in such a tree, each vertex is labeled with a set of concepts and each edge with a role. For simplicity, in (the implementation for) this thesis we use theories, and later theory trees, rather than such trees.

Our goal is to show satisfiability of an \(\mathcal{ALC}\) theory \(\Pi\). First of all, we translate \(\Pi\) into NNF. This is done by translating each \(\varphi \in \Pi\) to its NNF equivalent \(\text{nnf}(\varphi)\), defined inductively as follows:

\[
\begin{align*}
\text{nnf}(\neg \top) &= \bot \\
\text{nnf}(\neg \bot) &= \top \\
\text{nnf}(C \sqcap D) &= \text{nnf}(C) \sqcap \text{nnf}(D) \\
\text{nnf}(C \sqcup D) &= \text{nnf}(C) \sqcup \text{nnf}(D) \\
\text{nnf}(\forall R.C) &= \forall R.\text{nnf}(C) \\
\text{nnf}(\exists R.C) &= \exists R.\text{nnf}(C) \\
\text{nnf}(\neg C) &= \text{nnf}(C) \\
\text{nnf}(\neg \neg C) &= \text{nnf}(C) \\
\text{nnf}(R(a,b)) &= R(a,b) \\
\text{nnf}(C \models D) &= \text{nnf}(C) \sqsubseteq \text{nnf}(D) \\
\text{nnf}(\neg A) &= \neg \text{nnf}(A) \\
\end{align*}
\]

Observe that a concept inclusion \(C \sqsubseteq D\) may be read as “\((\neg C \sqcup D)\) for all individuals \(a\)”.

Moreover, we can view an equality \(C \models D\) as a pair of inclusions \(C \sqsubseteq D\) and \(D \sqsubseteq C\). From these observations, and the fact that roles cannot be negated, it follows that there is a clash only if we have both \(A(a)\) and \(\neg A(a)\), for some atomic concept \(A\) and individual \(a\). One should keep in mind that \(\top\) and \(\bot\) are not considered atomic concepts; alternatively, they could have been defined as abbreviations for \(A \sqcup \neg A\) resp. \(A \sqcap \neg A\), for an arbitrary atomic concept \(A\).

Figure 5.1 presents the tableau transformation rules for \(\mathcal{ALC}\), as was done for propositional logic on page 17. The first two rules, for intersection and union, are straightforward. The rules for universal and existential quantification are also easy to understand, referring to the semantics of \(\forall R.C\) and \(\exists R.C\).

The last four rules merit additional explanation. First, note that there is no rule for \(C \models D\) (where \(C\) is complex): this is because any such statements will initially be replaced by \(C \sqsubseteq D\) and \(D \sqsubseteq C\). Second, there are separate rules for \(A \sqsubseteq C\) and \(C \sqsubseteq D\): this is because inclusions with an atomic LHS may be treated “lazily”. That is, we apply \(A \sqsubseteq C\) only to those individuals \(a\) for which \(A(a)\) is already known — adding \(C(a)\) to the theory (i.e. possible model) in that case. We treat concept equalities with an atomic LHS lazily in a similar fashion, as expressed by the final two tableau rules.

In practice, this kind of laziness can greatly improve the reasoning performance. This is why, when authoring an actual theory, one would try to limit the use of GCIs as much as possible. GCIs get turned into unions, each of which essentially doubles the search space.

As Figure 4.2 did for propositional logic, Figure 5.2 lists the transformation procedure for \(\mathcal{ALC}\). The basic tableau algorithm which calls this procedure is identical to the one in Figure 4.1, except that the input theory is in \(\mathcal{ALC}\).

We will not go into soundness, termination, and completeness here. Soundness and completeness can be proved in essentially the same way as for propositional logic. Termination is elaborated on in the next subsection.
Figure 5.1: Transformation rules for $\mathcal{ALC}$. 
Procedure $\text{ALC}$-Transform$(S, \Pi, \varphi)$

Input: $S$ a stack, $\Pi$ a theory in NNF, and $\varphi \in \Pi$.

Output: if $\Pi$ can be transformed w.r.t. $\varphi$, then pushes the result(s) onto a copy of $S$ and returns that; otherwise, returns $S$.

$S' \leftarrow S$

switch the structure of $\varphi$ do

- case $(\lnot C \lor D)(a)$
  - if $C(a) \notin \Pi$ or $D(a) \notin \Pi$ then $\text{Push}(S', \Pi \cup \{C(a), D(a)\})$

- case $(\lnot C \land D)(a)$
  - if $C(a) \notin \Pi$ and $D(a) \notin \Pi$ then
    - nondeterministically do
      - $\text{Push}(S', \Pi \cup \{D(a)\})$
      - $\text{Push}(S', \Pi \cup \{C(a)\})$
    - or
      - $\text{Push}(S', \Pi \cup \{C(a)\})$
      - $\text{Push}(S', \Pi \cup \{D(a)\})$

- case $(\forall R.C)(a)$
  - if there is a $b$ with $R(a,b) \in \Pi$ and $C(b) \notin \Pi$ then $\text{Push}(S', \Pi \cup \{C(b)\})$

- case $(\exists R.C)(a)$
  - if there is no $c$ with $R(a,c) \in \Pi$ and $C(c) \in \Pi$ then
    - $\text{Push}(S', \Pi \cup \{R(a,b), C(b)\})$, where $b$ is a fresh individual

- case $A \subseteq C$
  - if there is an $a$ with $A(a) \in \Pi$ and $C(a) \notin \Pi$ then $\text{Push}(S', \Pi \cup \{C(a)\})$

- case $C \subseteq D$
  - if there is an individual $a$ in $\Pi$ with $(\text{nff}(\lnot C) \cup D)(a) \notin \Pi$ then $\text{Push}(S', \Pi \cup \{(\text{nff}(\lnot C) \cup D)(a)\})$

- case $A = C$
  - if there is an $a$ with either
    - (i) $A(a) \in \Pi$ and $C(a) \notin \Pi$, or
    - (ii) $\lnot A(a) \in \Pi$ and $\text{nff}(\lnot C)(a) \notin \Pi$
  - then
    - if (i) then $\text{Push}(S', \Pi \cup \{C(a)\})$ else $\text{Push}(S', \Pi \cup \{\text{nff}(\lnot C)(a)\})$

end

return $S'$

Figure 5.2: Transformation procedure for $\text{ALC}$.  

32
5.3.1 Subset blocking

In its present form, the algorithm need not terminate. This is caused by the availability of both existential quantifications and GCI. To illustrate why, suppose our theory contains the statement

\[ \top \sqsubseteq \exists R.A \]

and there is a some individual \( c_0 \). Since \( \top \) for any individual \( a \), the algorithm is forced to add \( R(c_0, c_1) \) and \( A(c_1) \), where \( c_1 \) is a fresh individual. But then, similarly, it needs to add \( R(c_1, c_2) \) and \( A(c_2) \); this process goes on infinitely. To regain termination we can use subset blocking, a method which, informally speaking, constructs cyclical models in pathological cases such as the one above. We will not discuss subset blocking any further (cf. [Hor97, § 3.3.2] or [BS01, § 5.2]), but merely mention that it was incorporated in the tableau reasoner implementation for this thesis.

5.3.2 Optimizations

The reasoning algorithms (and reasoning problems) for most DLs have a high complexity, but thanks to many optimizations, they turn out to be usable in practice nevertheless. Among these optimizations are the ones mentioned in § 4.1.3, but there are many more. See also Horrocks’ PhD thesis [Hor97], [BCM+03, Ch. 9], and [Hor02, Hor98].

The only optimization added to the implementation for this thesis is the prioritization of transformation rules. (That is, assuming we do not count lazy treatment of primitive concept inclusions/equalities and subset blocking as optimizations.) Basically, we can divide our transformation into nondeterministic vs. deterministic rules (e.g. unions are nondeterministic) and generating vs. nongenerating rules (the only generating rule being the one for existential quantifications). As explained for propositional logic in § 4.1.3, we are trying to apply transformation rules in an “efficient” order. In the implementation for this thesis, the order suggested in [TH05] is followed. This amounts to the following priorities for the \( \mathcal{ALC} \) transformation rules:

\[
\text{priority}(\varphi) := \begin{cases} 
7 & \text{if } \varphi = (\forall R.C)(a) \\
6 & \text{if } \varphi = (C \sqcap D)(a) \\
5 & \text{if } \varphi = A \sqsubseteq C \\
4 & \text{if } \varphi = A \sqsupseteq C \\
3 & \text{if } \varphi = (C \sqcup D)(a) \\
2 & \text{if } \varphi = C \sqsubseteq D \\
1 & \text{if } \varphi = (\exists R.C)(a) \\
0 & \text{otherwise}.
\end{cases}
\]

5.4 Semantic Tableaux for \( \mathcal{ALC} \) Theory Trees

For our purposes, the signature of an \( \mathcal{ALC} \) formula is taken to be the set of atomic concepts occurring in the formula, for these are the only “entities” that can produce a clash. Hence, atomic concepts will guide the creation of a tree decomposition. We then use the transformation rules expounded in Figure 5.3; these are the decomposition-based versions of the transformation rules in Figure 5.1. Similarly, Figure 5.4 lists the decomposition-based version of the \( \mathcal{ALC} \) transformation procedure given in Figure 5.2.
Figure 5.3: Theory tree transformation rules for ALC.
Procedure $\mathcal{ALC}$-Tree-Transform($S, T, idx, v, \varphi$)

**Input**: $S$ a stack, $T = (V, E, \pi)$ a theory tree in NNF, $idx : V \rightarrow \mathbb{N}$, and $\varphi \in \pi(v)$.

**Output**: if $T$ can be transformed w.r.t. $\varphi$, then pushes the result(s) onto a copy of $S$ and returns that; otherwise, returns $S$.

$S' \leftarrow S$

$I \leftarrow$ the set of all individuals in $\text{rng}(\pi)$

$\preceq \leftarrow$ the total preorder on $V \times I$ defined by: $(x, a) \preceq (y, b)$ iff $idx(x) \leq idx(y)$

**switch** the structure of $\varphi$ do

- **case** $(C \cap D)(a)$
  - if $C(a) \notin \text{rng}(\pi)$ and $D(a) \notin \text{rng}(\pi)$ then Push($S', T[v \leftarrow \{C(a), D(a)\}]$)
  - else if $C(a) \notin \text{rng}(\pi)$ and $D(a) \notin \text{rng}(\pi)$ then
    - $w \leftarrow v$’s parent, or $v$ itself if none
    - $C\text{-dist} \leftarrow |\text{atomic-concepts}(C) \cap w|$
    - $D\text{-dist} \leftarrow |\text{atomic-concepts}(D) \cap w|$
    - if $C\text{-dist} < D\text{-dist}$ then Push($S', T[v \leftarrow D(a)]$); Push($S', T[v \leftarrow C(a)]$)
  - else if $D\text{-dist} < C\text{-dist}$ then Push($S', T[v \leftarrow C(a)]$); Push($S', T[v \leftarrow D(a)]$)
  - else nondeterministically do one of the above

- **case** $(\forall R.C)(a)$

  - $X \leftarrow \{(v, b) \mid R(a, b) \in \pi(v) \text{ and } C(b) \notin \text{rng}(\pi)\}$
  - if $X = \emptyset$ then
    - Push($S', T[v \leftarrow \{C(b)\}]$)

- **case** $(\exists R.C)(a)$

  - if there is no $c$ with $R(a, c) \in \text{rng}(\pi)$ and $C(c) \in \text{rng}(\pi)$ then
    - Push($S', T[v \leftarrow \{R(a, b), C(b)\}]$, where $b$ is a fresh individual)

- **case** $A \sqsubseteq C$

  - $X \leftarrow \{(v, a) \mid A(a) \in \pi(v) \text{ and } C(a) \notin \text{rng}(\pi)\}$
  - if $X = \emptyset$ then
    - Push($S', T[v \leftarrow C(a)]$)

- **case** $C \sqsubseteq D$

  - $X \leftarrow \{(v, a) \mid a \in \text{sig}(\pi(v)) \text{ and } (\text{nnf}(\neg C) \cup D)(a) \notin \text{rng}(\pi)\}$
  - if $X = \emptyset$ then
    - Push($S', T[v \leftarrow (\text{nnf}(\neg C) \cup D)(a)]$)

- **case** $A = C$

  - $X \leftarrow \{(v, a) \mid A(a) \in \pi(v) \text{ and } C(a) \notin \text{rng}(\pi)\}$
  - $Y \leftarrow \{(v, a) \mid \neg A(a) \in \pi(v) \text{ and } \text{nnf}(\neg C)(a) \notin \text{rng}(\pi)\}$
  - if $X \cup Y = \emptyset$ then
    - if $(w, a) \in X$ then
      - Push($S', T[v \leftarrow C(a)]$)
    - else
      - Push($S', T[v \leftarrow \text{nnf}(\neg C)(a)]$)
  - end

end

return $S'$

Figure 5.4: Transformation procedure for $\mathcal{ALC}$ theory trees.
5.5 Combination with Existing Optimizations

The results reported above are only a very rough indication of the performance improvements that might be achieved in practice using this method, as most existing DL reasoner optimizations were not implemented. In this section, we argue that many of these optimization techniques may be combined with the theory tree heuristics proposed in this thesis. The discussion below considers each of the techniques covered in [BCM+03, § 9.5].

Preprocessing optimizations

So-called preprocessing optimizations attempt to normalize and simplify a given KB before we even start reasoning. Many of these optimizations fix “mistakes” in human-written KBs, e.g. by removing redundancy (rewriting, say, $C \sqcap D \sqcap C$ to $C \sqcap D$). Others are used to rewrite GCI-s into primitive concept inclusions for efficiency: in general, it would be a good thing to rewrite each GCI $A \sqcap C \sqsubseteq D$ to the primitive concept inclusion $A \sqsubseteq \neg C \sqcup D$.

Obviously, each of these optimizations would still be applicable if we intend to use the decomposition-based algorithm.

Optimizing classification

The task of “classifying” a KB amounts to computing all subsumptions between atomic concepts used in the KB, as will be explained in § 6.1.3. We can optimize this task by taking advantage of the fact that the subsumption relation is reflexive and transitive. It is also clear that whenever the KB contains a formula such as $A \sqsubseteq B_1 \sqcap \cdots \sqcap B_n$, then we know immediately that $B_1, \ldots, B_n$ are all subsumers of $A$.

These kinds of optimizations are also independent of the reasoning algorithm we use.

Optimizing subsumption testing

When using a tableau algorithm, subsumption testing boils down to testing the satisfiability of a concept. To prevent costly satisfiability (sub)tests from being run multiple times, we can use caching. For common-sense KBs, huge performance gains can be achieved by using caching.

As caching can obviously be used by both the plain and the decomposition-based algorithm, it is hard to foresee for which one it would make more of a difference. This is probably best verified experimentally.

Optimizing satisfiability testing

One optimization used to optimize satisfiability testing, semantic branching, has already been mentioned in § 4.1.3. Although semantic branching has been found to be most effective with randomly generated KBs, adding it should be beneficial for both the plain and decomposition-based algorithm.

Another optimization in this category, called local simplification, appears not very effective for common-sense KBs. We will not discuss it any further here (cf. [BCM+03, § 9.5.4.2]).

Finally, some other commonly used techniques for optimizing satisfiability testing are dependency directed backtracking and backjumping. A detailed explanation of these is beyond the scope of this thesis, but both seem compatible with the decomposition-based algorithm.

Heuristic guided search

As the theory tree algorithm itself is basically a combination of two heuristics, it is hard to argue theoretically whether (and how) it can be combined with heuristics such as MOMS (cf. § 4.1.3).
Figuring out (theoretically as well as experimentally) how existing heuristics can be integrated best into the decomposition-based algorithm would be one of the more interesting tasks for future work.

Caching the satisfiability status

As treated in [BCM+03, § 9.5.4.5], caching the satisfiability status is an optimization which appears most effective for artificial problems having a repetitive structure. It is not clear yet how this technique might be combined with the decomposition-based algorithm.
Chapter 6

Experimental Setup and Results

6.1 Setup

The basic setup for the experiments done for this thesis is as follows. Two kinds of $\mathcal{ALC}$ reasoners were implemented; one using the decomposition-based algorithm proposed in this thesis (cf. Figure 5.4), and one using a plain $\mathcal{ALC}$ tableau algorithm (cf. Figure 5.2). A number of $\mathcal{ALC}$ theories were then collected, and a number of queries were run using both algorithms to compare their performance.

6.1.1 Implementation

Naturally, it would have been much better to modify an existing DL reasoner (such as FaCT++, Pellet, or RacerPro) than to write one from scratch. The main reason for not doing this is that there was not enough time to study and understand the internals of one of these reasoners well enough for this.

The only optimization added to the reasoners is the prioritization of transformation rules (assuming we do not count subset blocking and lazy treatment of primitive concept inclusions/equalities as optimizations). Therefore, the results reported in § 6.2 are not at all conclusive and should be taken with a large grain of salt.

The implementation itself was written in OCaml. Since Amir has been kind enough to make his $\text{sp}$ utility for constructing tree decompositions available, that program was called by the software for this thesis. The $\text{ocamlgraph}$ library was used to represent theory trees, and the example theory trees shown in Appendix A were rendered using $\text{dot2tex}$ and L$\LaTeX$.

There are two details which have not been mentioned yet. First, contrary to Definition 3.1.3, formulas from an input theory are never assigned to multiple vertices when creating a tree decomposition. If a formula is eligible for (i.e. its signature “fits into”) multiple vertices, one is randomly chosen. Note that this differs from what is reported in [MMAU03].

Second, as announced in § 3.2, after adding a (negated) query $\phi$ to a vertex $v$, we direct the theory tree towards a vertex which is as far away from $v$ as possible. This is analogous to backwards reasoning such as used by many theorem provers. After directing the tree, $v$ should be located towards the bottom, and we should arrive at $v$ soon when we start reasoning. If, as one might expect, $\phi$ combined with the other formulas in $v$ (which can be considered “related” to $v$'s) already yield a contradiction, this will be discovered quickly.

---

1 Documentation and source code for the implementation can be viewed at http://hreker.com/msc-thesis/.
2 http://www-formal.stanford.edu/eyal/decomp/
6.1.2 Knowledge bases

Preliminary tests indicated that for KBs without any GCIs, there was usually little improvement (if any) in efficiency using the decomposition-based algorithm. Apparently, for GCI-less KBs the tableau algorithm never had many choices to make, as primitive concept inclusions $A \sqsubseteq C$ are treated lazily. Whenever it did encounter disjunctions $(C \sqcup D)(a)$, they were explicitly applied to the individual $a$, instead of being universally quantified as GCIs are. For this reason, only KBs containing a fair number of GCIs were selected for testing — however, having too many GCIs was also impractical because each one essentially doubles the search space. As a result, only a small number of KBs was deemed suitable for experimentation. Below is an overview of these KBs:3.

- **Name:** amino-acid  
  **Origin:** http://www.co-ode.org/ontologies/amino-acid/2005/10/11/amino-acid.owl  
  **Number of formulas:** 308  
  **Description:** A small ontology of amino acids and their properties.

- **Name:** wines  
  **Origin:** http://www.mindswap.org/2004/owl/wine/factoredWines.owl  
  **Number of formulas:** 44  
  **Description:** A fragment of the wines KB used in many DL tutorials. Actually, a manually randomly selected part of the factoredWines KB was used, for factoredWines itself turned out to be still too big.

- **Name:** miniTambis  
  **Origin:** http://www.mindswap.org/2005/debugging/ontologies/miniTambis.owl  
  **Number of formulas:** 177  
  **Description:** Simplified version of a biological science ontology developed by the TAMBIS (Transparent Access to Multiple Bioinformatics Information Sources) project.

- **Name:** people+pets  
  **Origin:** http://owl.man.ac.uk/tutorial/people+pets.rdf  
  **Number of formulas:** 94  
  **Description:** From a tutorial on OWL given by Ian Horrocks and Sean Bechhofer.

- **Name:** university  
  **Origin:** http://www.mindswap.org/2005/debugging/ontologies/University.owl  
  **Number of formulas:** 53  
  **Description:** Another example KB taken from the Mindswap website.

Other KBs that seemed suitable, were it not for their large size, are the pizza ontology (another commonly used example KB), the GALEN KB (a medical ontology), and the full version of the wines ontology.

Each ontology was first converted to the input format used by FaCT++ using the OWL Converter, then some further modifications were made to limit them to $ALC$ if needed. For instance, most of these ontologies were written in a more expressive description logic than $ALC$, which was dealt with by “flattening” unsupported subexpressions, i.e. replacing them by $\top$. It is

3 All of these KBs were retrieved on June 18, 2007.

4 [http://www.mygrid.org.uk/OWL/Converter](http://www.mygrid.org.uk/OWL/Converter)
assumed that the KBs can still be considered realistic enough examples after this (even though this flattening resulted in statements such as Quadruped ⊑ Animal ⊓ ∃ has_leg, T, which in its original form insisted that quadrupeds have exactly four legs — cf. Figure A.3). After this, a few logical simplifications were made, such as replacing all occurrences of T ⊓ C by C and T ⊔ C by T.

Figures A.1 through A.6 in Appendix A show excerpts of the theory trees that were created for each of these KBs.

6.1.3 Tests

A concept C is subsumed by D w.r.t. an ALC theory Π if Π |= C ⊑ D. In practice, given a KB Π, it is often useful to obtain an overview of what atomic concepts in Π subsume each other. Hence, a standard task for DL reasoners is to compute the “subsumption partial order” for Π, which is the relation ≤ defined by

\[ A ≤ B \quad \text{iff} \quad A, B ∈ \text{sig}(Π) \text{ and } Π |= A ⊑ B. \]

To compare the performance of the decomposition-based reasoning algorithm vs. the plain one, a random selection of concept subsumptions A ⊑ B was input as queries to both algorithms. To compute the answer to such a query, i.e. to find out whether Π |= A ⊑ B, we test whether Π ∪ \{(A ⊓ ¬B)(c_0)\} is satisfiable, with c_0 a fresh individual.

It should be noted that no optimizations specific to computing the subsumption partial order were employed, not even obvious ones like deducing A₁ ⊑ A₃ from A₁ ⊑ A₂ and A₂ ⊑ A₃. Another caveat is that we may get “overlap” if some concept C is found to be unsatisfiable w.r.t. Π (that is, Π |= ⊥ ⊑ C), because then C ⊑ D may be unnecessarily computed multiple times for different concepts D. Some sort of caching would be called for to prevent this.

Each query was run 10 times using both algorithms. Instead of recording the actual CPU time taken to answer a query, we count the number of transformation steps taken. This seemed a better measure for comparison, and is also the method used in [MMAU03]. Also, as in that publication, the cost of computing the initial tree decomposition was not taken into account, and neither was (slightly) modifying it after adding a query.

To prevent both algorithms from taking unacceptably long to answer a query, timeouts were used: the maximum number of transformation steps was limited to 2000 or 3000, depending on the KB in question.

6.2 Results

In most cases, the decomposition-based heuristics clearly did improve performance. The only exception to this was the amino-acid KB. As Figure A.2 shows, this was apparently due to the fact that its decomposition was not satisfactory. We see that some of the vertices seem much too large, and so do the links between the vertices. In fact, based on its contents, one may argue that amino-acid is not really a common-sense KB to begin with.

The experimental results are shown in Figure 6.1 (amino-acid has been omitted from this table for the reasons mentioned above).

The use of timeouts makes the results somewhat harder to interpret: they, of course, introduce uncertainty about how long a query really would have taken. It seems prudent to distinguish between the four kinds of timeout combinations that can arise for any given query:

1. No timeouts for either algorithm in any of the 10 iterations of the query. A typical example of this is listed in Figure 6.2(i). This was the most common situation for the KBs, except
<table>
<thead>
<tr>
<th>KB</th>
<th>wines</th>
<th>miniTambis</th>
<th>people+pets</th>
<th>university</th>
</tr>
</thead>
<tbody>
<tr>
<td>#queries with no timeouts</td>
<td>2</td>
<td>961</td>
<td>632</td>
<td>752</td>
</tr>
<tr>
<td>timeouts only for the</td>
<td>67</td>
<td>16</td>
<td>86</td>
<td>118</td>
</tr>
<tr>
<td>plain algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>timeouts only for the</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>decomposition-based algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>timeouts for both algorithms</td>
<td>17</td>
<td>169</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>plain algorithm faster</td>
<td>0%</td>
<td>27%</td>
<td>9%</td>
<td>6%</td>
</tr>
<tr>
<td>in % of queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decomposition-based algorithm</td>
<td>78%</td>
<td>56%</td>
<td>85%</td>
<td>86%</td>
</tr>
<tr>
<td>faster in % of queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total #operations of</td>
<td>47%</td>
<td>95%</td>
<td>51%</td>
<td>22%</td>
</tr>
<tr>
<td>decomposition-based vs. plain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>algorithm, over all queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Test results.

As the second row of the results table shows, there were only two wines queries without timeouts. Hence it seems wines is still too large. However, as with all four KBs, there were never timeouts for only the decomposition-based algorithm; always for either the plain algorithm or for both algorithms.

(2) Timeouts only for the plain algorithm, as shown in Figure 6.2(ii). This happened fairly often.

(3) Timeouts for both algorithms, such as for the queries in Figures 6.2(iii) and 6.2(iv). This occurred mostly in the wines and miniTambis KBs.

(4) Timeouts only for the decomposition-based algorithm. Interestingly, this never happened, which seems to suggest that the new heuristics indeed prevent "bad choices" from being made by the tableau algorithm. One can also conclude that even though the new algorithm performs worse than the standard one, it is never much slower.

The 6th and 7th rows of the results table indicate in what percentage of the queries the plain resp. decomposition-based algorithm was faster. The decomposition-based algorithm is considered "faster" than the plain one for a query \( \varphi \) if either (a) \( \varphi \) never results in a timeout for the decomposition-based algorithm, but it does for the plain one, or (b) \( \varphi \) never results in a timeout for either algorithm, and the decomposition-based algorithm takes less operations than the plain one, averaged over the 10 iterations of \( \varphi \).

We can see that for all KBs except miniTambis, the plain algorithm was almost never faster for any of the queries. For miniTambis, it did answer about 1/4th of the queries more quickly, while the new algorithm was faster for about half the queries. As the 7th row of the table also shows, for the other KBs, the decomposition-based algorithm performed better in a large majority of the queries.

The last (8th) row of the table shows how much faster the new algorithm was, averaged over all queries — including those with timeouts. It measures the number of operations taken by the decomposition-based algorithm versus the plain one, so the lower the percentage, the better. Here the results also seem quite good, except for miniTambis. Apparently, the latter was due to a lot of queries resulting in a timeout for both algorithms, whereas queries without timeouts generally took only a few operations. This causes the timeouts to dominate the percentage; taking only queries without any timeouts into account brings the percentage down to 83 instead of 95.
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Plain #Operations</th>
<th>Decomposition #Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124</td>
<td>59</td>
</tr>
<tr>
<td>2</td>
<td>125</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>151</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>159</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>70</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>79</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>83</td>
<td>43</td>
</tr>
</tbody>
</table>

(i) A miniTambis query.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Plain #Operations</th>
<th>Decomposition #Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>71</td>
</tr>
<tr>
<td>3</td>
<td>1204</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>259</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>43</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>35</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>2000</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
<td>53</td>
</tr>
<tr>
<td>9</td>
<td>2000</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>1864</td>
<td>55</td>
</tr>
</tbody>
</table>

(ii) A university query.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Plain #Operations</th>
<th>Decomposition #Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>263</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>426</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>127</td>
<td>2000</td>
</tr>
<tr>
<td>7</td>
<td>157</td>
<td>2000</td>
</tr>
<tr>
<td>8</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>9</td>
<td>418</td>
<td>2000</td>
</tr>
<tr>
<td>10</td>
<td>452</td>
<td>2000</td>
</tr>
</tbody>
</table>

(iii) A people+pets query.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Plain #Operations</th>
<th>Decomposition #Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>134</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>5</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>6</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>2000</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>2000</td>
</tr>
<tr>
<td>9</td>
<td>35</td>
<td>2000</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>2000</td>
</tr>
</tbody>
</table>

(iv) Another people+pets query.

Figure 6.2: Representative examples for various kinds of timeout combinations.
6.2.1 Comparison with results obtained by Amir et al.

Directly comparing these results to the ones obtained by Amir et al. [MMAU03] is not easy. As opposed to this thesis, they have modified an existing resolution-based theorem prover (SNARK), which means their results are much more realistic. Furthermore, their experiments were performed using only one large KB, on which nine queries were performed. Each query was run once using a variety of reasoning algorithms, some of which are incomplete. By contrast, for this thesis, hundreds of queries were run (10 times) on four rather small KBs.

We will therefore refrain from attempting to contrast the results reported by Amir et al. with the ones in this thesis.

6.2.2 Results using Lyndon’s interpolation property

Due to time constraints, only a limited amount of tests using Lyndon’s interpolation property was carried out. In such tests, we consider both positive and negative occurrences of atomic concepts in a formula (cf. § 2.3.1). This meant that “finer” tree decompositions were obtained: more, smaller vertices. Although this would seem like a good thing, the results seemed to disagree. Most of the queries were not answered much faster (sometimes even slower) than when using normal tree decompositions. However, the exact cause of this has not been investigated.
Chapter 7

Conclusions

7.1 Summary of Contributions

In Chapters 2 and 3, a different (partial) account of [Ami02] and [MMAU03] has been given. The intention was to present a number of concepts more precisely and more clearly. Below are some of the most important differences.

- Amir describes his reasoning algorithm by a listing of pseudocode [Ami02, p. 55], whereas this thesis formulates this process as the $\vdash_\sigma$ relation given in Definition 2.2.4. This makes it easier to prove soundness, termination, and completeness (although the former two are rather trivial). It also naturally leads to the observation that the decomposition-based reasoning method may be viewed as a labelled deductive system, or as a “restricted” variant of (say) resolution.

- Amir’s reasoning algorithm operates on undirected graphs, and starts by determining a relation $\prec$ dictating the direction in which formulas may be “sent” between vertices. In this thesis, we work with directed graphs, and the directions of the edges determine how formulas may be exchanged, an approach which is less convoluted.

- The completeness proof in this thesis is somewhat simpler than Amir’s. In essence, his one inductively breaks the theory tree into two arbitrary parts, while in this thesis we simply start with an arbitrary leaf and continue from there.

In the experiments reported on in [MMAU03], only queries were used whose signature is completely contained in that of some vertex of the theory tree. In [Ami02, § 4.2.3], Amir also makes this assumption, although he does briefly suggest decomposing a propositional query. In § 3.2 of this thesis, it is suggested that arbitrary queries can be answered by making some “small” (depending on the circumstances) modifications to the theory tree after adding the query.

The suggestion that we may use Lyndon’s interpolation theorem instead of Craig’s is also new, although the experimental results for this variant did not seem as promising.

The main original work done in this thesis is the adaptation of Amir’s method from resolution to semantic tableaux, as covered in Chapter 4. This adaptation should be usable for any logic for which tableau algorithms exist. The resulting approach was then applied to description logics and verified empirically, with encouraging results. Even though these results are not definitive, it seems plausible that more realistic tests (including existing reasoner optimizations) might confirm the usefulness of this approach.
7.2 Related Work

Improving the performance of reasoning for description logics (or any tableau algorithm, for that matter) using the method proposed in this thesis has apparently not been attempted before. Specifically concerning description logics, work by Grau [Gra05, GPSK06] also considers automatically decomposing description logic KBs. However, his aim is not to improve efficiency of reasoning but to isolate “modules” of KBs that are intuitively self-contained. In particular, it appears that the original KB need not even be a conservative extension of an extracted module.

More generally, a detailed overview of the use of “structure in reasoning” in reasoning is presented in [Ami02, § 4.6].

7.3 Future Directions

As mentioned before, for more realistic experiments one would need to integrate the decomposition-based heuristics into an existing DL reasoner. In order to do that, we would also need to decide how to combine existing heuristics and other optimizations (as introduced in § 5.5) with the algorithm proposed in this thesis. This presents both theoretical and practical challenges.

Another interesting direction to explore was hinted at in § 6.2. The only KB for which the new algorithm did not perform well was amino-acid, whose tree decomposition coincidentally also exhibited undesired characteristics (cf. Figure A.2). Hence, hopefully we can compute a tree decomposition and then look at features such as the number of vertices, their sizes, and the sizes of the shared vocabularies (i.e. the “links” between vertices). That way, we may be able to predict whether or not applying the decomposition-based heuristics would be beneficial.
Appendix A

Example Tree Decompositions

Please note that the indices in these figures do not quite correspond to the properties stated on p. 23: the requirement $idx(x) = idx(y)$ iff $parent(x) = parent(y)$ is not met.

Figure A.1: Excerpt from university.
Figure A.2: Excerpt from amino-acid.
Figure A.3: Excerpt from animals.
Figure A.4: Excerpt from miniTambis.
Figure A.6: Excerpt from people+pets.
Bibliography


