MASTER SCIENCES COGNITIVES ET APPLICATIONS
SPÉCIALITÉ TRAITEMENT AUTOMATIQUE DU LANGAGE

MASTER THESIS, JUNE 2013

Revision-based adaptation in propositional logic using a set of rules: study, algorithm and implementation

Author:
Gabin PERSONENI

Supervisors:
Jean LIEBER
Alice HERMANN

Acknowledgments

I would like to thank the following people for their help and support for their assistance and support throughout this internship. First and foremost I would like to thank Jean Lieber and Alice Hermann for:

- letting me in the REVISOR project, welcoming me into the team, and having me participate in the writing of a scientific article on this project,
- entrusting me with the authoring of a second scientific article, and their valuable participation in the writing of said article,
- their availability and valuable advices.

Then, I would like to thank each of the (many) authors of the first article; Valmi Dufour-Lussier for his assistance on the few technical issues that I encountered. The anonymous reviewers of both articles are also to be thanked for their constructive criticism which ultimately led to an improvement of these articles and indirectly of the present report. Finally, I would like to thank the other students of the Master, for the mutual support and inducement among us.

Table of Contents

Introduction

Case-based reasoning [Riesbeck and Schank, 1989] is a reasoning paradigm based on analogy making. Through the reuse of previous problem-solving experiences, called *cases*, a new problem, called the *target case* can be solved. Case-based reasoning is usually performed in four steps: firstly, a case similar to the target case is selected in the case base, then that case, called the source case, is *adapted* in order to solve the target case. The suggested solution is then revised based on user feedback. Finally the solution is stored in the case base to be reused. Thereby, case-based reasoning differs from most artificial intelligence techniques, for it is memory-based. This paradigm is inspired from cognitive sciences, as it mimics human reasoning that largely relies on previous experiences.

The objective of this internship was to design an adaptation operator to be used in case-based reasoning. That adaptation operator had to combine two distinct adaptation paradigms: revision-based adaptation, introduced by Jean Lieber in [Lieber, 2007], and rule-based adaptation. Revision-based adaptation consists in modifying minimally the source case so that it solves the target case. Revision-based adaptation is dependent of how modifications are measured. Rule-based adaptation accounts for adaptation knowledge in the form of *adaptation rules* or reformulations [Melis et al., 1998]. These rules represent possible substitutions within the source case, and are applied to it so that it solves the target case.

An adaptation operator in the propositional logic formalism, that is for cases expressed as propositional formulae, has been designed and the corresponding algorithm has been implemented, within the REVISOR library, a collection of revision engines in different formalisms. The algorithm is based on both the semantics tableaux method, a proof procedure used in many logical formalisms, and on the A^* algorithm, a widely used and efficient graph search algorithm. This allows for short computing times, with respect to the complexity of the problem.

Chapter 1 presents the scientific context of this internship. Chapter 2 introduces the logical formalism and notions used throughout this report. The revision process is specified for this formalism, and algorithms implementing revision operators are presented. Chapter 3 explains the role of the adaptation process in case-based reasoning and different specifications of this process. Chapter 4 expresses the adaptation problem as a search problem, then presents the A^* algorithm. Chapter 5 details the adaptation algorithm that has been designed, and proves it returns the expected results. A study of its complexity is also provided. Section 6 presents an implementation of this algorithm within REVISOR, a collection of revision and adaptation engines. Finally, Chapter 7 concludes and presents some future work.

Chapter 1

Scientific Context

This Chapter presents the scientific context of this internship. This internship was conducted in the Orpailleur team at LORIA. Orpailleur's research domains are knowledge discovery, data mining, semantic web and case-based reasoning. This internship was focused on case-based reasoning, and more precisely on belief revision and adaptation. This Chapter presents these reasoning paradigms. Section 1.1 introduces the case-based reasoning paradigm. Section 1.2 presents the belief revision process, and Section 1.3 shows how it can be used in the adaptation algorithm presented in this report.

1.1 Case-based Reasoning

Case-based reasoning is a reasoning paradigm based on the reuse of chunks of experience called *cases*. A case represents a problem-solving episode, that is the problem itself and one of its solutions. Intuitively, a case has a problem part and a solution part, though it is not formally necessary. A case-based reasoning system contains a case base, that is a collection of these problem-solving experiences. The input of a case-based reasoning system is a problem called the *target case*. It is a case whose solution part is ill-defined: the aim of the case-based reasoning system is to precise the solution part of the target case. A simple example of case-based reasoning is adapting cooking recipes: for instance, in order to cook an apple pie, one could use a recipe for a pear pie. Since apples and pears are similar fruits, a pear pie recipe can be adapted as an apple pie recipe by replacing pears with apples.

The R4 Model. In [Aamodt and Plaza, 1994], Aamodt and Plaza describe a case-based reasoning process in four steps, illustrated in figure 1.1, commonly called the R4 model or the "4 R's":

> **Retrieve.** For a given problem, the *Retrieve* step consists in retrieving from the case base a case similar to that problem. That retrieved case is also called the *source case*. For example, in order to solve the problem *"How to cook an apple pie ?"*, the system would search in the case base for matching recipes. Assuming the system does not have an apple pie recipe, it could retrieve an other pie recipe instead, for instance, a pear pie.

Reuse. The *Reuse* step consists in reusing the source case in order to specify the solution part of the target case. Usually the source case cannot be used to solve the target case without some changes: then, the source cased is modified, or *adapted* so that it can solve it. In the pie example, it consists in replacing the pears in the pie by apples. The difficulty of this step depends on how close to the target the retrieved source case is.

Revise. The *Revise* consists in collecting the feedback from the user of the case-based reasoning system. For example, after using the adapted pear pie recipe to cook an apple pie, the user could think that the apple pie is too sweet. He could then revise the solution by modifying the recipe so it contains less sugar.

Retain. Finally, the solved target case is added to the case base, to be used for further problem-solving episodes. Indeed, a larger case base make the case-based reasoning process more efficient and more accurate.

Note that several approaches can exist for each of these four steps. In particular, this report presents an adaptation algorithm that specifies the adaptation process of the *Reuse* step, here called the *adaptation* step. That step of case-based reasoning has only recently received attention in the case-based reasoning literature[1]. Yet, this step is necessary for the case based-reasoning system to be able to suggest solutions to new problems.

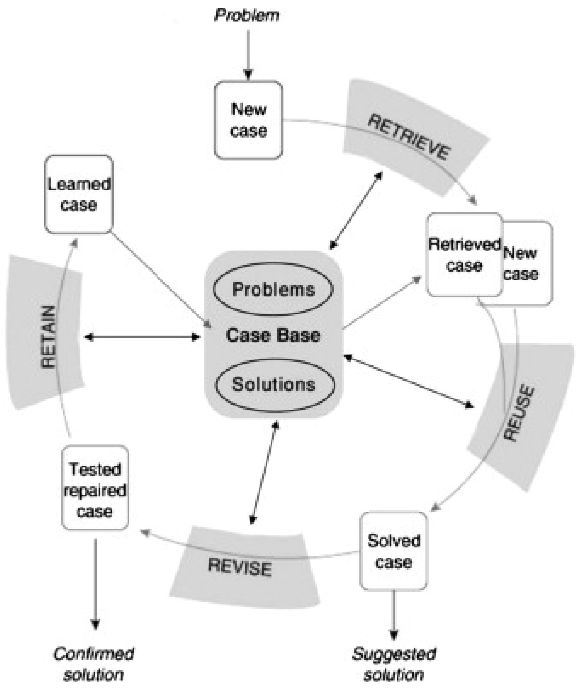


Figure 1.1: R4 case-based reasoning process [Aamodt and Plaza, 1994] .

[1]See, e.g, [Minor et al., 2007, Manzano et al., 2011, Coman and Muñoz-Avila, 2012, Rubin and Watson, 2012]

1.2 Belief Revision

Belief revision is a process consisting in incorporating new beliefs to a belief base. For this process, new beliefs are considered more reliable than older ones: if a new belief is inconsistent with the belief base, old beliefs are minimally modified to become consistent with the new. In other words, revision maximizes the amount of old beliefs that stay in the revised belief base. However, if the old and new beliefs are consistent, then the new ones are added to the belief base without changing the old ones. The main challenge in belief revision is to modify old beliefs as less as possible: thus, ways to measure changes in a belief base must be defined.

Revision is performed on beliefs describing a static world, that is, beliefs that are generally true rather than true at only one given moment. For example, the belief *"it is snowing"* is typically not about a static world, as it describes a transient state of the world, and thus could be outdated at some point. It would not make sense to revise a belief that is outdated, as it would not be necessary to keep such a belief in the belief base. However the belief *"if it snows, then it is cold outside"* is true, independently of the moment it is evaluated. Intuitively, belief revision does not reflect a change of the world, but rather a change of one's belief about that world.

Because of the many possible ways to measure change in a belief base, multiple revision operators exists. The AGM postulates defined in [Alchourrón et al., 1985] are accepted as a set of properties a belief revision operator should have.

1.3 Adaptation

Adaptation is one of the steps of the case-based reasoning process, also called the *Reuse* step. The adaptation algorithm presented in this report is based on the approach now called *revision-based adaptation*[2] or $\dot{+}$-*adaptation*, introduced in [Lieber, 2007]. This approach is based on a belief revision operator, $\dot{+}$. Revision-based adaptation takes into account *domain knowledge*, that is knowledge that does not belong to one case, but can be used to better understand the cases. For example, *"Apples and pears are fruits"* can be considered as domain knowledge.

The adaptation approach presented in this report is original as it proposes to account for domain and *adaptation knowledge*. Adaptation knowledge represents ways to adapt a source case to solve the target case. For example, *"In a pie recipe, apples can replace pears"* provides information on how a pear pie recipe can be adapted to cook an apple pie. The presented approach will account for that adaptation knowledge in the form of *adaptation rules*, also called reformulations in [Melis et al., 1998].

[2]Note that *revision-based adaptation* does not takes its name from the *Revise* step of the R4 Model, but rather from the belief revision process described in Section 1.2.

Chapter 2

Preliminaries

This Chapter provides necessary notions that will be used throughout that report. Section 2.1 first introduces propositional logic, the formalism in which the adaptation operator presented in this report will be defined. Then, Section 2.2 explains the notion of mathematical distances, and how these distances are used in propositional logic. Section 2.3 describes how belief revision is performed in propositional logic. Section 2.4 describes the notion of *implicant*, and Section 2.5 presents one procedure to compute implicants: the semantic tableaux method. Finally, Section 2.6 presents how implicants are used to perform belief revision.

2.1 Propositional Logic

Let $\mathcal{V} = \{a_1, \ldots, a_n\}$ be a set of n distinct symbols called propositional variables. A propositional formula built on $\mathcal{V}$ is either:

- a variable a_i,
- the conjunction of two propositional formulae: $\varphi_1 \wedge \varphi_2$,
- the disjunction of two propositional formulae: $\varphi_1 \vee \varphi_2$,
- the negation of a propositional formula: $\neg\varphi_1$,
- an implication of two propositional formulae: $\varphi_1 \Rightarrow \varphi_2$,
- an equivalence of two propositional formulae: $\varphi_1 \Leftrightarrow \varphi_2$,

where φ_1 and φ_2 are two propositional formulae. A literal ℓ is a propositional formula of the form a_i (positive literal) or $\neg a_i$ (negative literal), where $a_i \in \mathcal{V}$. If $\ell = \neg a_i$ is a negative literal, then $\neg\ell$ denotes the positive literal a_i (instead of the equivalent formula $\neg\neg a_i$). Let $\mathcal{L}$ be the set of the propositional formulae.

Interpretations and Models. Let $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$ be the set of booleans, where $\mathtt{T}$ stands for *true*, and $\mathtt{F}$ for *false*. An interpretation x is defined as an ordered n-tuple of booleans: $(x_1, \ldots, x_n)$. $\mathbb{B}^n$ represents the set of all interpretations. φ^x is the truth value of φ interpreted by x, and evaluates to:

- x_i, if $\varphi = a_i$,
- $(\varphi_1 \wedge \varphi_2)^x = \mathtt{T}$ iff $\varphi_1^x = \mathtt{T}$ and $\varphi_2^x = \mathtt{T}$,
- $(\varphi_1 \vee \varphi_2)^x$ iff $\varphi_1^x = \mathtt{T}$ or $\varphi_2^x = \mathtt{T}$,
- $(\neg\varphi_1)^x = \mathtt{T}$ iff $\varphi_1^x = \mathtt{F}$,
- $(\varphi_1 \Rightarrow \varphi_2)^x = (\neg\varphi_1 \vee \varphi_2)^x$, and
- $(\varphi_1 \Leftrightarrow \varphi_2)^x = ((\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1))^x$.

For $\varphi \in \mathcal{L}$, let $\mathcal{M}od(\varphi) = \{x \in \mathbb{B}^n \mid \varphi^x = \mathtt{T}\}$ called the set of models of φ. The two relations $\models$ and $\equiv$ between formulae are defined as follows: let φ_1, φ_2 be two propositional formulae,

$$\varphi_1 \models \varphi_2 \text{ iff } \mathcal{M}od(\varphi_1) \subseteq \mathcal{M}od(\varphi_2),$$
$$\varphi_1 \equiv \varphi_2 \text{ iff } \mathcal{M}od(\varphi_1) = \mathcal{M}od(\varphi_2).$$

$(\mathcal{L}, \models)$ is called the propositional logic with n variables.

Disjunctive Normal Form. A formula is in disjunctive normal form or DNF if it is a disjunction of clauses, where clauses are literals or conjunctions of literals. That is, a formula φ is in DNF, iff it is under one of these forms:

- $\varphi = \top$, where $\top$ is the disjunction of an empty clause.
- $\varphi = \bot$, where $\bot$ is the disjunction of zero clauses.
- $\varphi = \ell_1 \wedge \ldots \wedge \ell_n$, for $n \geq 1$ and n literals $\ell_1, \ldots, \ell_n$, where $\ell_1 \wedge \ldots \wedge \ell_n$ is a single clause.
- $\varphi = \varphi_1 \vee \varphi_2$, for any two formulae φ_1 and φ_2 in DNF.

A formula in DNF is satisfiable iff at least one of its clauses is satisfiable, and a clause is satisfiable iff it does not contain a literal ℓ and its negation $\neg\ell$. Note that a formula can have several distinct DNF.

Conjunctive Normal Form. Conjunctive normal form or CNF is the converse form to the DNF. A formula is in conjunctive normal form is a conjunction of clauses, where clauses are literals or disjunctions of literals.

Negative Normal Form. A formula is in negative normal form or NNF if it contains only the connectives $\wedge$, $\vee$ and $\neg$, and if $\neg$ appears only in front of propositional variables. Every formula can be put in NNF by applying, as rewriting rules oriented from left to right, the following equivalences, until none of these equivalences is applicable (for $\varphi, \varphi_1, \varphi_2 \in \mathcal{L}$):

$$\begin{aligned}
\varphi_1 \Leftrightarrow \varphi_2 &\equiv (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \\
\varphi_1 \Rightarrow \varphi_2 &\equiv \neg\varphi_1 \vee \varphi_2 \\
\neg\neg\varphi &\equiv \varphi \\
\neg(\varphi_1 \vee \varphi_2) &\equiv \neg\varphi_1 \wedge \neg\varphi_2 \\
\neg(\varphi_1 \wedge \varphi_2) &\equiv \neg\varphi_1 \vee \neg\varphi_2
\end{aligned}$$

A formula in DNF or in CNF is necessarily in NNF (the converse is false).

For example, the formula $\neg(\neg a \wedge b)$ can be put in NNF in a few steps: first using the equivalence $\neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2$, it becomes $(\neg\neg a \vee \neg b)$. Then, the double negation is removed, using the equivalence $\neg\neg\varphi \equiv \varphi$. Thus, the formula is in the form $(a \vee \neg b)$, and since the $\neg$ connective only appears preceding a propositional variable, the formula is therefore in NNF.

2.2 Distances

In mathematics, a distance d on a set $\mathcal{U}$ is a function $d : \mathcal{U} \times \mathcal{U} \to \mathbb{R}$ that verifies these four properties, for any elements of $\mathcal{U}$, x, y and z:

- Non-negativity: $d(x, y) \geq 0$
- Identity of indiscernibles: $d(x, y) = 0$ iff $x = y$
- Symmetry: $d(x, y) = d(y, x)$
- Subadditivity: $d(x, z) \leq d(x, y) + d(y, z)$

Distances on Interpretations. Distances can be defined on $\mathbb{B}^n$, the set of propositional logic interpretations. Let d be such a distance, φ a propositional formula, and x, y two interpretations in $\mathbb{B}^n$. The following shortcut is used:

$$d(\varphi, y) = \inf_{x \in \mathcal{M}od(\varphi_1)} d(x, y)$$

Indeed, a formula can be represented as the set of interpretations that satisfies it, thus, the distance between an interpretation y and a formula φ can be defined as the minimal distance between y and models of φ. Similarly, the distance between two formulae φ_1 and φ_2 can be defined as:

$$d(\varphi_1, \varphi_2) = \inf_{y \in \mathcal{M}od(\varphi_2)} d(\varphi_1, y)$$

Two properties can be deduced from these definitions: $d(\varphi, y) = 0$ iff y is a model of φ, and $d(\varphi_1, \varphi_2) = 0$ iff $\varphi_1 \wedge \varphi_2$ is consistent.

Hamming Distance. In [Hamming, 1950], Hamming defines a distance function on binary words of same length, and was first used in telecommunication to measure error in transmissions. An error in a transmission, or a *flip*, is characterized as a bit that differs from the original word and the transmitted word. The Hamming distance, noted d_H, is equal to the numbers of flips. Being n-tuples of booleans, propositional logic interpretations can be considered as binary words. Thus, the Hamming distance on the set of propositional logic interpretations, $\mathbb{B}^n$ is defined by, for $x, y \in \mathbb{B}^n$:

$$d_H(x, y) = |\{i \mid i \in \{1, 2, \ldots, n\}, x_i \neq y_i\}|$$

In other words, $d_H(x, y)$ is equal to the number of variables with a different interpretation in x and y. A flip in an interpretation x is defined as the change of truth value of one propositional variable a_i: this operation is also called a flip on a_i, noted $\mathtt{F}_{a_i}$.

2.3 Belief Revision in Propositional Logic

Let ψ be the beliefs of an agent, expressed as a propositional logic formula. The agent is confronted to new beliefs expressed by the formula μ. The new beliefs of the agent is denoted by $\psi \dotplus \mu$. If $\psi \wedge \mu$ is consistent, then $\psi \dotplus \mu = \psi \wedge \mu$. Otherwise, according to the *minimal change principle* [Alchourrón et al., 1985], ψ has to be modified minimally into a formula ψ' such that $\psi' \wedge \mu$ is consistent. Then $\psi \dotplus \mu = \psi' \wedge \mu$.

Revision Operators. A revision operator is defined by the distance function it uses to measure changes in the belief base, as such, multiple revision operators can be defined. The Dalal revision operator [Dalal, 1988] is based on the previously-defined Hamming distance. There exists a set of postulates defining the properties of a revision operator, called the AGM postulates, described by Alchourron, Gärdenfors and Makinson in [Alchourrón et al., 1985]. In [Katsuno and Mendelzon, 1991], Katsuno and Mendelzon define a new set of postulates equivalent to the AGM postulates for revision operators in propositional logic. For any belief revision operator $\circ$ in propositional logic:

(R1) $\psi \circ \mu$ implies μ.

(R2) If $\psi \wedge \mu$ is satisfiable then $\psi \circ \mu \equiv \psi \wedge \mu$.

(R3) If μ is satisfiable then $\psi \circ \mu$ is also satisfiable.

(R4) If $\psi_1 \equiv \psi_2$ and $\mu_1 \equiv \mu_2$ then $\psi_1 \circ \mu_1 \equiv \psi_2 \circ \mu_2$.

(R5) $(\psi \circ \mu) \wedge \varphi$ implies $\psi \circ (\mu \wedge \varphi)$.

(R6) If $(\psi \circ \mu) \wedge \varphi$ is satisfiable then $\psi \circ (\mu \wedge \varphi)$ implies $(\psi \circ \mu) \wedge \varphi$.

Let ψ and μ be two formulae, $\psi \dotplus^d \mu$ is a formula whose models are the models of μ that are the closest ones to those of ψ, according to the distance function d:

$$\mathcal{M}od(\psi \dotplus^d \mu) = \{y \in \mathcal{M}od(\mu) \mid d(\psi, y) = d(\psi, \mu)\}$$

a	b	$\psi = a \wedge b$	$\mu = \neg a \vee \neg b$
F	F	F	T
F	T	F	T
T	F	F	T
T	T	T	F

Figure 2.1: Truth table of the formulae $\psi = (a \wedge b)$ and $\mu = \neg a \vee \neg b$.

Note that this definition specifies $\psi \dot{+}^d \mu$ up to logical equivalence, but this is sufficient since a revision operator has to satisfy the irrelevance of syntax principle, according to the **(R4)** postulate.

The propositional formula $\psi \dot{+}^d \mu$ can be written as:

$$\bigvee_{x \in \mathcal{M}od(\psi \dot{+}^d \mu)} \varphi_x \mid \mathcal{M}od(\varphi_x) = x$$

$$\varphi_x = \bigwedge_{a_i \in \mathcal{V}} \begin{cases} a_i, \text{ if } x_i = \mathtt{T} \\ \neg a_i, \text{ if } x_i = \mathtt{F} \end{cases}$$

Example. This example describes the computation of $\psi \dot{+}^{d_H} \mu$, for $\psi = a \wedge b$ and $\mu = \neg a \vee \neg b$. Firstly, it is necessary to compute $\mathcal{M}od(\psi)$ and $\mathcal{M}od(\mu)$. Figure 2.1 shows the truth table of ψ and μ. ψ has one model $(\mathtt{T}, \mathtt{T})$ and μ has three: $(\mathtt{F}, \mathtt{F})$, $(\mathtt{F}, \mathtt{T})$, $(\mathtt{T}, \mathtt{F})$. Then, d_H is computed between every model of ψ and every model of μ, by counting the differences between each model:

$$d_H((\mathtt{T}, \mathtt{T}), (\mathtt{F}, \mathtt{F})) = 2$$
$$d_H((\mathtt{T}, \mathtt{T}), (\mathtt{F}, \mathtt{T})) = 1$$
$$d_H((\mathtt{T}, \mathtt{T}), (\mathtt{T}, \mathtt{F})) = 1$$

The distance is minimal between the models $(\mathtt{T}, \mathtt{T})$ of ψ and $(\mathtt{F}, \mathtt{T})$ of μ, and $(\mathtt{T}, \mathtt{T})$ of ψ and $(\mathtt{T}, \mathtt{F})$ of μ. Thus $\mathcal{M}od(\psi \dot{+}^{d_H} \mu) = \{(\mathtt{F}, \mathtt{T}), (\mathtt{T}, \mathtt{F})\}$, and $\psi \dot{+}^{d_H} \mu = (\neg a \wedge b) \vee (a \wedge \neg b)$.

2.4 Implicant Theory

A set of literals $\mathtt{L}$ is often assimilated to the conjunction of its elements, for example $\{a, \neg b, \neg c\}$ is assimilated to $a \wedge \neg b \wedge \neg c$ and vice-versa. In particular, $\mathtt{L}$ is satisfiable iff there is no literal $\ell \in \mathtt{L}$ such that $\neg \ell \in \mathtt{L}$.

An implicant of φ, I is a conjunction or a disjunction of literals, such that $I \models \varphi$ and I is satisfiable. However, as there exists a duality between conjunctive and disjunctive implicants, only conjunctive implicants will be considered. An implicant I of φ is prime if for any conjunction of literals C such that $C \subset I$, $C \not\models \varphi$. That is, a prime implicant I is minimal. Let $\mathtt{PI}(\varphi)$

be the set of prime implicants of φ. Then:

$$\varphi \equiv \bigvee_{I \in \mathtt{PI}(\varphi)} I$$

Note that, for a formula φ in the form of a conjunction of literals, $\mathtt{PI}(\varphi) = \{\varphi\}$. An algorithm to find efficiently prime implicants of a formula is detailed in [Socher, 1991].

2.5 Semantic Tableaux

In propositional logic, the method of semantic tableaux is used to determine the satisfiability of a formula. Intuitively, it consists in transforming the formula into one of its DNF: it is then sufficient to prove the satisfiability of one of its conjunctive clauses to prove the satisfiability of the formula. The semantic tableau of a formula φ is a tree structure representing φ, in which a branch represents a clause, and the disjunction of all the clauses is the DNF of φ.

Expansion Rules. Initially, the tableau of a formula φ is a tree with a single node representing φ. Assuming φ is in NNF, tableaux calculus defines two expansion rules $(\vee)$ and $(\wedge)$ that can respectively be applied on nodes representing formulae in the form of $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$:

- The $(\vee)$ rule is applied on nodes representing a disjunctive formula.

$$\frac{\varphi_1 \vee \varphi_2}{\varphi_1 \mid \varphi_2} \tag{$\vee$}$$

  That is, any node representing a disjunction can be expanded by splitting the current branch in two new branches, and adding respectively the first and second operands to the first and second branches:

$$\begin{tikzcd} & \varphi_1 \vee \varphi_2 \arrow[dl, no head] \arrow[dr, no head] & \\ \varphi_1 & & \varphi_2 \end{tikzcd}$$

- The $(\vee)$ rule is applied on nodes representing a conjunctive formula.

$$\frac{\varphi_1 \wedge \varphi_2}{\begin{array}{c}\varphi_1 \\ \varphi_2\end{array}} \tag{$\wedge$}$$

  That is, any node representing a conjunction can be expanded by adding its two operands to the current branch:

$$\begin{tikzcd} \varphi_1 \wedge \varphi_2 \arrow[d, no head] \\ \varphi_1 \arrow[d, no head] \\ \varphi_2 \end{tikzcd}$$

Note that nodes can be expanded at most once in each branch. The procedure stops when no more nodes can be expanded.

Figure 2.2 presents an example of the tableau of the formula $\neg(\neg a \land b) \land (c \lor d)$ built using these two rules. The formula must first be put into NNF: using the equivalences defined in Section 2.1, it can be deduced that $\neg(\neg a \land b) \land (c \lor d) \equiv (a \lor \neg b) \land (c \lor d)$. The $(\land)$ rule is first applied to the conjunction of $(a \lor \neg b) \land (c \lor d)$, so $a \lor \neg b$ and $c \lor d$ are added to the branch. Then the $(\lor)$ rule is applied on $a \lor \neg b$, and the branch is split in two. The formula a is added to one of the new branches, while $\neg b$ is added to the other. The $(\lor)$ is similarly applied on $c \lor d$. Note that, at this point, $c \lor d$ appears in two branches. Thus, these two branches are both split in two, and c and d both appear twice in the leaves of the tree.

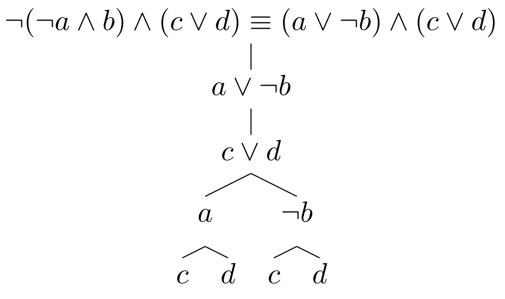

$\neg(\neg a \land b) \land (c \lor d) \equiv (a \lor \neg b) \land (c \lor d)$
$a \lor \neg b$
$c \lor d$
a
$\neg b$
c
d
c
d

Figure 2.2: Fully-expanded tableau for the formula $\neg(\neg a \land b) \land (c \lor d)$.

Clashes and Satisfiability. A branch contains a clash when there are a literal ℓ and its opposite literal $\neg\ell$ in this branch, it is said there is a clash on ℓ. As branches represents conjunctions of literals, a branch containing a clash is equivalent to $\bot$. Thus, the branch need not to be expanded: adding any literal ℓ to the branch would not affect the semantics of the formula it represents, as $\bot \land \ell \equiv \bot$.

If each branch contains at least one clash, then the formula is unsatisfiable: each branch is equivalent to $\bot$, and a DNF of the formula is a disjunction between them, thus the formula is equivalent to $\bot$. Conversely, if at least one branch does not contain a clash, then the formula is satisfiable, since its DNF contains a satisfiable clause. Branches without clashes are said to be consistent.

Simple Algorithm. Let `branches` be a function that for any propositional formula φ in NNF, returns the set of tableaux branches of φ. Each branch is represented by the set of every literal it contains. For any formula φ in NNF, $\texttt{branches}(\varphi)$ is defined as:

- If φ is a literal, such that $\varphi = \ell$, then: $\texttt{branches}(\varphi) = \{\{\ell\}\}$.
- If φ is in the form $\varphi_1 \lor \varphi_2$, with φ_1 and φ_2 two propositional formulae, then: $\texttt{branches}(\varphi) = \texttt{branches}(\varphi_1) \cup \texttt{branches}(\varphi_2)$.
- If φ is in the form $\varphi_1 \land \varphi_2$, with φ_1 and φ_2 two propositional formulae, then: $\texttt{branches}(\varphi) = \{B_1 \cup B_2 \mid (B_1, B_2) \in \texttt{branches}(\varphi_1) \times \texttt{branches}(\varphi_2) \text{ and } B_1 \cup B_2 \text{ consistent}\}$.

As φ is equivalent to the disjunction of its branches, the following equivalence holds for any propositional formula φ:

$$\varphi \equiv \bigvee_{B \in \mathtt{branches}(\varphi)} B \tag{2.1}$$

Tableaux in Implicant Theory. Let φ be a propositional formula. Branches of the tableau of φ can be seen as implicants of φ. Indeed, for any branch $B \in \mathtt{branches}(\varphi)$, it is easily deduced from (2.1) that $B \models \varphi$.

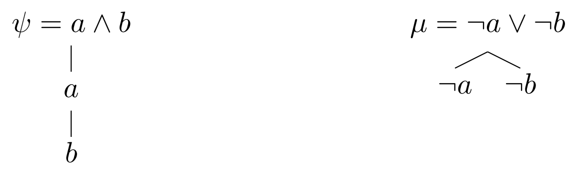
2.6 Implicant-based Revision

Implicant-based revision is a belief revision operation performed on set of implicants of formulae. Tableaux can also be used to perform implicant-based revision in propositional logic, using tableaux repairs [Schwind, 2011]: when revising a formula ψ by a formula μ, usually $\psi \wedge \mu$ is not satisfiable, that is every branch of its tableau contains a clash. Tableaux repairs consists in removing from the branches the literals causing clashes. This modifies the $\psi \wedge \mu$ so it becomes satisfiable.

Intuitively, implicant-based revision consists in finding implicants of ψ that are the closest to implicants of μ, according to some distance function. As implicants are formulae, computing the distance between two implicants can be done by computing the minimal distance between two of their respective models, as previously defined in Section 2.2.

$\psi \dot{+}^{d_H} \mu$ can be computed in a few steps. First, $\mathtt{branches}(\psi)$ and $\mathtt{branches}(\mu)$ are computed. Then every branch of ψ is combined with each branch of μ. If a branch contains a clash between a literal ℓ from ψ and a literal $\neg\ell$ from μ, ℓ from ψ is removed from the branch. Indeed, since ψ is being revised by μ, it is not possible to modify μ to repair the clash on ℓ. The operation of removing ℓ from ψ coincides with the flip operation defined in Section 2.2, as ℓ is removed in favor of $\neg\ell$. To each branch is associated a cost equal to the number of clashes repaired this way. Finally, $\psi \dot{+}^{d_H} \mu$ is determined by the disjunction of the branches of minimal cost.

Example. This example illustrates the computation of $\psi \dot{+}^{d_H} \mu$ for the two propositional formulae $\psi = a \wedge b$ and $\mu = \neg a \vee \neg b$. The figure 2.3 shows the computation of the tableaux branches of ψ and μ.



$\mathtt{branches}(\psi) = \{\{a, b\}\}$ $\qquad$ $\mathtt{branches}(\mu) = \{\{\neg a\}, \{\neg b\}\}$

Figure 2.3: Applying the tableaux method to $\psi = a \wedge b$ and $\mu = \neg a \vee \neg b$.

The single branch of ψ, $\{a, b\}$ can be combined with either of the two branches of μ, $\{\neg a\}, \{\neg b\}$. The union $\{a, b\} \cup \{\neg a\}$ results in a clash on a, while $\{a, b\} \cup \{\neg b\}$ results in a clash on b. These two clashes can be repaired by removing respectively a and b from them. Thus, the two branches $\{\neg a, b\}$ and $\{a, \neg b\}$ are obtained. As one clash has been repaired in each of them, the cost associated to them is of 1. Since there is no other branches with a cost less or equal to 1, $\psi \dot{+}^{d_H} \mu = (\neg a \wedge b) \vee (a \wedge \neg b)$, which of course coincides with the result found in Section 2.3.

Advantages of Implicant-based Revision. The truth table method used to compute revision with $\dot{+}^{d_H}$ has a time complexity of $O(2^n)$, as there is 2^n possible interpretations, where n is the number of variables. Implicant-based revision is also computed in exponential time, however, computing time does not depend on the number of variables n, but rather the branching factor in the tableaux of the formulae. Thus, computing time largely depends on the form of the formulae: computing the tableau of a formula in DNF can be done in linear time with respect to the size of the formula[1]. However computing the tableau of an equivalent formula in conjunctive normal form is done in exponential time, with respect to the number of clauses. The output of implicant-based revision is a set of tableaux branches, that is trivially changed into DNF. Consequent revisions operations on a belief base can take advantage of an output in DNF, as every revision beyond the first would be done on a belief base already in DNF.

[1]Here, the size of a formula relates to the total amount of literals occurrences in that formula. For example, the formula a is of size 1, $a \wedge a$ is of size 2, $a \wedge (b \vee \neg a)$ is of size 3.

Chapter 3

Adaptation in Case-based Reasoning

Let $(\mathcal{L}, \models)$ be the logic in which the knowledge containers of the case-based reasoning application are defined. A source case $\mathtt{Source} \in \mathcal{L}$ is a case of the case base. Often, such a case represents a specific experience: $\mathcal{M}od(\mathtt{Source})$ is a singleton. However, this assumption is not formally necessary (though it has an impact on the complexity of the algorithms). A target case $\mathtt{Target} \in \mathcal{L}$ represents a problem to be solved. This means that there is some missing information about $\mathtt{Target}$ and solving $\mathtt{Target}$ leads to adding information to it. So, adaptation of $\mathtt{Source}$ to solve $\mathtt{Target}$ consists in building a formula $\mathtt{CompletedTarget} \in \mathcal{L}$ that makes $\mathtt{Target}$ precise: $\mathtt{CompletedTarget} \models \mathtt{Target}$. To perform adaptation, the domain knowledge $\mathtt{DK} \in \mathcal{L}$ can be used. Therefore, the adaptation process has the following signature:

$$\mathtt{Adaptation} : (\mathtt{DK}, \mathtt{Source}, \mathtt{Target}) \mapsto \mathtt{CompletedTarget}$$

$(\mathtt{Source}, \mathtt{Target})$ is called the adaptation problem ($\mathtt{DK}$ is supposed to be fixed).

Of course, this does not completely specify the adaptation process. Several approaches are introduced in the case-based reasoning literature. Two of them are presented below: Section 3.1 presents the revision-based adaptation approach, then Section 3.2 presents rule-based adaptation. Finally, Section 3.3 presents a combination of these two approaches.

3.1 Revision-based Adaptation

Let $\dotplus$ be a revision operator in the logic $(\mathcal{L}, \models)$ used for a given case-based reasoning system. The $\dotplus$-adaptation is defined as follows:

$$\mathtt{CompletedTarget} = (\mathtt{DK} \wedge \mathtt{Source}) \dotplus (\mathtt{DK} \wedge \mathtt{Target})$$

Intuitively, the source case is modified minimally so that it satisfies the target case. Both cases are considered with respect to the domain knowledge.

Let $\psi = (\mathtt{DK} \wedge \mathtt{Source})$, $\mu = (\mathtt{DK} \wedge \mathtt{Target})$ and ψ' such that $\psi \dotplus \mu = \psi' \wedge \mu$. Revision-based adaptation consists in revising ψ by μ. Note that since $\mathtt{DK}$ appears in both ψ and μ, it will not be affected by the revision process, that is $\psi' \models \mathtt{DK}$. The advantage of using revision-based adaptation is that it always provides a result, which is not the case of all adaptation approaches.

In particular, next Section introduces an adaptation approach relying on adaptation knowledge that in some cases may fail to adapt the source case to the target case.

3.2 Rule-based Adaptation

Rule-based adaptation is a general approach to adaptation relying on domain-specific adaptation knowledge in the form of a set $\mathtt{AK}$ of adaptation rules (see, e.g., [Melis et al., 1998], where adaptation rules are called reformulations). An adaptation rule $\mathtt{R} \in \mathtt{AK}$, when applicable on the adaptation problem $(\mathtt{Source}, \mathtt{Target})$, maps $\mathtt{Source}$ into $\mathtt{CompletedTarget}$ which makes $\mathtt{Target}$ precise. Adaptation rules can be composed (or chained): if $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_q \in \mathtt{AK}$ are such that there exist $q+1$ cases $\mathtt{C}_0$, $\mathtt{C}_1$, ..., $\mathtt{C}_q$ verifying:

- $\mathtt{C}_0 = \mathtt{Source}$,
- $\mathtt{C}_q$ makes $\mathtt{Target}$ precise ($\mathtt{C}_q \models \mathtt{Target}$),
- for each $i \in \{1, \ldots, q\}$, $\mathtt{R}_i$ is applicable on $(\mathtt{C}_{i-1}, \mathtt{C}_i)$ and maps $\mathtt{C}_{i-1}$ into $\mathtt{C}_i$,

then $\mathtt{C}_q = \mathtt{CompletedTarget}$ is the result of the adaptation of $\mathtt{Source}$ to solve $\mathtt{Target}$. The sequence $\mathtt{R}_1; \mathtt{R}_2; \ldots; \mathtt{R}_q$ is called an *adaptation path*.

Given an adaptation problem $(\mathtt{Source}, \mathtt{Target})$, there may be several adaptation paths to solve it. In order to make a choice among them, a cost function is introduced: $\mathtt{cost} : \mathtt{R} \in \mathtt{AK} \mapsto \mathtt{cost}(\mathtt{R}) > 0$. The cost of an adaptation path is the sum of the costs of its adaptation rules.

An adaptation rule $\mathtt{R}$ is defined by two sets of literals, $\mathtt{left}$ and $\mathtt{right}$, and is denoted by $\mathtt{R} = \mathtt{left} \rightsquigarrow \mathtt{right}$. Let $(\mathtt{Source}, \mathtt{Target})$ be an adaptation problem. Several cases have to be considered:

- If $\mathtt{Source}$ is a conjunction of literals represented by a set of literals $\mathtt{L}$, then $\mathtt{R}$ is applicable on $\mathtt{Source}$ if $\mathtt{left} \subseteq \mathtt{L}$. In this situation:
$$\mathtt{R}(\mathtt{Source}) = \mathtt{R}(\mathtt{L}) = (\mathtt{L} \setminus \mathtt{left}) \cup \mathtt{right}$$
- If $\mathtt{Source}$ is in DNF, such that $\mathtt{Source} = \bigvee_i \mathtt{L}_i$, where the $\mathtt{L}_i$'s are conjunctions of literals, $\mathtt{R}$ is applicable on $\mathtt{Source}$ if it is applicable on at least one $\mathtt{L}_i$. Then:
$$\mathtt{R}(\mathtt{Source}) = \bigvee_i \begin{cases} \mathtt{R}(\mathtt{L}_i) & \text{if } \mathtt{R} \text{ is applicable to } \mathtt{L}_i, \\ \mathtt{L}_i & \text{otherwise.} \end{cases}$$
- If $\mathtt{Source}$ is not in DNF, it is then replaced by an equivalent formula in DNF.

Given an adaptation rule $\mathtt{R} = \mathtt{left} \rightsquigarrow \mathtt{right}$, $\mathtt{repairs}(\mathtt{R}) = \mathtt{left} \setminus \mathtt{right}$, that is the set of literals whose presence is necessary to apply $\mathtt{R}$, and that are effectively removed by application of $\mathtt{R}$. Every adaptation rule must be such that $\mathtt{repairs}(\mathtt{R}) \neq \varnothing$.

3.3 Combining Rule-based and Revision-based Adaptation

Let us consider the following distance on $\mathcal{U} = \mathbb{B}^n$ (for $x, y \in \mathcal{U}$):

$$\delta_{\mathtt{AK}}(x, y) = \inf\{\mathtt{cost}(p) \mid p\text{: adaptation path from } x \text{ to } y \text{ based on rules from } \mathtt{AK}\}$$

(x and y are interpretations that are assimilated to conjunctions of literals). By convention, $\inf \varnothing = +\infty$ and thus, if there is no adaptation path relating x to y, then $\delta_{\mathtt{AK}}(x, y) = +\infty$ and vice-versa. Otherwise, it can be shown that the infimum is always reached and so, $\delta_{\mathtt{AK}}(x, y) = 0$ iff $x = y$ (corresponding to the empty adaptation path).

It has been shown [Cojan and Lieber, 2012] that rule-based adaptation can be simulated by revision-based adaptation with no domain knowledge (i.e., DK is a tautology) and with the $\dotplus^{\delta_{\mathtt{AK}}}$ revision operator. When rule-based adaptation fails (no adaptation path from `Source` to `Target`), $\dotplus^{\delta_{\mathtt{AK}}}$-adaptation gives `CompletedTarget` equivalent to `Target` (no added information).

The failure of rule-based adaptation is due to the fact that no adaptation rules can be composed in order to solve the adaptation problem. In order to have an adaptation that always provide a result, the idea is to add $2n$ adaptation rules, one for each literal: given a literal ℓ, the flip of the literal ℓ is the adaptation rule $\mathtt{F}_\ell = \ell \rightsquigarrow \top$, where $\top$ is the empty conjunction of literals. It can be noticed that a cost is associated to each literal flip and that it may be the case that $\mathtt{cost}(\mathtt{F}_{\neg\ell}) \neq \mathtt{cost}(\mathtt{F}_\ell)$.

Now, let $d_{\mathtt{AK}}$ be the distance on $\mathcal{U} = \mathbb{B}^n$ defined, for $x, y \in \mathcal{U}$, by

$$d_{\mathtt{AK}}(x, y) = \inf\left\{\mathtt{cost}(p) \;\middle|\; \begin{array}{c} p\text{: adaptation path from } x \text{ to } y \\ \text{based on rules from } \mathtt{AK} \text{ and on flips of literals} \end{array}\right\}$$

Let $\dotplus_{\mathtt{AK}} = \dotplus^{d_{\mathtt{AK}}}$. $\dotplus_{\mathtt{AK}}$-adaptation is a revision-based adaptation using the adaptation rules, thus combining rule-based adaptation and revision-based adaptation, and that can take into account domain knowledge.

It can be noticed that if $\mathtt{AK} = \varnothing$ and $\mathtt{cost}(\mathtt{F}_\ell) = 1$ for every literal ℓ then $d_{\mathtt{AK}} = d_H$. Moreover the following assumption is made:

$$\text{For any } \mathtt{R} \in \mathtt{AK},\ \mathtt{cost}(\mathtt{R}) \leq \sum_{\ell \in \mathtt{repairs}(\mathtt{R})} \mathtt{cost}(\mathtt{F}_\ell) \tag{3.1}$$

In fact, this assumption does not involve any loss of generality. If an adaptation rule violates (3.1), then it does not appear in any optimal adaptation path: applying the flips of all the literals of its repair part will be less costly then applying the rule itself, and applying a flip never generates new clashes. However, it is always possible that a flip is used despite the existence of lower cost rules: indeed, the application of a rule can generate clashes by introducing new literals in the formula, and the subsequent repair costs could exceed those of using flips.

Chapter 4

Revision and Adaptation as Search Problems

While computing d_H between two interpretations is straightforward (and can be done in time proportional to the number of variables n by counting differences between the interpretations), the computation of $d_{\mathtt{AK}}$ poses a more complex problem. Indeed, it is necessary to determine which rules will appear in an optimal adaptation path, and in which order.

This Chapter presents how graph search algorithms can be used to efficiently implement revision and adaptation operators. Firstly, basics of graph theory and graph searching are introduced in Section 4.1. A naive graph modeling of revision problems, a second graph modeling based on implicant revision and a final modeling of the previously-presented adaptation approach are presented in Section 4.2. Finally, an efficient search algorithm to compute adaptation will be explained in Section 4.3.

4.1 Graph Theory and Search Problems

Graphs are structures representing a set of objects in which pairs of elements can be connected by links. Each element is represented by a vertex, and links are represented by edges. Edges are expressed as pairs of vertices. These pairs of vertices can be considered to be ordered (to represent an asymmetric link), in which case the graph is said to be directed. Graphs are commonly used to represent state spaces: each vertex represents a possible state of the world, while edges represents possible transitions between those states. Edges are usually given an arbitrary cost, that is the cost of transitioning from one state to another by following this edge.

Graph search algorithms are used to solve search problems in state spaces. A problem is represented by one or several initial states[1], and a criteria for a state to be considered final. Solutions to a problem are presented as paths, that is sequences of edges, that from the initial state leads to one final state. To each path can be associated a cost that is the sum of the costs of every edge. A solution is said to be *optimal* when it is one of the least costly solutions.

[1]Solving a problem with several initial states, represented by the set $\mathtt{S}_0$, is equivalent to solving this problem with only one initial state, and null-cost transitions from this initial state to each state of $\mathtt{S}_0$.

4.2 Revision and Adaptation

The set of propositional logic interpretations, $\mathbb{B}^n$, is the set of possible states of the world. Belief revision for an operator $\dot{+}^d$ can be realized as a search problem in state space: states are interpretations, and a transition from a state $\mathtt{S}_x$ representing an interpretation x and a state $\mathtt{S}_y$ representing an interpretation y has a cost of $d(x, y)$. $\psi \dot{+}^d \mu$ is then computed by first enumerating ψ and μ models that will determine, respectively, the initial and final states, then finding the shortest paths between them. Models of μ that are linked to models of ψ by a shortest path are then set as the models $\psi \dot{+}^d \mu$.

Implicant-based Revision. The implicant-based revision algorithm described in Section 2.6 implementing $\dot{+}^{d_H}$ can too be realized as a search problem. However, states are no more elements of $\mathbb{B}^n$, but ordered pair of implicants or tableaux branches. Initial states are states whose first element is a branch of the tableau of ψ, and the second a branch of the tableau of μ. Possible transitions from one state to another are determined by flips, that is the removal of a literal from a branch of ψ to solve a clash. A state is final iff there is no clash between its branch from ψ and its branch from μ. Note that the use of flips require the graph to be directed: each flip remove a literal with a cost of 1, however there is no converse transition that adds literals. Because transitions can only remove clashes, there is no cycles, and there is no dead ends except for final states. Also, it is easy to estimate the cost of the optimal solution: that cost is equal to minimum number of clashes in an initial state.

Adaptation as a Search Problem. Intuitively, rule-based adaptation can be similarly realized as a search problem using adaptation rules between states. Thus, realizing the adaption operator defined in Section 3.3–$\dot{+}_{\mathtt{AK}}$–as a search problem consists in adding the adaptation rules to the implicant-based revision search problem. The search problem realizing $\dot{+}_{\mathtt{AK}}$ is much more complex than the implicant-based revision problem: because rules can generate clashes, finding a solution is not straightforward, and cycles can be generated. For the same reason, estimating the cost of the optimal solution is not trivial.

4.3 A* Algorithm

Given that computing $\dot{+}_{\mathtt{AK}}$ requires finding optimal adaptation paths, an optimal search algorithm must be used. An algorithm is optimal if it finds solutions with a minimal cost. A* is a heuristic-based best-first search algorithm [Pearl, 1984] that is optimal when using an admissible heuristic function. It is suited for searching state spaces where there are a finite number of transitions from a given state to its successor states and for which the cost of a path is additive (the cost of a path is the sum of the cost of the transitions it contains). Given a state $\mathtt{S}$, let $\mathcal{F}^*(\mathtt{S})$ be the minimum of the costs of the paths from an initial state to a final state that contain $\mathtt{S}$. $\mathcal{F}^*(\mathtt{S})$ is defined as:

$\mathcal{F}^*(\mathtt{S}) = \mathcal{G}^*(\mathtt{S}) + \mathcal{H}^*(\mathtt{S})$

where $\mathcal{G}^*(\mathtt{S})$ is the minimal cost of the paths from an initial state to $\mathtt{S}$

and $\mathcal{H}^*(\mathtt{S})$ is the minimal cost of the path from $\mathtt{S}$ to a final state.

Usually, as in the $\dotplus_{\mathtt{AK}}$ problem, $\mathcal{F}^*$ is unknown and is approximated by a function $\mathcal{F}$ defined as:

$$\mathcal{F}(\mathtt{S}) = \mathcal{G}(\mathtt{S}) + \mathcal{H}(\mathtt{S})$$

For the A* algorithm to be optimal, it is necessary that for any state $\mathtt{S}$, $\mathcal{G}$ and $\mathcal{H}$ verifies the following inequalities:

$$\mathcal{G}(\mathtt{S}) \geq \mathcal{G}^*(\mathtt{S})$$
$$\mathcal{H}(\mathtt{S}) \leq \mathcal{H}^*(\mathtt{S})$$

Usually, $\mathcal{G}(\mathtt{S})$ is the cost of the path that has already been generated for reaching $\mathtt{S}$ and thus, $\mathcal{G}(\mathtt{S}) \geq \mathcal{G}^*(\mathtt{S})$. Then, the main difficulty is to find an admissible $\mathcal{H}$, also called the *heuristic* function. The constant function 0 is an admissible $\mathcal{H}$ (it is then equivalent to run Dijkstra's algorithm [Dijkstra, 1959], that is also optimal) but the closer an admissible $\mathcal{H}$ is to $\mathcal{H}^*$, the faster the search is.

The A* algorithm consists in searching the state space, starting from the initial states, in ascending order of their $\mathcal{F}(\mathtt{S})$. Once A* reaches a final state, it is known that the path used to reach it is an optimal solution.

Space Complexity Issues. Given the total number of states that grows exponentially with the number of variables, it would be inefficient to generate every state at the start of the execution of the algorithm. Thus, states will not be generated until the search algorithm visits them. Because A* does not need to visit every state to find an optimal solution, this makes the algorithm effectively less memory expensive.

Chapter 5

Algorithm of Adaptation Based on Tableaux Repairs

The adaptation algorithm is based on the revision of the source case by the target case, both w.r.t. the domain knowledge. As such, the algorithm performs the revision of a formula ψ by a formula μ, with:

$$\psi = \texttt{DK} \wedge \texttt{Source} \qquad \mu = \texttt{DK} \wedge \texttt{Target}$$
$$\text{and so:} \qquad \texttt{CompletedTarget} = \psi \dotplus_{\texttt{AK}} \mu$$

Section 5.1 presents the inputs and outputs of the algorithm. Then a running example is introduced in Section 5.2, and its execution its detailed throughout this Chapter. Section 5.3 presents the algorithm, which uses an heuristic function $\mathcal{H}$, that is defined and proven to be admissible in Section 5.4. Section 5.5 details an example and Section 5.6 studies the termination and the complexity of the algorithm.

5.1 Inputs and Output

The inputs of the algorithm are:

- Two propositional formulae ψ and μ in NNF.
- A set AK of adaptation rules.
- A function `cost` that associates to each adaptation rule and each literal flip a positive real number.

The algorithm outputs $\psi \dotplus_{\texttt{AK}} \mu$ in DNF.

5.2 Introducing the Running Example

Let $\psi = a \wedge b$ and $\mu = \neg a \wedge \neg c$ be two propositional formulae. The computation of $\psi \dot{+}_{\mathtt{AK}} \mu$ will be detailed along with the presentation of the algorithm. For this purpose, the following adaptation knowledge AK will be used:

$$\begin{aligned}\mathtt{AK} &= \{\mathtt{R}_1, \mathtt{R}_2\}\\ \mathtt{R}_1 &= a \wedge b \rightsquigarrow b \wedge c\\ \mathtt{R}_2 &= b \wedge c \rightsquigarrow a \wedge b\end{aligned}$$

The following cost function will be used:

$\mathtt{cost}(\mathtt{R}_1) = 0.5$	$\mathtt{cost}(\mathtt{R}_2) = 0.5$
$\mathtt{cost}(\mathtt{F}_a) = 2$	$\mathtt{cost}(\mathtt{F}_{\neg a}) = 1$
$\mathtt{cost}(\mathtt{F}_b) = 1$	$\mathtt{cost}(\mathtt{F}_{\neg b}) = 1$
$\mathtt{cost}(\mathtt{F}_c) = 1$	$\mathtt{cost}(\mathtt{F}_{\neg c}) = 1$

5.3 Algorithm

The algorithm uses an A* search, where a state is an ordered pair $(\mathtt{L}, \mathtt{M})$ of individually consistent sets of literals. That is L is consistent, M is consistent, but $\mathtt{L} \cup \mathtt{M}$ may not be. The function branches, defined in Section 2.5, is used by the algorithm to generate the set of initial states. In order to abide by Katsuno and Mendelzon postulate of irrelevance to the syntax, the function min-branches introduced, that convert the output of the branches function into the set of prime implicants of the formula, represented as sets of literals. Indeed, equivalent formulae have the same set of prime implicants.

For any set of literals L, let $\mathtt{L}^{\neg} = \{\neg\ell \mid \ell \in \mathtt{L}\}$. For example, if $\mathtt{L} = \{a, \neg b, \neg c\}$ then $\mathtt{L}^{\neg} = \{\neg a, b, c\}$. L is said to be consistent if and only if $\mathtt{L} \cap \mathtt{L}^{\neg} = \varnothing$.

Given the propositional formulae ψ and μ, the set of the initial states is:

$$\mathtt{min\text{-}branches}(\psi) \times \mathtt{min\text{-}branches}(\mu)$$

Figure 5.1 describes the computation of the initial states for the running example. Note that, because ψ and μ are conjunctions of literals, the set of branches of their respective tableaux coincides with the set of their primes implicants.

A final state is a state $(\mathtt{L}, \mathtt{M})$ such that $\mathtt{L} \cap \mathtt{M}^{\neg} = \varnothing$ (which is equivalent to the fact that $\mathtt{L} \cup \mathtt{M}$ is consistent). If $(\mathtt{L}, \mathtt{M})$ is a non final state, then there exits $\ell \in \mathtt{L}$ such that $\ell \in \mathtt{L} \cap \mathtt{M}^{\neg}$: there is a clash on ℓ.

The transitions from a state to another state are defined as follows. There is a transition σ from the state $x = (\mathtt{L}_x, \mathtt{M}_x)$ to the state $y = (\mathtt{L}_y, \mathtt{M}_y)$ if $\mathtt{M}_x = \mathtt{M}_y$ and:

- there exists a literal $\ell \in \mathtt{L}_x$ such that $\mathtt{L}_y = \mathtt{F}_\ell(\mathtt{L}_x)$ or

$$\begin{aligned}
\texttt{branches}(\psi = a \wedge b) &= \{B_1 \cup B_2 \mid (B_1, B_2) \in \{\{a\}\} \times \{\{b\}\} \text{ and } B_1 \cup B_2 \text{ consistent.}\} \\
&= \{\{a\} \cup \{b\}\} \\
&= \{\{a, b\}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{branches}(\mu = \neg a \wedge \neg c) &= \{B_1 \cup B_2 \mid (B_1, B_2) \in \{\{\neg a\}\} \times \{\{\neg c\}\} \text{ and } B_1 \cup B_2 \text{ consistent.}\} \\
&= \{\{\neg a\} \cup \{\neg c\}\} \\
&= \{\{\neg a, \neg c\}\}
\end{aligned}$$

$$\texttt{min-branches}(\psi) \times \texttt{min-branches}(\mu) = \{(\{a, b\}, \{\neg a, \neg c\})\}$$

Figure 5.1: Computing the initial states of the running example.

- there exists an adaptation rule $\mathtt{R}$ such that $\mathtt{R}$ is applicable on $\mathtt{L}_x$ and $\mathtt{L}_y = \mathtt{R}(\mathtt{L}_x)$.

The cost of a transition σ is the cost of the rule (literal flip or adaptation rule) it uses.

Using a slight variant of the A* algorithm, it is possible to determine the set of least costly transition sequences leading to a final state of the problem. That is, the algorithm does not stop after finding the first optimal solution, but after finding every other optimal solutions (i.e., the solutions with the same minimal cost). The detailed algorithm is presented in Algorithm 1 and is explained hereafter.

The algorithm first creates the initial states (line 3). For every initial state $\mathtt{S}_0$, $\mathcal{G}(\mathtt{S}_0) = 0$, that is the cost of reaching $\mathtt{S}_0$ is 0. In the case no value of $\mathcal{G}$ is set for a state $\mathtt{S}$, $\mathcal{G}(\mathtt{S})$ evaluates to $+\infty$, meaning there is no known path from an initial state to $\mathtt{S}$. The algorithm then finds a state $\mathtt{S}_c$ minimizing $\mathcal{F}(\mathtt{S}_c) = \mathcal{G}(\mathtt{S}_c) + \mathcal{H}(\mathtt{S}_c)$, that is the cost of reaching $\mathtt{S}_c$ from an initial state plus the heuristic cost of reaching a final state from $\mathtt{S}_c$ (line 10). For each rule or flip σ that can be applied on $\mathtt{S}_c$, it creates the state $\mathtt{S}_d$ such that $\mathtt{S}_c \xrightarrow{\sigma} \mathtt{S}_d$. If $\mathcal{G}(\mathtt{S}_d) > \mathcal{G}(\mathtt{S}_c) + \mathtt{cost}(\sigma)$, that is there is no known less or equally costly path to $\mathtt{S}_d$, then $\mathcal{G}(\mathtt{S}_d)$ is set to $\mathcal{G}(\mathtt{S}_c) + \mathtt{cost}(\sigma)$. Each generated $\mathtt{S}_d$ is added to the set of states to explore, while $\mathtt{S}_c$ is removed from it (lines 15 to 20). The algorithm repeats the previous step until it finds a state $\mathtt{S}_f$ minimizing $\mathcal{F}(\mathtt{S}_f)$ and that is a final state (lines 11 to 13). From this point, the algorithm carries on but ignores any state $\mathtt{S}_c$ such that $\mathcal{F}(\mathtt{S}_c) > \mathcal{F}(\mathtt{S}_f)$, which cannot lead to a less costly solution. It stops when there is no more states that can lead to a less costly solution (line 9). Finally, all final states $\mathtt{S}_f$ minimizing $\mathcal{F}(\mathtt{S}_f)$ are returned in the form of tableaux branches (line 22), that is for each state $\mathtt{S}_f$ defined by $(\mathtt{L}_f, \mathtt{M}_f)$, the branch containing the literals $\mathtt{L}_f \cup \mathtt{M}_f$ is returned.

```
1 revise($\psi, \mu, \mathtt{AK}, \mathtt{cost}$)
  Input: $\psi$ and $\mu$, two propositional formulae in NNF. $\psi$ has to be revised by $\mu$.
         AK, the set of adaptations rules to perform the adaptation operation.
         cost, a function that associates to every literal flip and adaptation rule a positive
  real number.
  Output: $\psi \dot{+}_{\mathtt{AK}} \mu$ in DNF
2 begin
      // open-states is initiated by initial states with cost 0.
3     open-states $\leftarrow$ min-branches($\psi$) $\times$ min-branches($\mu$)
4     $\mathcal{G}(\mathtt{S})$ evaluates to $+\infty$ by default
5     $\mathcal{G}(\mathtt{S}) \leftarrow 0$ for each $\mathtt{S} \in$ open-states
6     $\mathcal{F}(\mathtt{S})$ evaluates to $\mathcal{G}(\mathtt{S}) + \mathcal{H}(\mathtt{S})$
7     Solutions $\leftarrow \varnothing$
8     solutionCost $\leftarrow +\infty$
9     while $\{\mathtt{S} \mid \mathtt{S} \in$ open-states$, \mathcal{F}(\mathtt{S}) \leq$ solutionCost$\} \neq \varnothing$ do
          // $(\mathtt{L}_c, \mathtt{M}_c)$ is the current state.
10        $(\mathtt{L}_c, \mathtt{M}_c) \leftarrow$ a $\mathtt{S} \in$ open-states minimizing $\mathcal{F}(\mathtt{S})$
11        if $(\mathtt{L}_c \cap \mathtt{M}_c^{\neg}) = \varnothing$ then
12            Solutions $\leftarrow$ Solutions $\cup \{\mathtt{L}_c \cup \mathtt{M}_c\}$
13            solutionCost $\leftarrow \mathcal{F}((\mathtt{L}_c, \mathtt{M}_c))$
14        else
15            for each $\sigma \in \mathtt{AK} \cup \{\mathtt{F}_\ell$ for every literal $\ell\}$ such that $\sigma$ is applicable on $\mathtt{L}_c$ do
16                open-states $\leftarrow$ open-states $\cup \{(\sigma(\mathtt{L}_c),\ \mathtt{M}_c)\}$
17                $\mathcal{G}((\sigma(\mathtt{L}_c),\ \mathtt{M}_c)) \leftarrow \min(\mathcal{G}((\sigma(\mathtt{L}_c),\ \mathtt{M}_c)), \mathcal{G}((\mathtt{L}_c, \mathtt{M}_c)) + \mathtt{cost}(\sigma))$
18            end
19        end
20        open-states $\leftarrow$ open-states $\setminus \{(\mathtt{L}_c, \mathtt{M}_c)\}$
21    end
22    return Solutions
23 end
```
Algorithm 1: Algorithm of adaptation based on tableaux repairs.

5.4 Heuristics

The function $\mathcal{H}$ used in the algorithm is defined by: for $\mathtt{S} = (\mathtt{L}, \mathtt{M}), \mathcal{H}(\mathtt{S}) = \mathtt{ERC}(\mathtt{L} \cap \mathtt{M}^{\neg})$ where ERC is defined as follows (ERC stands for Estimated Repair Cost):

$$\mathtt{ERC}(\{\ell\}) = \min\{\mathtt{cost}(\mathtt{F}_\ell)\} \cup \left\{ \frac{\mathtt{cost}(\mathtt{R})}{|\mathtt{repairs}(\mathtt{R})|} \;\middle|\; \begin{array}{c} \mathtt{R} \in \mathtt{AK}, \text{ such that} \\ \ell \in \mathtt{repairs}(\mathtt{R}) \end{array} \right\} \qquad \text{for any literal } \ell$$

$$\mathtt{ERC}(\mathtt{L}) = \sum_{\ell \in \mathtt{L}} \mathtt{ERC}(\{\ell\}) \qquad \text{for any set of literals } \mathtt{L}$$

Proposition. *$\mathcal{H}$ is admissible.*

Proof. A heuristic $\mathcal{H}$ is consistent if it verifies $\mathcal{H}(\mathtt{S}_x) \leq c(\mathtt{S}_x, \mathtt{S}_y) + \mathcal{H}(\mathtt{S}_y)$, where $c(\mathtt{S}_x, \mathtt{S}_y)$ is the minimal cost for paths from $\mathtt{S}_x$ to $\mathtt{S}_y$. A consistent heuristic is also admissible [Pearl, 1984]. The consistency of $\mathcal{H}$ can be inductively proven, as shown below. In this proof the notation $X \setminus Y$ is used, where X and Y are two sets of literals. $X \setminus Y = X \cap \overline{Y}$, where $\overline{Y} = \{\ell \mid \ell \in \mathcal{V} \cup \mathcal{V}^\neg, \ell \notin Y\}$.
Base Case. Let σ be a one-step transition from $\mathtt{S}_x = (\mathtt{L}_x, \mathtt{M})$ to $\mathtt{S}_y = (\mathtt{L}_y, \mathtt{M})$: $\mathtt{S}_x \xrightarrow{\sigma} \mathtt{S}_y$. The goal of the base case proof is to show that $\mathcal{H}(\mathtt{S}_x) \leq c(\mathtt{S}_x, \mathtt{S}_y) + \mathcal{H}(\mathtt{S}_y)$, where $c(\mathtt{S}_x, \mathtt{S}_y) = \mathtt{cost}(\sigma)$. Two situations are considered:

- σ is a literal flip $\mathtt{F}_\ell$. In this case $\ell \in (\mathtt{L}_x \cap \mathtt{M}^\neg)$ and $\mathtt{L}_y = \mathtt{L}_x \setminus \{\ell\}$. Thus:

$$\mathcal{H}(\mathtt{S}_y) = \mathtt{ERC}((\mathtt{L}_x \setminus \{\ell\}) \cap \mathtt{M}^\neg) = \mathtt{ERC}((\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus \{\ell\})$$

  Because $\ell \in (\mathtt{L}_x \cap \mathtt{M}^\neg)$:

$$\mathcal{H}(\mathtt{S}_y) = \mathtt{ERC}((\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus \{\ell\}) \geq \mathtt{ERC}(\mathtt{L}_x \cap \mathtt{M}^\neg) - \mathtt{ERC}(\{\ell\})$$

  Then, $\mathcal{H}(\mathtt{S}_x) \leq \mathcal{H}(\mathtt{S}_y) + \mathtt{ERC}(\{\ell\})$ which entails that $\mathcal{H}(\mathtt{S}_x) \leq \mathtt{cost}(\mathtt{F}_\ell) + \mathcal{H}(\mathtt{S}_y)$.

- σ is an adaptation rule $\mathtt{R} = \mathtt{left} \rightsquigarrow \mathtt{right}$, with: $\mathtt{left} \subseteq \mathtt{L}_x$, since $\mathtt{R}$ is applicable on $\mathtt{S}_x$; $\mathtt{repairs}(\mathtt{R}) = \mathtt{left} \setminus \mathtt{right} \neq \varnothing$; and $\mathtt{L}_y = (\mathtt{L}_x \setminus \mathtt{left}) \cup \mathtt{right}$.
  $\mathcal{H}(S_x) - \mathcal{H}(S_y) = \mathtt{ERC}(\mathtt{L}_x \cap \mathtt{M}^\neg) - \mathtt{ERC}(\mathtt{L}_y \cap \mathtt{M}^\neg) \leq \mathtt{ERC}((\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus (\mathtt{L}_y \cap \mathtt{M}^\neg))$.
  $(\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus (\mathtt{L}_y \cap \mathtt{M}^\neg) \subseteq \mathtt{repairs}(\mathtt{R})$ can be proven as shown below:

$$\begin{aligned}
(\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus (\mathtt{L}_y \cap \mathtt{M}^\neg) &= (\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus ((\mathtt{L}_x \setminus \mathtt{left}) \cup \mathtt{right} \cap \mathtt{M}^\neg) \text{ by def. of } \mathtt{L}_y \\
&= \mathtt{L}_x \cap \mathtt{M}^\neg \cap \overline{((\mathtt{L}_x \cap \overline{\mathtt{left}}) \cup \mathtt{right}) \cap \mathtt{M}^\neg} \\
&= \mathtt{L}_x \cap \mathtt{M}^\neg \cap (((\overline{\mathtt{L}_x} \cup \mathtt{left}) \cap \overline{\mathtt{right}}) \cup \overline{\mathtt{M}^\neg}) \\
&= (\mathtt{L}_x \cap \mathtt{M}^\neg \cap ((\overline{\mathtt{L}_x} \cup \mathtt{left}) \cap \overline{\mathtt{right}})) \cup \underbrace{(\mathtt{L}_x \cap \mathtt{M}^\neg \cap \overline{\mathtt{M}^\neg})}_{\varnothing} \\
&= \underbrace{(\mathtt{L}_x \cap \mathtt{M}^\neg \cap \overline{\mathtt{L}_x} \cap \overline{\mathtt{right}})}_{\varnothing} \cup (\mathtt{L}_x \cap \mathtt{M}^\neg \cap \mathtt{left} \cap \overline{\mathtt{right}}) \\
&= \mathtt{L}_x \cap \mathtt{M}^\neg \cap (\mathtt{left} \setminus \mathtt{right}) \\
&= \mathtt{L}_x \cap \mathtt{M}^\neg \cap \mathtt{repairs}(\mathtt{R}) \subseteq \mathtt{repairs}(\mathtt{R})
\end{aligned}$$

  Thus, $\mathcal{H}(S_x) - \mathcal{H}(S_y) \leq \mathtt{ERC}((\mathtt{L}_x \cap \mathtt{M}^\neg) \setminus (\mathtt{L}_y \cap \mathtt{M}^\neg)) \leq \mathtt{ERC}(\mathtt{repairs}(\mathtt{R}))$, and $\mathcal{H}(S_x) \leq \mathtt{ERC}(\mathtt{repairs}(\mathtt{R})) + \mathcal{H}(S_y)$. Therefore it is sufficient to prove that $\mathtt{ERC}(\mathtt{repairs}(\mathtt{R})) \leq \mathtt{cost}(\mathtt{R})$, which is proven below.

  For $\ell \in \mathtt{repairs}(\mathtt{R})$, $\mathtt{ERC}(\{\ell\}) \leq \dfrac{\mathtt{cost}(\mathtt{R})}{|\mathtt{repairs}(\mathtt{R})|}$ according to the definition of $\mathtt{ERC}$. Therefore:

$$\begin{aligned}
\mathtt{ERC}(\mathtt{repairs}(\mathtt{R})) &= \sum_{\ell \in \mathtt{repairs}(\mathtt{R})} \mathtt{ERC}(\{\ell\}) \\
&\leq \sum_{\ell \in \mathtt{repairs}(\mathtt{R})} \frac{\mathtt{cost}(\mathtt{R})}{|\mathtt{repairs}(\mathtt{R})|} = \mathtt{cost}(\mathtt{R})
\end{aligned}$$

So, in any case, if $\mathtt{S}_x \xrightarrow{\sigma} \mathtt{S}_y$, $\mathcal{H}(\mathtt{S}_x) \leq c(\mathtt{S}_x, \mathtt{S}_y) + \mathcal{H}(\mathtt{S}_y)$, which proves the base case.
Inductive Step. It is assumed that the property is true for a given $n \geq 1$:

$$\text{For any path of length } n\text{, from } \mathtt{S} \text{ to } \mathtt{T},\ \mathcal{H}(\mathtt{S}) \leq c(\mathtt{S}, \mathtt{T}) + \mathcal{H}(\mathtt{T}) \tag{5.1}$$

Now, let $\mathtt{S}_x = \mathtt{S}^0 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_n} \mathtt{S}^n \xrightarrow{\sigma_{n+1}} \mathtt{S}^{n+1} = \mathtt{S}_y$ be a path of length $n+1$. $\mathtt{S}_x \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_n} \mathtt{S}^n$ is a path of length n, so, according to (5.1):

$$\mathcal{H}(\mathtt{S}_x) \leq c(\mathtt{S}_x, \mathtt{S}^n) + \mathcal{H}(\mathtt{S}^n) \tag{5.2}$$

$\mathtt{S}^n \xrightarrow{\sigma_{n+1}} \mathtt{S}_y$ is a path of length 1, so, according to the base case:

$$\mathcal{H}(\mathtt{S}^n) \leq c(\mathtt{S}^n, \mathtt{S}_y) + \mathcal{H}(\mathtt{S}_y) \tag{5.3}$$

Finally, since c is additive:

$$c(\mathtt{S}_x, \mathtt{S}_y) = c(\mathtt{S}_x, \mathtt{S}^n) + c(\mathtt{S}^n, \mathtt{S}_y) \tag{5.4}$$

From (5.2), (5.3) and (5.4) it comes that $\mathcal{H}(\mathtt{S}_x) \leq c(\mathtt{S}_x, \mathtt{S}_y) + \mathcal{H}(\mathtt{S}_y)$ for any path from $\mathtt{S}_x$ to $\mathtt{S}_y$ of length $n+1$. Therefore, the proposition is proven.

$\square$

5.5 Execution on the Running Example

This section details the execution of the algorithm for the running example defined in Section 5.2:

$\psi = a \wedge b$	$\mathtt{AK} = \{\mathtt{R}_1, \mathtt{R}_2\}$	$\mathtt{cost}(\mathtt{R}_1) = \mathtt{cost}(\mathtt{R}_2) = 0.5$
$\mu = \neg a \wedge \neg c$	$\mathtt{R}_1 = a \wedge b \rightsquigarrow b \wedge c$	$\mathtt{cost}(\mathtt{F}_\ell) = 1$ for $\ell \neq a$
	$\mathtt{R}_2 = b \wedge c \rightsquigarrow a \wedge b$	$\mathtt{cost}(\mathtt{F}_a) = 2$

As computed previously, $\mathtt{min\text{-}branches}(\psi) = \{\{a, b\}\}$ and $\mathtt{min\text{-}branches}(\mu) = \{\{\neg a, \neg c\}\}$. Figure 5.2 shows the different branches developed by the algorithm. At the initial state $\mathtt{S}_0$, the cost is 0. The algorithm repairs the clash on the variable a, for a minimal cost of 0.5. Two branches are developed, the first using a flip on a and the second using the rule R_1. The state $\mathtt{S}_1$ is a final state with a cost of 2. The variable `solutionCost` is set to that cost of 2. That cost is defined as the best final cost. The state $\mathtt{S}_2$ has a clash on the variable c. To repair that clash, the minimal cost is 0.5. Adding the cost of that state, the cost is lower than `solutionCost`. Thus, the clash is repaired to possibly find the solution with a lower final cost. To repair the clash on the variable c, the rule R_2 and the flip $\mathtt{F}_c$ are used. The use of $\mathtt{F}_c$ leads to a final state $\mathtt{S}_3$ with a final cost of 1.5. That cost becomes the best final cost. The use of R_2 gets back to a state $\mathtt{S}_4$ equivalent to the initial state $\mathtt{S}_0$ but with a cost of 1. To repair the clash on the variable a, the minimal cost is 0.5. As the cost to reach that state plus the cost to repair that clash is equal to `solutionCost`, the clash on the variable a is again repaired. The state $\mathtt{S}_5$ is reached using $\mathtt{F}_a$. That final state is not a best solution because the cost of that state is higher than `solutionCost`. The state $\mathtt{S}_6$ is not final but that branch is not developed because the cost to reach that state plus the cost to repair the clash on the variable c is higher than `solutionCost`. The algorithm returns the solution $L_3 \cup M_3$, i.e. $\{b, \neg a, \neg c\}$.

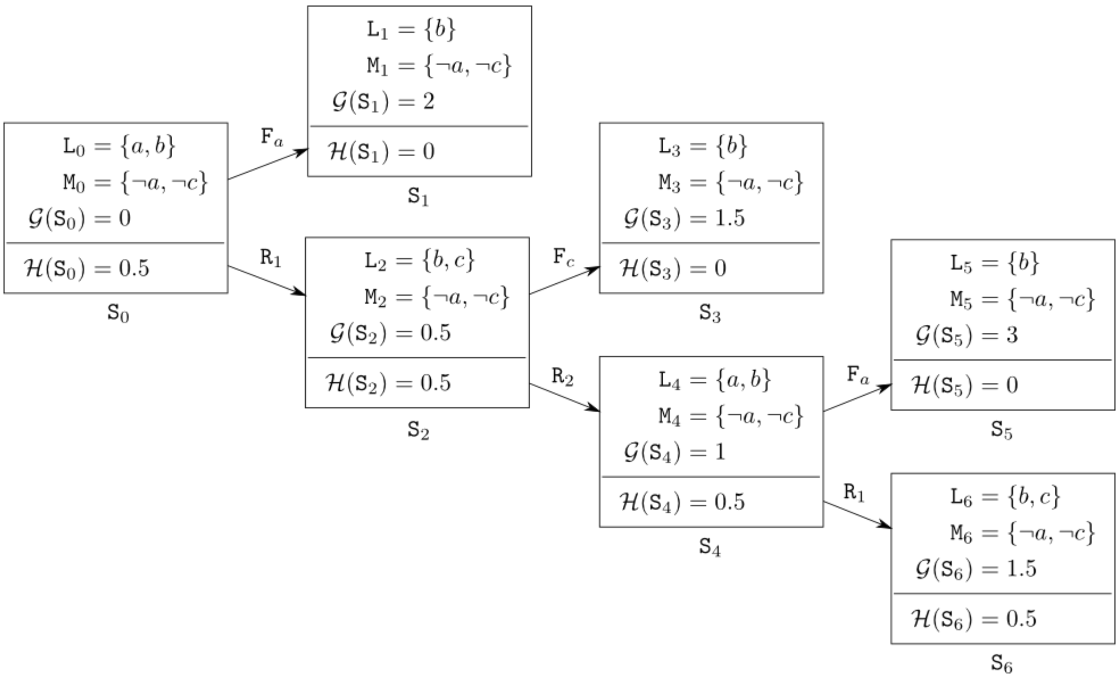


Figure 5.2: Application of the algorithm on the running example.

5.6 Termination and Complexity

In this section, the termination and complexity of the algorithm are studied when no heuristics is used ($\mathcal{H} = 0$). Since the heuristics introduced in Section 5.4 is admissible, the complexity without heuristics is an upper bound for the complexity with heuristics.

Proposition. *The previously presented adaptation algorithm always terminates. Its complexity in the worst case is in $O(4^n \times t^{\varrho+1} \times (n + \varrho \log(t)))$, where n is the number of variables, $t = |\mathit{AK}| + |\mathcal{V}|$, and ϱ is the sum of each flip cost divided by the minimum transition cost.*

According to [Eiter and Gottlob, 1992], the complexity of a revision operator is $\mathsf{P}^{\mathsf{NP}[O(\log n)]}$-hard[1] hence this high worst-case complexity of the algorithm is not a surprise.

Proof. Let $\mathtt{G}$ be the graph generated by the algorithm: it is a directed acyclic labeled graph (that corresponds to the union of exploration trees from each initial state), whose vertices are labeled by states, and edges are labeled by transitions: if (v, w) is an edge of $\mathtt{G}$, then $\mathtt{S}_x \xrightarrow{\sigma} \mathtt{S}_y$ is a transition where $\mathtt{S}_x = \lambda(v)$, $\mathtt{S}_y = \lambda(w)$ and $\sigma = \lambda((v, w))$ (λ is the label

[1]More precisely, the complexity of the Dalal revision operator $\dotplus_{\text{Dalal}}$ is $\mathsf{P}^{\mathsf{NP}[O(\log n)]}$-complete and the presented algorithm can be used for computing $\dotplus_{\text{Dalal}}$: $\dotplus_{\text{Dalal}} = \dotplus_{\mathtt{AK}}$ with $\mathtt{AK} = \varnothing$ and $\mathtt{cost}(\mathtt{F}_\ell) = 1$ for each literal ℓ.

function associated with $\mathtt{G}$). The roots of $\mathtt{G}$ (vertices of rank 0) correspond to the s_0 initial states. Let s_k be the numbers of vertices of rank k. s_k verifies $s_k \leq s_0 t^k$, where $t = n + |\mathtt{AK}|$ is an upper bound of the number of transitions from a given state $\mathtt{S}$ (recall that there are n propositional variables and that if a flip $\mathtt{F}_\ell$ is applicable on $\mathtt{S}$, then $\mathtt{F}_{\neg\ell}$ is not). In order to prove the termination of the algorithm, it is sufficient to prove that $\mathtt{G}$ is finite: each step of the loop is executed in finite time and each node of $\mathtt{G}$ is considered exactly once.

There exists a solution, i.e. a path p from a vertex labeled by an initial state to another one labeled by a final state. Indeed, let $(\mathtt{L}_0, \mathtt{M})$ be an initial state. Then, the application of all the flips $\mathtt{F}_\ell$ for $\ell \in (\mathtt{L}_0 \cap \mathtt{M}^\neg) \subseteq \mathtt{L}_0$ leads to a state $(\mathtt{L}_0 \setminus (\mathtt{L}_0 \cap \mathtt{M}^\neg), \mathtt{M}) = (\mathtt{L}_0 \cap \overline{\mathtt{M}^\neg}, \mathtt{M})$ which is consistent because $\mathtt{L}_0 \cap \overline{\mathtt{M}^\neg} \cap \mathtt{M}^\neg = \varnothing$. Let $\mathcal{C}_\mathtt{F} = \mathtt{cost}(p)$.

This involves that there exists (at least) one optimal solution p^*: the set $\{p \mid p \text{ is a path with } \mathtt{cost}(p) \leq \mathcal{C}_\mathtt{F}\}$ is finite, so its infimum is reached. Let $\mathcal{C}^* = \mathtt{cost}(p^*)$ and v^* be the final vertex of p^*. It can be proven that the rank of v^*, r^*, verifies:

$$r^* \leq \left\lfloor \frac{\mathcal{C}^*}{\mathtt{mc}} \right\rfloor \leq \left\lfloor \frac{\mathcal{C}_\mathtt{F}}{\mathtt{mc}} \right\rfloor \leq \left\lfloor \frac{\mathtt{fc}}{\mathtt{mc}} \right\rfloor = \varrho \tag{5.5}$$

$$\text{where } \mathtt{mc} = \min\{\mathtt{cost}(\sigma) \mid \sigma \in \mathtt{AK} \cup \{\mathtt{F}_\ell \text{ for every literal } \ell\}\},$$

$$\mathtt{fc} = \sum_{\ell \in \mathcal{V} \cup \mathcal{V}^\neg} \mathtt{cost}(\mathtt{F}_\ell), \text{ and } \lfloor x \rfloor \text{ is the integer part of } x$$

Indeed, $\mathcal{C}^* = \sum_{i=1}^{r^*} \mathtt{cost}(\sigma_i)$, where σ_i is the i^{th} transition of p^*. Therefore, $\mathcal{C}^* \geq r^* \times \mathtt{mc}$ and then, $r^* \leq \frac{\mathcal{C}^*}{\mathtt{mc}} \leq \varrho$. Since r^* is an integer, inequation (5.5) follows.

Therefore, each optimal solution has a length that is less or equal to $\left\lfloor \frac{\mathcal{C}^*}{\mathtt{mc}} \right\rfloor$. Since the graph $\mathtt{G}$ is searched by increasing cost $\mathcal{G}(\mathtt{S})$ (admitting $\mathcal{H} = 0$), every vertex v of $\mathtt{G}$ has a rank lower to this constant. This involves that $\mathtt{G}$ is finite and so, the algorithm terminates. Moreover, this involves that the number of vertices $N_\mathtt{G}$ verifies:

$$N_\mathtt{G} \leq \sum_{k=0}^{r^*} s_k \leq \sum_{k=0}^{r^*} s_0 t^k = s_0 \frac{t^{r^*+1} - 1}{t - 1} \leq s_0 t^{r^*+1} \leq s_0 t^{\varrho+1}$$

which gives an upper bound for the number of iterations of the loop in the algorithm.

The count of rank 0 vertices, s_0, is bounded by $|\mathtt{min\text{-}branches}(\psi)| \times |\mathtt{min\text{-}branches}(\mu)|$, which in the worst case is no greater than $2^n \times 2^n = 4^n$. It is assumed that each vertex v is stored on a sorted structure using an ordering on $\mathcal{F}(\lambda(v))$. As such, finding the state $\mathtt{S}$ minimizing $\mathcal{F}(\mathtt{S})$ is done in constant time, while creating a state is done in logarithmic time w.r.t. the size of the structure, in order of $N_\mathtt{G}$. Thus, the complexity of the algorithm is in $O(N_\mathtt{G} \log(N_\mathtt{G}))$, and $N_\mathtt{G}$ is of order $O(4^n \times t^{\varrho+1})$. More precisely, the complexity of the algorithm is in $O(4^n \times t^{\varrho+1} \times (n + \varrho \log(t)))$. $\square$

Chapter 6

Implementation

One of the tasks of this internship was to implement the adaptation operator presented in this report. REVISOR [Cojan et al., 2013] is a Java software library providing several revision engines for different formalisms, available for download on the site `http://revisor.loria.fr`. Its first version contained three revision engines:

- REVISOR/CLC, that performs revision on Conjunctions of Linear Constraints [Blansché et al., 2010]. REVISOR/CLC has been implemented by Julien Cojan, Cynthia Florentin and Sophie Pierrat.
- REVISOR/PL, that performs implicant-based revision on Propositional Logic formulae.
- REVISOR/QA, that performs revision in Qualitative Algebra [Dufour-Lussier et al., 2010]. REVISOR/QA has been implemented by Valmi Dufour-Lussier.

In its second version, REVISOR introduces a new revision engine, REVISOR/PLAK, that performs rule-based and revision-based adaptation on Propositional Logic formulae [Personeni et al., 2013], as defined previously in this report. Java version 7 is required to use the REVISOR engines. REVISOR/CLC and REVISOR/QA also require respectively additional libraries and a Perl environment in order to operate properly.

Section 6.1 presents the REVISOR/PL revision engine, then Section 6.2 presents REVISOR/PLAK. For both revision engines, a detailed example is presented. Section 6.3 provides an evaluation of REVISOR/PLAK computing time and explain its results.

6.1 REVISOR/PL

REVISOR/PL is a revision engine in the Propositional Logic formalism: it implements an implicant-based revision operator using semantic tableaux as defined in Section 2.6. It can also perform $\dot{+}$-adaptation using that revision operator. Figure 6.1 presents an example in Java code of how REVISOR/PL can be used to compute adaptation of propositional cases. Note that in REVISOR/PL, the $\wedge$, $\vee$, $\neg$ and $\Rightarrow$ connectives are respectively replaced by `&`, `|`, `!` and `>`. The code presented solves the following adaptation problem:

$$\begin{aligned}\texttt{Source} &= \text{pie} \land \text{apple} \land \text{pie\_shell} \land \text{sugar}\\ \texttt{Target} &= \text{pie} \land \neg\text{apple}\\ \texttt{DK} &= (\text{pear} \lor \text{apple} \Rightarrow \text{fruit}) \land (\text{fruit} \Rightarrow \text{pear} \lor \text{apple})\end{aligned}$$

That is, REVISOR/PL has to produce a pie recipe without apple, through minimal modifications of an apple pie recipe. To this end, it uses the domain knowledge that apples and pears are fruits and, conversely, the only available fruits are apples and pears.

```
// Set the cost of variable flip to 1
RevisorPL.resetWeights();
// Initialize the source and target cases, and the domain knowledge
PLFormula source = RevisorPL.parseFormula("pie & apple & pie_shell &
sugar");
PLFormula target = RevisorPL.parseFormula("pie & !apple");
PLFormula dk = RevisorPL.parseFormula("(pear | apple > fruit) &
(fruit > pear | apple)");
// Compute the adaptation of the source case to solve the target case
PLFormula result = RevisorPL.adapt(source, target, dk);
RevisorPL.print(result);
```

Figure 6.1: Adapting an apple pie recipe with REVISOR/PL.

The code in figure 6.1 gives the following result:

```
((pie & pie_shell & sugar & pear & fruit & !apple))
```

Thus the result of the adaptation is:

$$\text{pie} \land \text{pie\_shell} \land \text{sugar} \land \text{pear} \land \text{fruit} \land \neg\text{apple}$$

That is, a pear pie recipe. Apples are removed from the original recipe. However, the original recipe is a fruit pie recipe (according to the domain knowledge: REVISOR/PL must either replace the apples with another fruit, or solve the contradiction that a fruit pie contains no fruit. Adding pears to the recipe does not cause a contradiction, and does not add any costs. Then the best solution is to replace apples with pears.

6.2 REVISOR/PLAK

REVISOR/PLAK is a revision engine in the Propositional Logic formalism, based on the previous engine REVISOR/PL. REVISOR/PLAK implements the algorithm `revise` previously presented in Section 5.3. Given a set of adaptation rules `AK` and two propositional formulae ψ and

μ in the form of the disjunction of their prime implicants, REVISOR/PLAK computes $\psi \dot{+}_{\texttt{AK}} \mu$. That means REVISOR/PLAK has not yet an implementation of the `min-branches` function and prime implicants must be computed through other means.

Figure 6.2 presents an example in Java code on how REVISOR/PLAK can be used to compute adaptation using adaptation rules. Note that the syntax of adaption rules differs from the `left` $\rightsquigarrow$ `right` syntax introduced in this report. In REVISOR/PLAK, rules are written as triples of conjunctions of literals in the form `context` : `left` $*\!=$ `right`. This notation corresponds to `context` $\cup$ `left` $\rightsquigarrow$ `context` $\cup$ `right`.

```
// Set the cost of variable flip to 1
RevisorPLAK.clearFlipCosts();
// Initialize psi and mu as respectively the sets of prime implicants
of Source ∧ DK and Target ∧ DK
PLFormula psi = RevisorPLAK.parseFormula("fruit & pie_shell & egg &
pie & pear & sugar");
PLFormula mu = RevisorPLAK.parseFormula("(fruit & !cinnamon & peach &
!egg & pie & !pear) | (fruit & !cinnamon & apple & !egg & pie & !pear)
| (!fruit & !cinnamon & !apple & !peach & !egg & pie & !pear)");
// Instantiate the rules
RuleSet rules = new RuleSet();
rules.addRule("cake :  egg *= banana", 0.3);
rules.addRule("pie :  egg *= flour & cider_vinegar", 0.3);
rules.addRule(":  pear *= peach", 0.7);
rules.addRule(":  pear *= apple & cinnamon", 0.3);
rules.addRule(":  cinnamon *= vanilla_sugar", 0.3);
rules.addRule(":  cinnamon *= orange_blossom", 0.3);
// Compute the adaptation  PLFormula result =
RevisorPLAK.adaptAK(psi, mu, rules);
result = RevisorPLAK.simplifiedDNF(result);
```

Figure 6.2: Adapting a pear pie recipe with REVISOR/PLAK.

In this example, the goal is to produce a pie recipe without pear, cinnamon or eggs. To this end, a pear pie recipe containing eggs will be used:

$$\begin{aligned}\texttt{Source} &= \text{pie} \wedge \text{pie\_shell} \wedge \text{pear} \wedge \text{sugar} \wedge \text{egg} \\ \texttt{Target} &= \text{pie} \wedge \neg\text{pear} \wedge \neg\text{cinnamon} \wedge \neg\text{egg}\end{aligned}$$

REVISOR/PLAK is tasked with the adaptation of `Source` to solve `Target`, using the following domain knowledge:

$$\texttt{DK} = (\text{apple} \vee \text{peach} \vee \text{pear}) \Leftrightarrow \text{fruit}$$

Apples, peaches and pears are fruits and, conversely, the only available fruits are apples, peaches and pears.

$$\texttt{AK} = \{\texttt{R}_1, \texttt{R}_2, \texttt{R}_3, \texttt{R}_4, \texttt{R}_5, \texttt{R}_6\}$$

$\texttt{R}_1 = \text{cake} \wedge \text{egg} \rightsquigarrow \text{cake} \wedge \text{banana}$	$\texttt{R}_4 = \text{pear} \rightsquigarrow \text{apple} \wedge \text{cinnamon}$
$\texttt{R}_2 = \text{pie} \wedge \text{egg} \rightsquigarrow \text{pie} \wedge \text{flour} \wedge \text{cider\_vinegar}$	$\texttt{R}_5 = \text{cinnamon} \rightsquigarrow \text{orange\_blossom}$
$\texttt{R}_3 = \text{pear} \rightsquigarrow \text{peach}$	$\texttt{R}_6 = \text{cinnamon} \rightsquigarrow \text{vanilla\_sugar}$

Each rule has a cost of 0.3 except $\texttt{R}_3$ which has a cost of 0.7. Indeed, pears are more similar to apples than peaches. Each flip has a cost of 1.

REVISOR/PLAK gives the following result (in less than 20ms):

```
((cider_vinegar & flour & orange_blossom & fruit & !cinnamon
& pie_shell & !pear & pie & !egg & sugar & apple) | (cider_vinegar
& flour & vanilla_sugar & fruit & !cinnamon & pie_shell & !pear
& pie & !egg & sugar & apple))
```

Which can be expressed in propositional logic as:

$$\text{pie} \wedge \text{pie\_shell} \wedge \text{sugar} \wedge \text{fruit} \wedge \neg\text{egg} \wedge \text{flour} \wedge \text{cider\_vinegar} \wedge$$
$$\neg\text{pear} \wedge \text{apple} \wedge \neg\text{cinnamon} \wedge (\text{vanilla\_sugar} \vee \text{orange\_blossom})$$

Two recipes are proposed. In both, pears have been replaced by apples and eggs by flour and cider vinegar. For the cinnamon, two choices were available: either vanilla sugar or orange blossom could replace it: since $\texttt{cost}(\texttt{R}_5) = \texttt{cost}(\texttt{R}_6)$, the disjunction of this possibilities is presented; if $\texttt{cost}(\texttt{R}_5) < \texttt{cost}(\texttt{R}_6)$, only the orange blossom alternative would be given.

In order to propose these two recipe REVISOR/PLAK uses either the rules $\{\texttt{R}_2, \texttt{R}_4, \texttt{R}_5\}$ or $\{\texttt{R}_2, \texttt{R}_4, \texttt{R}_6\}$. Note that the order in which the rules can be applied does not affect the results, however $\texttt{R}_5$ and $\texttt{R}_6$ cannot be applied before $\texttt{R}_4$. Indeed, both $\texttt{R}_5$ and $\texttt{R}_6$ needs the presence of the literal *cinnamon* to be applied, and it is only introduced by the rule $\texttt{R}_4$. Thus, 6 adaptations paths are proposed by REVISOR/PLAK:

$\texttt{R}_2, \texttt{R}_4, \texttt{R}_5$	$\texttt{R}_2, \texttt{R}_4, \texttt{R}_6$
$\texttt{R}_4, \texttt{R}_2, \texttt{R}_5$	$\texttt{R}_4, \texttt{R}_2, \texttt{R}_6$
$\texttt{R}_4, \texttt{R}_5, \texttt{R}_2$	$\texttt{R}_4, \texttt{R}_6, \texttt{R}_2$

6.3 Evaluation

So far, empirical data shows that the computing time is largely dependent upon ϱ (that is the sum of each flip divided by the minimum transition cost, as defined in Section 5.6). However, a million-fold increase of ϱ may result in only a tenfold increase of computing time, despite worst-case complexity being exponential with respect to ϱ. This shows that actual performance is far better than worse-case complexity may suggest. The current implementation is vulnerable to sets of rules $\{(\mathtt{left}_i, \mathtt{right}_i) \mid i \in \{0, 1, \ldots, n\}\}$ such that $\mathtt{left}_i \subseteq \mathtt{right}_{i-1}$ for $i \in \{1, \ldots, n\}$ and $\mathtt{left}_0 \subseteq \mathtt{right}_n$, that is, rules that could potentially create infinitely looping paths. While the algorithm always terminate, the algorithm could generate very long finite paths using such rules, if the total cost of the loop is very small compared to the cost of the solution. The most important factor for computing time is the distance between ψ and μ. That is, having an adequate source case to solve the target case make the adaptation problem much easier to solve. REVISOR/PLAK computing time has been evaluated on randomly generated adaptation problems where the source case is a conjunction of literals and the target case is a formula in DNF. Formulae have been generated with a set of 50 variables. 40 adaptation rules have been provided to perform the adaptation. The current implementation solves 87.5% (with a standard deviation of 4.6 points on series of 50 tests) of tested adaptation problems in less than one second, on a computer with a 2.13Ghz processor and 3GB of available memory. Problems which could not be solved in less than one second usually did not have enough memory available to terminate, while others terminate in 19 milliseconds on average.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The objective of this internship was to design and implement an adaptation algorithm to be used in a case-based reasoning process. Given a problem or target case, and a source case retrieved by the case-based reasoning system, the algorithm had to adapt the latter to solve the problem posed by the target case.

As a result of this internship, an original algorithm for revision-based and rule-based adaptation based on tableaux repairs in propositional logic has been proposed. This algorithm modifies minimally the source case so that it becomes consistent with the target case. The semantic tableaux method is applied separately on the source and target cases. Then the consistent branches are combined, which leads to a set of branches, each of them containing one or more clashes (unless the source case is consistent with the target case and need not to be adapted). Those clashes are repaired with adaptation rules which modify the source case. Adaptation rules allow to substitute a given part of the source case by a formula. If no rule is available a flip on the literal deletes it from the source case. Flips allows the algorithm to always find a solution.

The algorithm is based on the A^* procedure: for each literal ℓ, the minimal cost to repair a clash on ℓ is computed. The heuristic $\mathcal{H}$ for a state is defined as the sum, for each clash in this state, of the minimal cost to repair that clash. Thus, at each step, the state developed is the one for which the cost plus the estimated cost to repair the clash is the lowest. Once a solution is found, only states whose cost is lower or equal to the best final cost are developed. This adaptation algorithm has been implemented in the REVISOR/PLAK tool.

7.2 Future Work

Belief revision is one of the operations of belief change. There are other ones (see [Peppas, 2008], for a synthesis) such as contraction ($\psi \dot{-} \mu$ is a belief base obtained by minimally modifying ψ so that it does not entail μ) or integrity constraint belief merging [Konieczny and Pérez, 2002] ($\Delta_\mu(\{\psi_1, \ldots, \psi_n\})$ is a belief base obtained by minimally modifying the belief bases ψ_i into ψ'_i such that $\mu \wedge \bigwedge_i \psi'_i$ is consistent). A future work consists in studying how the tableaux repair approach presented in this report can be modified for such belief change operations. This would have an impact on case-based reasoning; in particular, integrity constraint belief merging can be used for multiple case adaptation (i.e., combining several source cases to solve the target case), see [Cojan and Lieber, 2009] for details.

Optimizing the algorithm. One downside of the algorithm is that in many practical situations, the order in which rules are applied does not affect the result: then the algorithm explores redundant adaptation paths. A future work may consist in finding a way to generate less equivalent adaptation paths, which would importantly decrease the computing time. For example, the order in which two consecutive flips are applied does not change the result obtained. Thus, it could be possible to apply consecutive flips only in one arbitrarily chosen order. Then, a similar approach could be used on some sets of adaptation rules.

Currently, the implemented algorithm may generate states already explored, as shown in the running example (see figure 5.2). Because A^* is a best-first search algorithm and cost are strictly positive, cycles are often ignored quickly. Still, exploring states no more than once would result in an improvement in computing time. Though, checking if a state has already been explored is not inexpensive in memory and computing time: that implies storing every explored state in a data structure, and comparing every newly generated state to the states in that structure. A future work may consist in evaluating the benefits and drawbacks of implementing that feature. Another possible approach to not explore cycles would be to detect which rules of `AK` create cycles, and to check for pre-exploration only when those rules are used.

Bibliography

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59.

[Alchourrón et al., 1985] Alchourrón, C. E., Gärdenfors, P., and Makinson, D. (1985). On the Logic of Theory Change: partial meet functions for contraction and revision. *Journal of Symbolic Logic*, 50:510–530.

[Blansché et al., 2010] Blansché, A., Cojan, J., Dufour Lussier, V., Lieber, J., Molli, P., Nauer, E., Skaf Molli, H., and Toussaint, Y. (2010). TAAABLE 3: Adaptation of ingredient quantities and of textual preparations. In *18h International Conference on Case-Based Reasoning - ICCBR 2010, "Computer Cooking Contest" Workshop Proceedings*.

[Cojan et al., 2013] Cojan, J., Dufour Lussier, V., Hermann, A., Le Ber, F., Lieber, J., Nauer, E., and Personeni, G. (2013). REVISOR : un ensemble de moteurs d'adaptation de cas par révision des croyances. In *Journée Intelligence Artificielle Fondamentale*.

[Cojan and Lieber, 2009] Cojan, J. and Lieber, J. (2009). Belief Merging-based Case Combination. In *Case-Based Reasoning Research and Development (ICCBR 2009)*, pages 105–119.

[Cojan and Lieber, 2012] Cojan, J. and Lieber, J. (2012). Belief revision-based case-based reasoning. In Richard, G., editor, *Proceedings of the ECAI-2012 Workshop SAMAI: Similarity and Analogy-based Methods in AI*, pages 33–39.

[Coman and Muñoz-Avila, 2012] Coman, A. and Muñoz-Avila, H. (2012). Diverse plan generation by plan adaptation and by first-principles planning: A comparative study. In Díaz-Agudo, B. and Watson, I., editors, *Case-Based Reasoning Research and Development (ICCBR 2012)*, volume 7466 of *LNCS*, pages 32–46. Springer.

[Dalal, 1988] Dalal, M. (1988). Investigations into a theory of knowledge base revision: Preliminary report. In *AAAI*, pages 475–479.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[Dufour-Lussier et al., 2010] Dufour-Lussier, V., Lieber, J., Nauer, E., and Toussaint, Y. (2010). Text adaptation using formal concept analysis. In *Case-Based Reasoning. Research and Development*, pages 96–110. Springer.

[Eiter and Gottlob, 1992] Eiter, T. and Gottlob, G. (1992). On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57:227–270.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160.

[Katsuno and Mendelzon, 1991] Katsuno, H. and Mendelzon, A. (1991). Propositional knowledge base revision and minimal change. *Artificial Intelligence*, 52(3):263–294.

[Konieczny and Pérez, 2002] Konieczny, S. and Pérez, R. P. (2002). Merging information under constraints: a logical framework. *Journal of Logic and Computation*, 12(5):773–808.

[Lieber, 2007] Lieber, J. (2007). Application of the Revision Theory to Adaptation in Case-Based Reasoning: the Conservative Adaptation. In *Proceedings of the 7th International Conference on Case-Based Reasoning (ICCBR-07)*, LNCS 4626, pages 239–253. Springer, Belfast.

[Manzano et al., 2011] Manzano, S., Ontañón, S., and Plaza, E. (2011). Amalgam-based Reuse for Multiagent Case-based Reasoning. In *19th International Conference on Case Based Reasoning - ICCBR'2011*, pages 122–136.

[Melis et al., 1998] Melis, E., Lieber, J., and Napoli, A. (1998). Reformulation in Case-Based Reasoning. In Smyth, B. and Cunningham, P., editors, *Fourth European Workshop on Case-Based Reasoning, EWCBR-98*, LNCS 1488, pages 172–183. Springer.

[Minor et al., 2007] Minor, M., Bergmann, R., Görg, S., and Walter, K. (2007). Towards Case-Based Adaptation of Workflows. In *Proceedings of the 7th International Conference on Case-Based Reasoning*, LNCS 4626, pages 421–435, Belfast. Springer.

[Pearl, 1984] Pearl, J. (1984). *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Co., Reading, MA.

[Peppas, 2008] Peppas, P. (2008). Belief Revision. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, chapter 8, pages 317–359. Elsevier.

[Personeni et al., 2013] Personeni, G., Hermann, A., and Lieber, J. (2013). Adaptation de cas propositionnels par réparations fondées sur des connaissances d'adaptation. In *21ème atelier Français de Raisonnement à Partir de Cas*.

[Riesbeck and Schank, 1989] Riesbeck, C. K. and Schank, R. C. (1989). *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey.

[Rubin and Watson, 2012] Rubin, J. and Watson, I. (2012). Opponent type adaptation for case-based strategies in adversarial games. In Díaz-Agudo, B. and Watson, I., editors, *Case-Based Reasoning Research and Development (ICCBR 2012)*, volume 7466 of *LNCS*, pages 357–368. Springer.

[Schwind, 2011] Schwind, C. (2011). Belief base change on implicant sets: How to give up elements of a belief based on literal level. In *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, CIS '11, pages 239–243, Washington, DC, USA. IEEE Computer Society.

[Socher, 1991] Socher, R. (1991). Optimizing the clausal normal form transformation. *J. Autom. Reason.*, 7(3):325–336.