FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Towards Computing Optimal Solutions for Belief Base Contraction

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.)
in Praktische Informatik

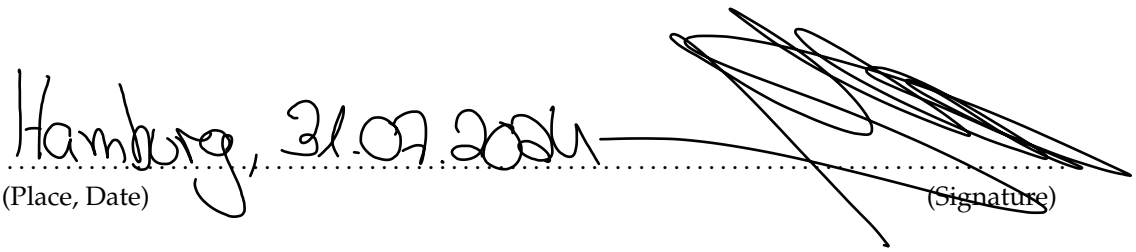
submitted by
Sebastian Mueller

First examiner:	Dr. Jandson Santos Ribeiro Santos Artificial Intelligence Group
Advisor:	Prof. Dr. Matthias Thimm Artificial Intelligence Group

Statement

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

	Yes	No
I agree to have this thesis published in the library.	☒	☐
I agree to have this thesis published on the webpage of the artificial intelligence group.	☒	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐
The source code is available under a GNU General Public License (GPLv3).	☒	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☒	☐



(Place, Date) (Signature)

Abstract

The domain of belief change examines how a rational agent can adjust its beliefs when new and potentially contradictory information arises. To address inconsistencies in belief systems, there are two main approaches: computing minimal inconsistent subsets, called kernels, and computing maximal consistent subsets, called residuals. The search for these kernels and residuals plays an important role in incorporating new information while maintaining basic beliefs. Several belief change algorithms for computing kernels and residues have been proposed in the literature. These computed kernels and residuals can be used to span a search tree that contains all potential solutions for belief base contraction and make them consistent. A major challenge remains to find the optimal solution out of all possible solutions.

In this paper, an innovative approach is presented that builds on established rank-based methods but complements them with a unique strategy for weight computation. Rather than relying solely on ranks, this approach assigns specific values to formulas within a belief base, including values based on inconsistency measures. These values not only enable the ranking of possible solutions to determine the optimal solution, but also facilitate the implementation of a branch-and-bound strategy used to explore the search tree more efficiently. The branch-and-bound implementation in this thesis includes three different pruning approaches that enable the computation of an optimal solution that yields the maximum value, the minimum value, or a combination of both, using different measures for the maximum and minimum values.

Consequently, this work provides an implementation that finds optimal solutions for both maximality and minimality problems. A maximality problem seeks to maximize a given value within the belief base, while a minimality problem aims to minimize a given value. Our approach shows significant advances over existing algorithms, offering an average time saving of 50% in computing kernels and finding optimal solutions in less than half the time required by conventional methods. The implemented branch-and-bound algorithm outperforms known strategies by efficiently pruning the search space and utilizing innovative techniques such as divide-and-conquer and sliding window methods. This work establishes a robust framework for belief base contraction and shows outstanding performance in both maximization and minimization problem solving scenarios.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

In the area of artificial intelligence, often abbreviated as "AI", a belief system constitutes a meticulously organized representation of an agent's convictions, viewpoints, accumulated knowledge, and underlying presumptions concerning the world. This construct serves as a mechanism for encoding and managing an agent's comprehension of its operational environment, encompassing its own capabilities and constraints. In essence, a belief system provides the bedrock upon which decision-making, logical reasoning, and interactions pivot within a multitude of AI applications.

An agent's epistemic state, as articulated in the work of Gardenfors [Gä88], characterizes the assortment of beliefs held by that agent at any given juncture, offering an abstract representation of its cognitive condition. Numerous techniques exist to model this epistemic state. One prevalent methodology involves representing it either as a belief set or a belief base. A belief set comprises propositions that are logically consistent with one another, as a belief set is inherently logically closed, while a belief base encompasses arbitrary sets of propositions.

Embedded within a model of the epistemic state, an agent may harbor diverse epistemic attitudes toward each constituent proposition. For instance, within a probabilistic framework, an agent may accept or reject a proposition with a specific probability. Such acceptances or rejections precipitate changes in the agent's beliefs, thereby engendering an altered epistemic state. This process of belief alteration is formalized as a function that produces a novel epistemic state predicated on both the input and the current epistemic state [Rib13].

Example 1. A formidable challenge in the domain of belief change lies in an agent's imperative to determine which belief to relinquish when confronted with novel information that contradicts its existing beliefs. For instance, consider a database featuring the ensuing pieces of information:

α: Every ball hit over the fence is a homerun.

β: The most recent hit sailed over the fence.

γ: The most recent hit traveled a distance of 416 feet.

δ: The fence at Coors Field measures 415 feet.

If this database is coupled with an agent capable of executing logical inferences, the following factual inference can be drawn from the information α to δ:

ϵ: The most recent hit was a homerun.

However, suppose that the last hit turns out to be a fly-out, flagrantly contradicting the established belief ϵ. Consequently, it becomes imperative to introduce the fact $\neg\epsilon$, that is, the negation of ϵ, into the database. However, doing so would

render the database inherently inconsistent. To restore consistency, certain beliefs from the original database must be retracted. However, it is unwise to jettison all beliefs indiscriminately, as this would entail an unnecessary forfeiture of valuable information. Thus, a decision must be made regarding whether to retract α, β, γ, or δ.

The crux of the problem in belief change is that pure logical considerations alone do not offer guidance on which beliefs to relinquish; such determinations necessitate additional means. What further complicates matters is that beliefs contained within a database have logical implications. Hence, when withdrawing a belief, one must also make judicious choices regarding which of its consequences to retain and which to discard. For instance, if the decision is made to abandon α in the aforementioned scenario, α carries with it logical entailments, including the following two propositions:

α': Every ball hit over the fence, with the exception of the most recent hit, is considered a homerun.

α'': Every ball hit over the fence is a homerun, except at Coors Field.

It is evident that these propositions should not persist in the revised database. Hence, the ability to methodically modify a belief base is crucial in resolving these intricate challenges.

In the realm of belief base modification, finding kernels and remainders plays a pivotal role in accommodating new information while preserving foundational beliefs. This paper presents a novel approach that builds upon Dixon's [Dix94] rank-based methodology but introduces a distinct perspective centered on value-based systems.

1.1. Objectives

The objectives of this thesis are laid out in several phases. First, there is the implementation of known algorithms for finding kernels and remainders, drawing on the work of Resina et al. [RRW14]. Next, we have the weight assignment. Rather than relying solely on ranks, the approach in this paper assigns specific values to each formula within a belief base, enabling the computation of kernel and remainder values to identify optimal solutions. Furthermore, this approach computes a hitting set tree using the computed kernels. During the spanning set tree, the assigned values can be used to implement a Branch-and-Bound strategy to efficiently find the optimal solution, saving computing resources and time. Therefore, the first research question can be formulated as follows:

Research Question 1: *How effective are the algorithms for identifying kernels in handling belief changes according to Ribeiro [Rib13] and how can the algorithm be improve*

To answer this question, we will begin by implementing the algorithms proposed by Ribeiro [Rib13] for finding kernels and remainders. We will then conduct a series of tests to evaluate the accuracy and efficiency of these algorithms. To improve the algorithms proposed by Ribeiro [Rib13], we will implement several strategies such as sliding window and divide and conquer, as proposed by [CW15].

Research Question 2: *What is the impact of different weight assignment strategies on the computation of kernels in a belief base?*

This question will be addressed by developing and implementing three phases of weight assignment strategies: uniform weight assignment, random weight assignment, and inconsistency measures according to Niskanen et al. [NKTJ23]. Each strategy will be applied to the belief base, and the resulting kernels will be computed. We will compare the outcomes in terms of cardinality, computational efficiency, and the degree of inconsistency. Statistical analysis will be conducted to determine which weight assignment strategy yields the most optimal and consistent results.

Research Question 3: *How does the incorporation of a Branch-and-Bound strategy influence the efficiency and outcome of the hitting set tree algorithm?*

To explore this question, we will develop a hitting set tree algorithm that uses kernels identified in the initial steps. We will then integrate a Branch-and-Bound strategy using the weight assignment techniques. By comparing the performance of the hitting set tree algorithm with and without the Branch-and-Bound strategy, we will assess the improvements in computational efficiency and accuracy. Detailed performance metrics will be collected and a comparative analysis will be performed to highlight the benefits and limitations of using the Branch-and-Bound strategy.

Research Question 4: *What are the key differences and patterns observed when generating hitting set trees with uniform versus random weight assignment strategies?*

This question will be investigated by systematically computing hitting set trees using both uniform and random weight assignment strategies. We will analyze the results to discern any significant differences in the structure, size, and computational requirements of the generated hitting-set trees. Patterns and trends will be identified to understand how different weight assignment strategies impact the development and efficiency of the hitting set trees. The findings will be documented, and their implications for optimizing belief base computations will be discussed.

1.2. Contributions

This thesis aims to address the challenges posed by the advancements and improvements introduced to the field of belief base contraction and algorithm optimization. The original contributions of this thesis are:

- The enhancement of known algorithms designed for computing kernels and remainders [Rib13], by incorporating a sliding window technique and a divide and conquer strategy, as proposed by [CW15]. These enhancements have significantly increased the algorithm's efficiency and accuracy, enabling it to handle larger and more complex belief bases with improved performance.

- The development of an algorithm for the assignment of values within belief bases. This was achieved through three distinct approaches:
  - **Cardinality:** Each element was assigned a value of 1, facilitating the computation of the cardinality of kernels.
  - **Random Assignment:** Values were randomly assigned to each element, providing a varied approach to weight assignment that supports robust experimentation and analysis.
  - **Inconsistency Weight Assignment:** An advanced algorithm was implemented to assign an random measure to each element, based on the work of Kuhlmann et al. [KGLT23]. This approach quantifies the degree of inconsistency associated with each formula, providing a deeper understanding of the belief base dynamics.

- The development of an advanced hitting-set (HS) tree algorithm designed to generate a comprehensive HS-tree using various search strategies. This includes a hybrid search that dynamically switches between depth-first and breadth-first searches, as well as a priority-based search that prioritizes branches based on their assigned values. By utilizing the identified kernels to construct the tree, this algorithm offers a structured and efficient approach to managing belief base modifications. The implementation of this sophisticated HS-tree algorithm provides a robust foundation for further analysis and optimization in belief base processing, enhancing the ability to handle complex and dynamic belief systems effectively.

- The integration of a Branch-and-Bound strategy into the hitting-set tree algorithm. This strategy leverages the assigned values to intelligently navigate the search space, selectively exploring the most promising branches and thereby enhancing the efficiency and effectiveness of the algorithm. The Branch-and-Bound strategy has been shown to significantly reduce computational overhead and improve the speed of finding optimal solutions.

- The implementation of an algorithm designed to find the optimal solution for the contraction of the belief base. This algorithm builds on previous contributions, using improved kernel and remainder computation, weight assignment strategies, and the optimized hitting-set tree algorithm. The result is a powerful and efficient tool for the contraction of the belief base that provides optimal solutions with improved accuracy and reduced computational requirements.

1.3. Structure

The thesis is structured into several chapters. Section 1 presents the introduction, followed by Section 2, which lays out the theoretical foundation of belief base theory, belief base contraction, hitting set trees, and advanced stratagies. Section 3 cover related work found in the current literature, which will be analyzed and discussed. Section 4 provides an overview of the approach of this thesis and the fundamentals of the Branch-and-Bound Framework. In Section 5 we describe the implementation of our system and improvements, which will be evaluated in Section 6. Finally, in Section 7, we present our conclusions and findings and provide an outlook on future work.

2. Background

The purpose of this chapter is to present the fundamental concepts of belief change and the corresponding belief change operators. Additionally, we will discuss the algorithms and strategies used to identify kernel and remainder sets in belief bases. This chapter starts with formal preliminaries and then introduces the AGM theory, providing the foundational knowledge required to understand the subsequent discussions on belief change methodologies and processes.

2.1. Formal Preliminaries

In the rest of this thesis, we will utilize the following definitions and concepts, beginning with the definition of a (propositional) atom.

An *atom* is a basic proposition that can either be *true* or *false* and does not contain any logical connectives. Atoms are represented by lower case Latin letters or by capital Latin letters followed by an Arabic number. For example:

p: *The sky is blue.*
q: *It is raining.*

The truth values *true* and *false* in propositional logic are sometimes denoted by 1 and 0, respectively. A *signature* is a finite set of atoms.

A *propositional formula* is constructed from atoms using the following logical connectives:

- $\neg$ (negation)
- $\wedge$ (conjunction)
- $\vee$ (disjunction)
- $\rightarrow$ (implication)
- $\longleftrightarrow$ (equivalence)

Given a specific truth assignment to the atoms in a formula, the formula can be either *true* or *false*. Propositional formulas are denoted by lower case Greek letters, as follows:

α: p
β: $\neg p \vee q$

A *literal* is an atom or the negation of an atom.

The set of all atoms occurring in a given propositional formula α, i.e., its signature, is denoted by $Var(\alpha)$.

A *propositional logic language* $\mathcal{L}$ on a set of atoms σ, denoted as $\mathcal{L}_\sigma$, consists of the set of all propositional formulas that can be formed from the signature σ.

An interpretation M that makes a formula α true is called a *model* of α, denoted by $M \models \alpha$. The set of all models of α is expressed by *Mod*(α).

For the remainder of this thesis, we will use the following concepts of entailment and satisfiability in propositional logic.

Definition 1. The expression $\alpha \models \beta$ states that β is an *entailment* of α, meaning that every interpretation (assignment of truth values to variables) that makes α true also makes β true. This can be expressed as *Mod*$(\alpha) \subseteq$ *Mod*(β). Similarly, a set of formulas B entails a formula α (denoted as $B \models \alpha$) if every interpretation that makes all formulas in B true also makes α true.

A set of formulas B is *satisfiable* if there exists an interpretation that makes all the formulas in B true. Conversely, it is *unsatisfiable* if no such interpretation exists.

Definition 2. Let B be a *belief set* or *belief theory* in a propositional logic language $\mathcal{L}$ and let the *consequence operator* $Cn()$ be defined as: $Cn(X) = \{\alpha \in \mathcal{L} \mid X \models \alpha\}$. Then A is a deductively closed set of beliefs if $A = Cn(A)$.

A belief is represented by a propositional formula, and an arbitrary set of beliefs, also known as a belief base, is considered to be the logical conjunction of its members.

2.2. AGM Theory

The starting point of belief change was the landmark paper from 1985 by Alchurrón, Gärdenfors and Makinson [AGM85]. With their work, the authors have created a basis for the investigation of changes in belief states and databases [FH11], commonly known as the AGM theory. The AGM theory provides a formal framework for examining changes in epistemic states, which are depicted as logically closed sets of sentences. These sets of sentences, embodying an agent's epistemic state, are termed *belief sets*. Assuming agents operate within a certain logic $\langle \mathscr{L}, Cn \rangle$ that adheres to the AGM assumptions, these belief sets are collections of sentences that remain consistent under logical implication. In other words, given the logic framework $\langle \mathscr{L}, Cn \rangle$, a belief set $\mathcal{K}$ fulfills the condition $\mathcal{K} = Cn(\mathcal{K})$, or equivalently, $\mathcal{K} \in \mathbb{K}_{\mathscr{L}}$.

The AGM theory contemplates three distinct epistemic attitudes toward a sentence. For a given belief set $\mathcal{K}$ of an agent, the sentence α can be:

accepted:	if $\alpha \in \mathcal{K}$.
rejected:	if $\alpha \notin \mathcal{K}$ and $\mathcal{K} \cup \{\alpha\}$ is inconsistent.
indeterminate:	if $\alpha \notin \mathcal{K}$ and $\mathcal{K} \cup \{\alpha\}$ remains consistent.

Given a belief set $\mathcal{K}$ within the logic $\langle \mathscr{L}, Cn \rangle$, the symbols $K + \alpha$, $\mathcal{K} - \alpha$, and $\mathcal{K} * \alpha$ signify the belief set resulting from an expansion by α, a contraction by α, and a revision by α, respectively, as explained in more detail as follows.

Expansion is the simplest of those operations and can be achieved using the following formula:

$$\mathcal{K} + \alpha = Cn(\mathcal{K} \cup \{\alpha\})$$

Revision entails the steadfast acceptance of a sentence α. In addition to ensuring the acceptance of the input ($\alpha \in \mathcal{K}*\alpha$) and the coherence of the resulting set of beliefs ($\mathcal{K} * \alpha$ is consistent), revision must also ensure that the alteration is carried out in a manner that is as minimal as possible. Unlike expansion, revision is significantly more intricate due to its incorporation of "extra-logical" elements.

Contraction involves eliminating a sentence α from the belief set $\mathcal{K}$. In addition to ensuring that the input becomes indeterminate in the updated belief set ($\alpha \notin \mathcal{K} - \alpha$), contraction should also ensure that $\mathcal{K} - \alpha$ remains a valid belief set and that the modification is conducted with a degree of minimalism. The process of contraction is further influenced by non-strictly logical considerations.

Example 2. In a time when the geocentric model, with Earth at the center of the universe, dominated astronomy, Nicolaus Copernicus proposed the revolutionary heliocentric model, placing the Sun at the center. Despite resistance from the scientific community, Copernicus recognized the need for a paradigm shift in cosmology. This shift required more than logical criteria; it demanded a fundamental reassessment of established beliefs. The resulting Copernican Revolution transformed our understanding of the universe, illustrating the complexity of belief revision and the broader scope of scientific discovery.

Yet, the process of choosing which beliefs to discard during this revision cannot be guided solely by logical criteria. Consequently, revision must be characterized through a collection of rationality postulates, distinct from the way expansion was defined. In this context, revision is established by adhering to a set of principles that govern rational updating of beliefs. The postulates that govern this process are described in Section 2.3.2.

2.3. Belief Change

An epistemic input can cause a belief change to be processed as an operation. An input can trigger the following three operations [Rib13].

expansion:	makes the agent accept a new sentence.
revision:	makes the agent accept a new sentence in a consistent manner.
contraction:	makes the agent abandon the belief in a sentence.

Expansion, as already mentioned, represents a straightforward operation involving the acceptance of a sentence α. In contrast, revision demands consistent acceptance of the sentence α, even when it conflicts with the existing belief base $\mathcal{K}$. This is done under the condition that the revised belief base maintains coherence and thoroughness concerning logical implications. On the other hand, contraction entails the systematic removal of the previously held sentence α from the belief base $\mathcal{K}$. The

challenge in contraction lies in determining which accompanying sentences should be rejected along with α to ensure that the resulting belief base remains closed concerning logical consequences.

Among these three operations, expansion is uniquely determined. However, in contrast, both contraction and revision are guided by sets of postulates [AGM85]. In these operations, certain axioms — fundamental principles that form the basis of the logical system — might be discarded, leading to occasional situations where only a solitary plausible choice emerges. Nonetheless, this introduces one of the most distinct elements of the AGM theory: the rationality postulates. These postulates, which will be explored in the upcoming sections, play a pivotal role in shaping the theory's foundation.

The study of belief revision focuses on understanding how an agent's epistemic state dynamically evolves, specifically how its attitudes toward elements in the model change due to an external trigger known as epistemic input. For our purposes, we are primarily concerned with the effect of this input on the agent's epistemic state.

Transitioning from belief sets to belief bases offers a more nuanced and flexible approach to epistemic reasoning. While belief sets provide a foundational understanding of an agent's epistemic state, belief bases add an extra dimension of complexity and adaptability.

Belief bases, in essence, represent a broader perspective on how an agent's beliefs are structured and managed. Unlike belief sets, which are defined as logically closed sets of sentences, belief bases encompass arbitrary collections of sentences without the stringent requirement of logical closure. This flexibility allows belief bases to capture a wider range of epistemic states, accommodating situations where beliefs might not adhere to strict logical consistency.

The transition from belief sets to belief bases acknowledges that in real-world scenarios, an agent's beliefs may encompass a diverse array of statements, some of which might not be immediately subject to logical closure. By transitioning to belief bases, we allow for a more nuanced understanding of belief dynamics, where epistemic attitudes can encompass a broader spectrum of acceptance, rejection, or indeterminacy, without the requirement that all logical consequences of the beliefs are included.

Building on the foundations of belief sets, the introduction of belief bases enhances our understanding of belief revision and the rationality postulates that guide it. This approach introduces a new set of principles and considerations for manipulating and transforming belief bases, providing a more detailed framework for studying belief dynamics.

2.4. Belief Base Theory

Belief base theory studies the dynamics of epistemic states represented as arbitrary sets of sentences B. It studies a belief system that admits four types of epistemic

attitudes with respect to a sentence α:

reject:	α is not consistent with B.
explictly accept:	$\alpha \in B$.
implicitly accept:	$\alpha \in Cn(B)$, but $\alpha \notin B$,
undetermined:	α consistent with B and $\alpha \notin Cn(B)$.

If $\alpha \in Cn(B)$, we will simply say that α is accepted, i.e., α is accepted if it is implicitly or explicitly accepted.

Belief base theory encompasses the same three fundamental modes of belief change as AGM theory: expansion, revision, and contraction. In the context of expansion, a statement α transitions to an accepted state, defined simply as $B + \alpha = B \cup \{\alpha\}$. Just as in AGM theory, the processes of contraction and revision are rigorously articulated through a set of rationality postulates.

However, it's worth noting that terminology in the realm of belief systems can be nuanced. Some scholars, like Refenes [Ref91], employ the term *belief base* to describe a finite representation of a belief set. Our perspective diverges from Refenes' approach and aligns with authors such as Fuhrmann, Hansson, and Wassermann [Fuh96, Han99, Was99]. In the paradigm advocated by this second group of scholars, belief bases represent a distinct belief system. Within this framework, the agent's explicit beliefs are demarcated from those that emerge solely as consequences of these explicit convictions.

To illustrate the distinction between the belief set and belief base approaches, consider the following example:

Example 3. Extending our exploration of belief revision in Copernicus's paradigm-shifting work, imagine Copernicus firmly believes in the heliocentric model (α) as the cornerstone of his cosmological theory. He also holds another belief (β) about predictable planetary positions based on this model. In his worldview, these beliefs culminate in the central tenet $\alpha \leftrightarrow \beta$, signifying their intrinsic connection. However, as Copernicus continues his observations, he discovers a discrepancy ($\neg\beta$) contradicting his predictions. He faces a dilemma: whether to retain both α and $\alpha \leftrightarrow \beta$ or reassess his beliefs in light of this empirical contradiction.

In the belief set approach, Copernicus confronts a critical decision. Both α and $\alpha \leftrightarrow \beta$ are elements within his belief set, and he cannot reconcile the empirical evidence of $\neg\beta$ with these existing beliefs. He recognizes that maintaining both α and $\alpha \leftrightarrow \beta$ simultaneously is untenable, and a choice must be made. However, the removal of $\alpha \leftrightarrow \beta$ does not occur automatically; it hinges on a deliberate selection mechanism.

Contrastingly, within the belief base approach, the situation is more streamlined. The sentence $\alpha \leftrightarrow \beta$ is derived from the other beliefs, primarily α and β. When Copernicus acknowledges the empirical anomaly ($\neg\beta$), the logical structure of belief bases ensures that $\alpha \leftrightarrow \beta$ is automatically removed. This immediate revision is guided by the inherent dynamics of belief bases, simplifying the process of belief

management in the face of empirical evidence.

Both approaches offer distinct advantages. In the belief set approach, equivalent epistemic states are treated uniformly, abstracting away from the syntactic form of beliefs. Conversely, the belief base approach offers greater expressiveness and holds heightened computational intrigue.

2.4.1. Base Contraction

This section introduces the concept of contraction using a set of rationality principles known as *AGM postulates for contraction*. Subsequently, we will explore specific approaches to contraction known as the "partial meet contraction" and "kernel contraction" operation. Both the aforementioned postulates and the construction of partial meet were initially put forth in the seminal work by Alchourrón et at. [AGM85], wherein Hansson introduced kernel contraction [Han94]. Additionally, this section will encompass the representation theorem that establishes the connection between the construction and the postulates, as presented in the same sources.

2.4.2. Base Contraction Postulates

When an agent begins to question the validity of some of its beliefs, it embarks on a process known as contraction, which essentially corresponds to the act of open-mindedness.

Consider the following example to illustrate this concept:

Example 4. Imagine a seasoned physicist who has dedicated years to a particular theory of particle physics, firmly convinced that it provides the most accurate description of the universe's fundamental building blocks. However, at a scientific conference, the physicist encounters a new, intriguing theory proposed by a young researcher. This theory challenges some of the core tenets of the physicist's long-standing beliefs. Instead of dismissing it outright, the physicist decides to exercise open-mindedness. They engage in the process of contraction, temporarily setting aside their deep-seated convictions in favor of exploring the new theory with an unbiased perspective. This act of contraction allows the physicist to consider alternative viewpoints, fostering a spirit of scientific inquiry and discovery. It exemplifies the essence of open-mindedness within the realm of scientific exploration.

Belief base contraction is an operation within a belief base B that renders a sentence α indeterminate. In essence, an agent, who initially accepts (implicitly or explicitly) the sentence α, should, after contraction, hold no definitive stance on α. Additionally, it is preferable that the modification to the agent's belief base remains as minimal as possible. Fermé and Hansson [FH18] have delineated the postulates that a contracted belief base B must adhere to, and these postulates are as follows:

(**success**) If $\alpha \notin Cn(\emptyset)$ then $\alpha \notin Cn(B - \alpha)$

The success postulate ensures that if contracting α from B, then the contracted belief base B' should be consistent and α should become undetermined.

(**inclusion**) $B - \alpha \subseteq B$

Inclusion means that when contracting α from B, the resulting base $B - \alpha$ should always be a subset of B. In other words, you should never end up believing fewer things than you did before the contraction. This postulate ensures that your belief contraction does not lead to losing beliefs but rather refines or narrows down your beliefs while staying within the scope of what you originally believed.

(**relevance**) If $\beta \in B$ and $\beta \notin B - \alpha$, then there is a B' such that $B - \alpha \subseteq B' \subseteq B$ and $\alpha \notin Cn(B')$, but $\alpha \in Cn(B' \cup \{\beta\})$

The relevance postulate means that if you keep believing something while removing another belief, there should be a way to do it in a way that does not make the removed belief immediately true again. But when you add back the original belief, the removed belief should become true once more. This postulate ensures that your beliefs are consistent and that the removal or contraction of beliefs is done in a way that maintains logical coherence.

(**uniformity**) If $p \in Cn(B')$ if and only if $q \in Cn(A')$ for all subsets A' of A, then $A - p = A - q$.

If two beliefs (p and q) have the same logical consequences in all possible subsets of your beliefs (B' of B), then when you remove or contract either p or q you should end up with the same remaining set of beliefs ($A - p = A - q$). In other words, if the consequences of p and q are consistently the same in all situations, then removing either of them should lead to the same final set of beliefs.

(**core-retainment**) If $\beta \in B$ and $\beta \notin B - \alpha$, then there is a B' such that $B' \subseteq B$ and $\alpha \notin Cn(B')$, but $\alpha \in Cn(B' \cup \{\beta\})$

If you believe something (β) and that belief persists even after you have removed or contracted another belief (α), then there must be a way to retain your original beliefs (B') such that α is not logically implied by your retained beliefs, but adding back β to your retained beliefs makes α logically implied again. In simpler terms, it means that if you keep believing something while removing another belief, there should be a way to do it so that the removed belief does not immediately become true again. But when you add back the original belief, the removed belief should become true once more.

Example 5. Suppose we have a belief base represented by the following set of propositional formulas:

$$B = \{p \wedge q, \neg r, s \vee t\}$$

Now, let's consider applying the aforementioned postulates to this belief base. A

possible contraction that satisfies all postulates is:

$$B' = \{p \wedge q, s \vee t\}$$

- **Success:** Here, $\neg r$ has been removed, and the belief base remains consistent.
- **Inclusion:** In this case, the belief base B' is a proper subset of B.
- **Relevance:** Here, we have removed $\neg r$ because it was not relevant to the remaining formulas.
- **Uniformity:** In this case, the formula $\neg r$ is the same in all contexts, so we removed it from B'
- **Core-retainment:** In this case, the core beliefs are $p \wedge q$ and $s \vee t$, so we must retain them in B'. The core-retainment postulate ensures that the core beliefs remain unchanged during contraction.

Consequently, we can effectively characterize belief base contraction with just four postulates: success, inclusion, uniformity, and a minimality criterion, which can be either relevance or core-retainment. Each potential minimality criterion aligns with one of the constructions presented in the subsequent sections. Specifically, the former corresponds to partial meet contraction, while the latter aligns with kernel contraction.

2.4.3. Base Partial Meet Contraction

The first kind of contraction operation that will be present in this section is known as partial meet contraction and was originally presented in [AGM85]. This construction is based in the concept of remainder set, that is a set of maximal subsets (of a given set) that fails to imply a given sentence.

Definition 3. Let B be a set of sentences and α a sentence. The set $B \perp \alpha$ (B remainder α) is the set of sets such that $X \in B \perp \alpha$ if and only if:

a) $X \subseteq B$

b) $X \nvdash \alpha$

c) For all X' such that $X \subset X' \subseteq B$, $X' \vdash \alpha$.

$B \perp \alpha$ is called the remainder set of B by α, and its elements are the remainders of B by α. From the definition of $B \perp \alpha$, it follows that:

- $B \perp \alpha = \{B\}$ if and only if $B \nvdash \alpha$.
- $B \perp \alpha = \emptyset$ if and only if $\vdash \alpha$.

The first statement means that the set of remainders of B by α, denoted as $B \perp \alpha$, contains only one element, which is the set B itself, if and only if B is not logically consistent with α. In other words, if adding α to B does not result in a consistent set ($B \nvdash \alpha$), then the remainder set $B \perp \alpha$ will consist of just the original set B. This implies that adding α does not change the inconsistency of B. The second statement means that the set of remainders of B by α, denoted as $B \perp \alpha$, is empty (contains no elements) if and only if α is logically provable (derivable) from B. In other words, if α can be derived from $B (\vdash \alpha)$, then there are no remainders of B by α; adding α does not leave any sentences in B. Conversely, if $B \perp \alpha$ is empty, it means that α can be logically derived from B.

The partial meet contraction is obtained by intersecting some elements of the remainder set. The choice of those elements is performed by a selection function.

Definition 4. Let B be a set of sentences. A selection function for B is a function γ such that, for all sentences α:

a) If $B \perp \alpha \neq \emptyset$, then $\gamma(B \perp \alpha)$ is a non-empty subset of $B \perp \alpha$.

b) If $B \perp \alpha = \emptyset$, then $\gamma(B \perp \alpha) = \{B\}$.

A partial meet contraction is obtained by intersecting the elements chosen by the selection function.

Definition 5. Let B be a set of sentences and γ a selection function for B. The partial meet contraction on B that is generated by γ is the operation $-_\gamma$ such that for all sentences α:

$$B -_\gamma \alpha = \bigcap \gamma(B \perp \alpha).$$

An operator $-$ on B is a partial meet contraction if and only if there is a selection function γ for B such that for all sentences α: $B - \alpha = B -_\gamma \alpha$.

2.4.4. Base Kernel Contraction

The partial meet contraction operators on a set B are founded upon the selection of maximal subsets of B that do not entail α. An alternative approach involves constructing a contraction operator based on the selection of elements within B that do entail α and then discarding them during the contraction of B by α. This alternative approach was introduced by Hansson in [Han94], resulting in a new contraction operator known as kernel contraction. Kernel contraction can be viewed as a generalization of the safe contraction initially defined by Alchourrón and Makinson in [AM85].

Kernel contraction relies on the selection of sentences within a set B that actively contribute to the entailment of α, and how this selection is employed during the contraction by α. Formally:

Definition 6. Let B be a belief base, i.e., $B \subseteq \mathcal{L}$, and let $\alpha \in \mathcal{L}$. The set $B \perp\!\!\!\perp \alpha$ is a set such that $X \in B \perp\!\!\!\perp \alpha$ if and only if:

a) $X \subseteq B$.

b) $\alpha \in Cn(X)$.

c) if $X' \subset X$, then $\alpha \notin Cn(X')$.

$B \perp\!\!\!\perp \alpha$ is called the kernel set of B with respect to α and its elements are the α-kernels of B.

To contract a belief from α from a set B one must give up sentences in each α-kernel, otherwise α would continue being implied by B. The so-called incision function selects the beliefs to be discarded.

Definition 7. Let B be a set of sentences. An incision function σ for B is a function such that for all sentences α:

a) $\sigma(B \perp\!\!\!\perp \alpha) \subseteq \bigcup(B \perp\!\!\!\perp \alpha)$.

b) If $\emptyset \neq X \in B \perp\!\!\!\perp \alpha$, then $X \cap \sigma(B \perp\!\!\!\perp \alpha) \neq \emptyset$.

Definition 8. Let σ be an incision function for a belief base B. The kernel contraction operation $-_\sigma$ is formally defined as:

$$B -_\sigma \alpha = B \setminus \sigma(B \perp\!\!\!\perp \alpha)$$

Example 6. Let $B = \{p, p \vee q, p \leftrightarrow q, r\}$. Suppose that we intend to contract B by $p \wedge q$. The elements of the kernel set of B with respect to $p \wedge q$ are the minimal subsets of B that imply $p \wedge q$. Hence $B \perp\!\!\!\perp (p \wedge q) = \{\{p, p \leftrightarrow q\}, \{p \vee q, p \leftrightarrow q\}\}$. An incision function must choose at least one sentence from each element of $B \perp\!\!\!\perp (p \wedge q)$. An example of an incision function is $\sigma(B \perp\!\!\!\perp (p \wedge q)) = \{p \vee q, p \leftrightarrow q\}$. In this case, $B - \sigma(p \wedge q) = B \setminus \{p \vee q, p \leftrightarrow q\} = \{p, r\}$.

In summary, belief base theory provides a comprehensive framework for understanding the dynamics of belief change, encompassing expansion, revision, and contraction. This framework helps differentiate between explicitly and implicitly accepted beliefs, as well as those that are rejected or undetermined. The theories and examples discussed illustrate the complexities and nuances of managing belief systems. In the related work section, we will explore various algorithms developed for belief base contraction. These algorithms, along with their methodologies and implications, will be analyzed and discussed in detail.

3. Related Work

This chapter reviews the literature relevant to this thesis, categorizing and presenting key research contributions.

3.1. Boolean Satisfiability Solving (SAT)

A SAT solver is a tool designed to solve instances of the NP-complete *Boolean satisfiability problem* [BHV21]. The *Boolean satisfiability problem* involves determining whether there exists a truth assignment to the variables in a propositional formula (a model) that makes the formula true. Most SAT solvers not only determine the satisfiability of a formula but also provide a possible solution, i.e., a model, if the formula is satisfiable. A common input format for SAT solvers is the DIMACS format, where variables are represented by consecutive natural numbers and the formula is in Conjunctive Normal Form (CNF).

Definition 9. Let a propositional formula in **C**onjunctive **N**ormal **F**orm (**CNF**) be a conjunction of one or more disjunctions, where the disjunctions' disjuncts are literals. These disjunctions are commonly referred to as clauses. We denote the set of clauses of a CNF formula β by $C(\beta)$.

Every propositional formula can be converted to CNF. The naive approach involves repeatedly applying Boolean transformation rules to the initial non-CNF formula until a CNF formula is obtained. Although this method is guaranteed to work, it can result in an exponential increase in the number of clauses. Therefore, more efficient methods have been proposed in the literature. One such method is the Tseitin transformation [Tse83], which produces a CNF formula that is equisatisfiable with the initial formula, with only a linear increase in the number of clauses (proportional to the size of the original formula). The Tseitin transformation introduces new auxiliary variables, which become part of the resulting CNF formula. The number of these new variables is also linear relative to the size of the original formula.

In each clause of the formula, the logical connective $\wedge$ between literals is omitted, and a '0' is appended to the end to mark the end of the clause. Typically, each line contains one clause, while '-' marks a negation. Comment lines start with a 'c'. There is a problem line at the beginning of each DIMACS instance that has the following form:

`p cnf <variables> <clauses>` where `<variables>` is the number of distinct variables (i.e., the highest variable) in the encoding, and `<clauses>` is the total number of clauses.

Example 7. Let B be a belief base with the following set of formulas:

$$B = \{A0 \vee \neg A1, A1 \vee A2, \neg A0 \vee \neg A2\}$$

The DIMACS instance of B can be generated by using the Tseitin transformation as follows:

p cnf 6 12
-4 1 -2 0
4 -1 0
4 2 0
-5 2 3 0
5 -2 0
5 -3 0
-6 -1 -3 0
6 1 0
6 3 0
4 0
5 0
6 0

This SAT instance has three auxiliary variables and is satisfiable with the model $\{-1, -2, 3, 4, 5, 6\}$.

In this thesis, we use a SAT solver to determine whether $B \models \alpha$ by applying the contrapositive approach: B does not entail α if there exists an interpretation that makes all clauses in B true while making α false. This is equivalent to checking if $B \cup \neg\alpha$ is satisfiable. If $B \cup \neg\alpha$ is satisfiable, there exists an interpretation where B is true and α is false, indicating that B does not entail α.

We utilize a SAT solver, such as MiniSat, to determine the satisfiability of a set of clauses. By inputting $B \cup \neg\alpha$ into the SAT solver, we can check for the existence of an interpretation where B is true and α is false. If the SAT solver finds $B \cup \neg\alpha$ unsatisfiable, it means no such interpretation exists, and thus $B \models \alpha$.

To verify whether $B \models \alpha$, the SAT solver is invoked with $B \cup \neg\alpha$ to check its satisfiability. The results from the SAT solver can be interpreted as follows:

- If $B \cup \neg\alpha$ is *unsatisfiable*, then B entails α.
- If $B \cup \neg\alpha$ is *satisfiable*, then B does not entail α.

3.2. Algorithms for Belief Change

This section covers a comprehensive overview of various references is provided that have implemented algorithms for finding a kernel or remainder set in belief bases or belief sets.

Table 1 presents various references and their methodologies to compute both the kernel and remainder in belief revision. Several strategies have been employed over the years, each with its own characteristics and approaches. The following shall give a short overview of the strategies used to compute a kernel or remainder set.

1. **Maxichoice**:

   The Maxichoice strategy emphasizes the identification of the largest possible consistent subset of the belief base. This approach, by focusing on maximal consistent sets, ensures that as much of the original information as possible is retained while integrating new, potentially conflicting data.

2. **Minimal cuts**:

   This strategy operates by determining the smallest subsets of the belief base that, when removed, restore consistency with the new information. By focusing on minimal alterations to the belief base, it strives to disturb the original set of beliefs as little as possible.

3. **Hitting set tree**:

   A hitting set tree, first introduced by [Rei87], is a systematic way to explore all minimal inconsistent subsets (MIS) of the belief base. For every MIS, a hitting set identifies at least one belief from it to be removed to restore consistency. The tree structure allows for an organized traversal of all possible hitting sets, ensuring a comprehensive exploration.

4. **sliding window**: This approach involves examining the belief base using a "window" of a fixed size that slides across it. Within each window, the method checks for inconsistencies and revises beliefs as needed. By doing so, it can efficiently handle large belief bases by focusing on smaller, manageable sections at a time.

5. **divide-and-conquer**:

   Drawing inspiration from the classic algorithmic paradigm, this strategy breaks the belief base into smaller parts, resolves inconsistencies in each, and then merges the results. This modular approach can offer improved efficiency and scalability, especially for large and complex belief bases.

6. **Expand and Shrink**:

   Beginning by expanding the belief base with the new information, any contradictions are then addressed by subsequently "shrinking" the belief set, removing beliefs that contribute to inconsistencies.

7. **Binary Search**:

   Employing a binary search mechanism, this strategy repeatedly divides the belief base into two halves, determining which half contains the inconsistency and then further narrowing down until the source of contradiction is isolated.

These algorithms employ different strategies, as shown in Table 1, and logic frameworks to tackle the problem of identifying kernels and remainders in formal

contexts. Table 1 summarizes the related work of implementations, highlighting various strategies and logic frameworks used by different authors.

Table 1 lists a range of references along with the year of publication, the strategies they used (such as described above), and the type of logic framework employed (FOL, DL, OWL, CPL).

- **FOL (First-Order Logic)**: A formal logical system used in mathematics, philosophy, linguistics, and computer science. It provides a framework for defining logical relations among objects and is used for reasoning about the properties of these objects.
- **DL (Description Logic)**: A family of formal knowledge representation languages designed for representing knowledge about the world and reasoning about it. DLs are used in artificial intelligence to describe the concepts and relationships within a domain and are the basis for ontology languages such as OWL.
- **OWL (Web Ontology Language)**: A language designed for use by applications that need to process the content of information instead of just presenting information to humans. It is used for representing rich and complex knowledge about things, groups of things, and relations between things.
- **CPL (Classical Propositional Logic)**: A type of logic where formulas represent propositions that can either be true or false. It is used in various areas of computer science and mathematics for problem-solving and logical reasoning.

Table 1 clearly shows the variety of methods and frameworks used by different researchers over the years to address the problem of belief base contraction. Each dot in the table indicates the use of a specific strategy or logic framework in the corresponding reference.

Each strategy has its own advantages and is tailored to specific types of problems or belief bases. The choice of strategy depends on the nature of the belief base, the context of the revision, and the desired outcomes.

3.3. Expand and Shrink

A major objective of this thesis will be focused on algorithms for finding kernels, for example Algorithm 1 as introduced by Ribeiro [Rib13].

This algorithm, called "Expand-shrink with Kernel-black-box", uses the strategy of expanding and shrinking strategy as explained above. This algorithm is a straightforward example for an expand and shrink algorithm because both phases, the expand and shrink phase is done iteratively, meaning element by element. The expand and shrink phases can be altered by using one of the strategies included in Table 1, like sliding window or divide-and-conquer, as proposed by Cobe and Wassermann [CW15] for remainder sets.

Reference	Year	Strategy									Logic
		Kernel	Remainder	Maxichoice	Minimal cuts	Hitting tree set	sliding window	divide-and-conquer	Expand/Shrink	Binary search	Logic Framework
[DW93]	1993		•	•							FOL
[Dix94]	1994	•	•	•							FOL
[Jun01]	2001	•						•			N/A
[Kal06]	2006	•	•				•			•	OWL, DL
[KPHS07]	2007	•	•	•			•		•		OWL, DL
[QHH+08]	2008	•				•					DL
[SQJH08]	2008	•				•		•		•	OWL, DL
[JQH09]	2009	•				•			•		DL
[Moo10]	2010	•	•		•		•		•		DL
[Hor11]	2011	•	•				•	•	•		DL
[Rib13]	2013	•	•		•				•		CPL
[RRW14]	2014	•	•		•				•		DL
[CW15]	2015		•				•	•	•		OWL
[JBQ19]	2019	•				•					DL
[Gui20]	2020	•	•			•			•		DL

Table 1: Related work of Implementations

Algorithm 1 takes a belief base B and a sentence α and computes an α-Kernel of B. The algorithm consists of two phases. The expand phase in the function `Expand-shrink()` and the shrink phase in the procedure `Kernel-black-box()`.

The algorithm processes the input as follows:

In line 3 B' is initialized as an empty set. In lines 4 and 5 a loop is called which adds each element β from B to B', if B' entails α until B' is equal to B. This is a very simple expand technique that checks each element of B, namely each β, separately. In line 6 the algorithm checks if B' entails α. In lines 7 and 8 the algorithm proceeds with the shrink phase by calling the "Kernel-black-box" procedure with B' and α as arguments.

The procedure in line 9 aims to find a subset of B' which entails α. In line 11 the procedure checks for each β in B', if $A1$ is entailed by $B' \setminus \{\beta\}$. If so, β is removed from B' (line 12 and 13).

Algorithm 1 from Ribeiro [Rib13] will be explained using the following exemplary

Algorithm 1: Algorithm expand shrink with Kernel-black-box

Input: B, α
Output: Result of the expand shrink algorithm

```
1  Function Expand-shrink(B, α):
2      Heuristic to find one element of B ⊥⊥ α
3      B' ← ∅
4      for β ∈ B do
5          B' ← B' ∪ {β}
6          if α ∈ Cn(B') then
7              Shrink
8              return Kernel-black-box(B', α)
9  Procedure Kernel-black-box(B, α):
10     Find one element of the kernel B ⊥⊥ α
11     for β ∈ B do
12         if α ∈ Cn(B \ {β}) then
13             B ← B \ {β}
14     return B
```

belief base.

Example 8. Let B be the set defined as:

$$\begin{aligned} B = \{&A0, \\ &\neg A0 \vee A1, \\ &\neg A1 \vee A2, \\ &\neg A0 \wedge \neg A1, \\ &A1 \wedge \neg A2, \\ &\neg A0 \vee \neg A1\} \end{aligned}$$

and let α be defined as:

$$\alpha = A1$$

The algorithm:

1. starts with an empty set B' (line 3).
2. Iterates over the elements of B (line 4).
3. Inserts A0 (line 5) and checks that A1 $\notin Cn(B')$ (line 6).
4. Inserts $\neg$A0 $\vee$ A1 (line 5) and checks that A1 $\in Cn(B')$ (line 6).
5. Calls the procedure Kernel-black-box with inputs B' and A1 (line 8).

6. Checks that A1 $\notin Cn(B \setminus \{\beta\})$ (line 12).

7. Checks that A1 $\notin Cn(B \setminus \{\beta\})$ (line 12).

The resulting α-kernel $k1$ is

$$k1 = \{A0, \neg A0 \vee A1\}$$

The α-kernel we have found serves as a starting point for constructing a hitting set tree, which will enable us to systematically investigate the potential solutions for a contraction of B. In the subsequent section, we will dive into the concept of hitting set trees in more detail and discuss how they can be employed to identify the optimal solution.

3.4. Hitting set tree

In this section we will outline the fundamentals of the hitting set tree as proposed by Reiter [Rei87]. According to Reiter [Rei87] a hitting set is defined as follows:

Definition 10. A hitting set is some subset of the component set $\mathcal{U}$ that contains at least one element of each conflict set in the given collection of conflict sets. More formally, a set $H \subseteq \bigcup_{S \in \mathcal{C}} S$ is a hitting set for $\mathcal{C}$ if and only if $H \cap S \neq \emptyset$ for every $S \in \mathcal{C}$. H is a minimal hitting set for $\mathcal{C}$ if and only if there is no proper subset of H which is also a hitting set for $\mathcal{C}$.

Reiter first introduced the system of a hitting set tree, which is defined as follows:

Definition 11. Reiter's algorithm constructs a Hitting-set tree (HS-tree) for the collection of conflict sets B. This tree is a set of nodes v and edges E. Each node $v \in V$ has a label $v.label \in C$, i.e., $v.label$ is a conflict set and each edge $e \in E$ has a label $e.label$ where $e.label \in \bigcup_{S \in C} S$, i.e., $e.label$ is an element of some $S \in C$. A function $P(v)$, the path function, returns the set of edge labels on the path from the root node to a node ν.

The construction of an HS-tree T is carried out in a breadth-first fashion, according to the following rules:

- Firstly, a root node v_{root} for T is generated. v_{root} is labelled with an arbitrary conflict set $S \in C$.
- If a node v is labelled by a set $S \in C$ then for each $\phi \in S$ a successor node v_ϕ is attached to v via an edge e_ϕ labelled with ϕ.
- Each successor node v_ϕ is then labelled with a set $S' \in C$ such that $S' \cap P(v_\phi) = \emptyset$. If no such S' exists, v_ϕ is labelled with a '✓' . A node labeled by '✓' indicates a *terminating node* (leaf) v_{leaf} which has no successors.

We will now use the belief base B from Example 8 and kernels k computed by Algorithm 1 to span a hitting set tree HST. Referring back to Example 8 , we have already computed a kernel $k1$ that will be used as the root node. As this kernel $k1$ has two sets, there are two branches that can be expanded. To find a successor node we have to call Algorithm 1 with the set

$$B \setminus \{A0\}$$

for the first branch and with the set

$$B \setminus \{\neg A0 \vee A1\}$$

for the second branch.

Pursuing with the algorithm, the resulting kernels, that are the successor nodes of $k1$ are as follows:

$$k2 = \{A1 \wedge \neg A2\}$$
$$k3 = \{A0, \neg A0 \wedge \neg A1\}$$

The kernel $k2$ is a singleton set, therefore it only has one successor node. Whereas kernel $k3$ itself has again two successor nodes. The following nodes that are computed by calling Algorithm 1 again are as follows:

$$k4 = \{A1 \wedge \neg A2\}$$
$$k5 = \{A1 \wedge \neg A2\}$$

Both kernels $k4$ and $k5$ are singleton sets, therefore each of them has only one successor node. For a detailed execution of Algorithm 1 reference is made to Example 8.

An example illustrating the construction of an HS-tree T based on the previous example is given in Figure 1.

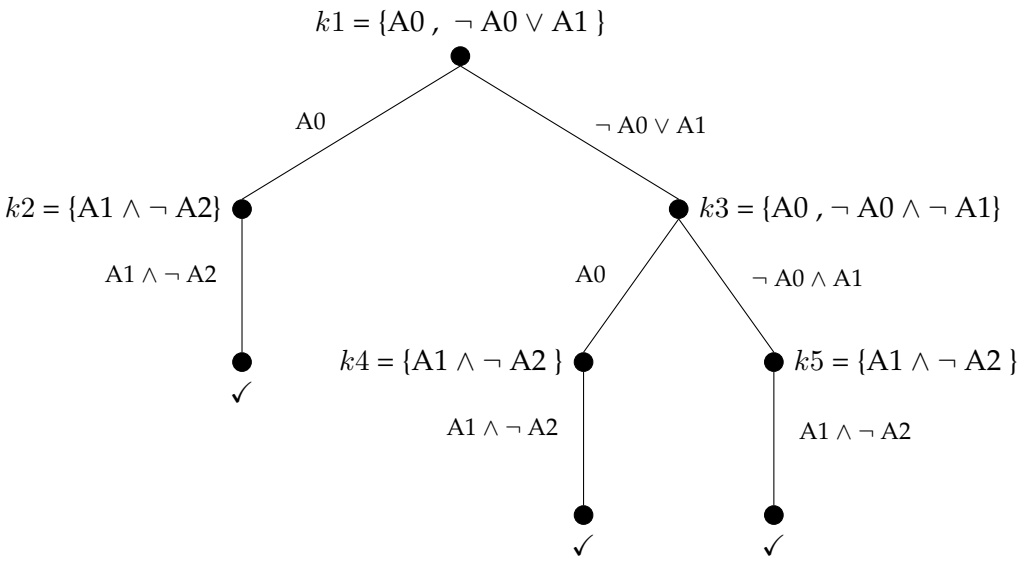


Figure 1: HS-tree for the conflict set B'

Each set $P(v)$ such that v is labeled with '✓' is a hitting set HS for B'. Therefore, the collection $\mathcal{C}$ of all hitting sets HS_i for B' (paths $P(v)$), found in T, contains all hitting sets HS_i for B' [Rei87].

The collection $\mathcal{C}(\mathcal{B})$ of all hitting sets HS_i of B are as follows:

$$\mathcal{C}(B) = \{ \{A1 \wedge \neg A2, A0\}, \{A1 \wedge \neg A2, A0, \neg A0 \vee A1\}, \{A1 \wedge \neg A2, \neg A0 \wedge A1, \neg A0 \vee A1\} \}$$

As illustrated in Figure 2, each set of the collection $\mathcal{C}$ refers to a Path $P(v)$ from a leaf node v_{leaf} to the root node v_{root}. In Figure 2 the color coding shown above was applied to each hitting set HS_i.

The collection of hitting sets derived from the collection $\mathcal{C}$ serves as a crucial step toward contracting the inconsistent belief base B into a consistent state. Each set within $\mathcal{C}$ represents a potential solution for restoring consistency by retracting certain elements β from B.

The resulting HS-tree can be analyzed by determining the tree depth, the number of kernels and the number of branches. We can see that the HS-tree shown in Figure 2 has a tree depth = 3, the number of kernels = 5, and the number of branches = 7

The number of branches always corporates to the number of elements of each α-kernel, so we can see that for belief bases B with a higher complexity and α-kernels with a higher number of elements β, the tree can become bigger and more complex. In Example 8 we saw that the Algorithm 1 calls a SAT-solver in every iteration to check whether α is entailed by B'.

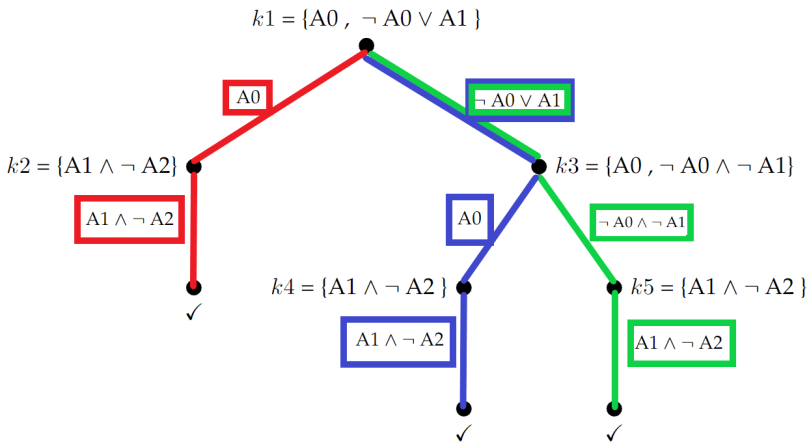


Figure 2: Color coded HS-tree for the conflict set B'

It is important to note that in this thesis, we compute a hitting set tree without determining minimal hitting sets, as we are going to obtain an optimal solution based on values calculated from assigned weights of the elements of a hitting set. According to Reiter [Rei87] a minimal hitting set can be defined as follows:

Definition 12. Suppose $\mathcal{C}$ is a collection of sets. A hitting set for $\mathcal{C}$ is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \emptyset$ for each $S \in \mathcal{C}$. A hitting set for $\mathcal{C}$ is minimal if and only if no proper subset of it is a hitting set for $\mathcal{C}$. A minimal hitting set in the HS-tree corresponds to a set $P(v)$ for a terminating node v such that there is no other terminating node v' in the HS-tree where $P(v') \subset P(v)$.

In the previous example, it is evident that the first hitting set $HS_1 = \{A1 \wedge \neg A2, A0\}$ from the collection $\mathcal{C}(B)$ is a subset of the second hitting set $HS_2 = \{A1 \wedge \neg A2, A0, \neg A0 \vee A1\}$, such that $P(v_{HS_1}) \subset P(v_{HS_1})$. In the remainder of this thesis, we will use a combination of a value of the paths $P(v_{HS_i}$ and the cardinality of the hitting sets HS_i to obtain the optimal sulution. We will demonstrate why determining the minimal hitting sets is redundant when the optimal solution is identified by a combination of the value of the path and the cardinality.

3.5. Branch-and-Bound Algorithms

The Branch-and-Bound (B&B) strategy is a fundamental methodology for solving NP-hard optimization problems. First proposed by Land and Doig [LD60], B&B is better understood as a family of algorithms sharing a core procedure. This procedure implicitly enumerates all possible solutions by storing partial solutions (subproblems) in a tree structure. Unexplored nodes generate children by partitioning the solution space into smaller regions (branching), and rules prune regions that are suboptimal (bounding). Once the entire tree is explored, the best solution can be

returned. While Lawler and Wood [LW66] provided an early overview, more recent advances in the field of B&B were compiled by Morrison et al. [MJSS16].

In a most recent work Bläsius et al. [BFSW23] present a sophisticated B&B algorithm for solving the NP-hard hitting set problem, which seeks a minimal subset of a universe that intersects each set in a given collection. The algorithm leverages various lower bounds on solution size, such as the max-degree, sum-degree, efficiency, packing, and sum-over-packing bounds, along with data reduction rules. Unlike the approach of this thesis, Bläsius et al. focus on finding minimal hitting sets, not hitting sets with a maximal value. The use of lower bounds in the algorithm is to improve the efficiency of the B&B process by pruning branches that cannot lead to a smaller hitting set, thus helping to find the minimal hitting set more quickly.

Unlike the approach of this thesis, Bläsius et al. [BFSW23] do not focus on computing optimal solutions based on a maximum value of the hitting sets. As we will discuss in the following sections, solving a maximality problem using a hitting set tree within a B&B framework is exceptionally challenging. The difficulty lies in accurately estimating the potential value of unexplored subproblems, which is crucial for effective branch pruning. Consequently, known B&B algorithms struggle to prune any branches effectively, leading to the expansion of the entire search space when attempting to find an optimal solution with the maximum value. In the next section, we will formally describe the approach and strategies developed in this thesis, setting the stage for the detailed explanation of the implementation of the application developed as part of this research.

4. Systems approach and strategies

This chapter addresses the various components of the system that form the foundation of this thesis. First, we outline the necessary requirements for computing an optimal solution for belief base contraction. Following this, we present the system model built upon these requirements and derive the approach to achieve the goal of finding an optimal solution.

4.1. Weight assignment

In this section, the requirements for our weight assignment approach will be outlined. In order to do this, we will present related work that dealt with the challenges of ranking belief bases and elements of a belief base.

In the case of the previous Example 8 , any of the hitting sets within $\mathcal{C}$ could feasibly be applied to contract B and alleviate the inconsistency. However, the decision process for selecting the most suitable set from $\mathcal{C}$ calls for additional criteria or mechanisms to inform such a choice, ensuring an optimal outcome for belief contraction. Therefore, a major challenge of this thesis is to develop a weight assignment heuristic that assigns a value to each formula of the belief base B. This work has developed three different weight assignment methodologies that will be described in the following.

4.2. Known rank-based approaches

There has been research done in the area of ontology ranking [PSLP03, DPF$^+$05], and on ranking individual axioms of an ontology [Kal06] or ranking a formula in a belief set [DW93, Dix94].

Among all references, the approach proposed by Dixon [DW93, Dix94] stands out due to its unique approach of computing a rank for a formula within the belief set to efficiently compute a belief change operation. In 1993, Dixon, in collaboration with Wobcke, introduced a rank-based approach for belief change algorithms of belief bases, as documented in [DW93]. This method paved the way for subsequent developments in the field. Notably, in 1994, Dixon further expanded on this approach in [Dix94].

Dixon's rank-based approach utilized a unique system of ranks to achieve minimal changes in the entrenchment ordering of formulas within belief bases. The central idea was to identify kernels that produced the least disruption to the existing entrenchment structure. By doing so, Dixon aimed to find solutions that preserved the foundational beliefs while accommodating new information.

While numerous methodologies for calculating hitting sets and constructing hitting set trees are documented in the literature, there remains a gap in the development of a comprehensive framework that effectively aids in identifying the optimal solution for revisions in belief systems.

4.3. Assignment strategies

Contrary to Dixon's method, the proposed value-based system for identifying an optimal solution unfolds across three stages, each contributing a distinct aspect to the solution-finding process.

At the heart of this system lies a fundamental concept: the assignment of weights w to each constituent element β within the dataset of the belief base B. Subsequent to the computation of kernels and the hitting set tree, each hitting set HS is accorded a specific value ν. This is achieved through the summation of weights w corresponding to the individual elements β residing in the path $P(v)$ from one leaf node v_{leaf} to the root node v_{root}.

Definition 13. Each $\beta \in B$ will be assigned a weight w, the value ν for each hitting set $HS \in C(B)$ is the sum of these weights w, wherein $w(\beta)$ denotes the weight w of element β and $\nu(HS)$ denotes the value ν of hitting set HS.

For the weight assignment, three different approaches were developed. The first approach is called the cardinality measure, the optimal solution will be calculated by the number of elements of the hitting set HS_i. In the second approach, the optimal solution will be calculated by the sum of the assigned random measures. In the third approach, the optimal solution will be calculated by the sum of the assigned inconsistency measures and the cardinality of the hitting set HS_i. For each approach we define a separate value ν_x as follows:

Definition 14. Let $\nu_x(HS_i)$ be the value that is calculated by the sum of the assigned weights w_i of the elements β_i of a HS_i. Let $\nu_c(HS_i)$ be the cardinality value, calculated by the sum of the assigned cardinality weights $w_c(\beta_i)$ of each element β_i of the hitting set HS_i. Further, let $\nu_r(HS_i)$ be the random value, calculated by the sum of the assigned random weights $w_r(\beta_i)$ of each element β_i of the hitting set HS_i and let $\nu_{I_c}(HS_i)$ be the inconsistency value, calculated by the sum of the assigned inconsistency weights $w_{I_c}(\beta_i)$ of each element β_i of the hitting set HS_i.

For the remainder of this thesis, we will denote the belief bases B and the hitting sets HS_i with its elements β_i and the assigned weights w_x by $\beta_i^{w_x}$, such that a belief base B with elements β_1 to β_i and assigned inconsistency weights w_I will be denoted as follows:

$$B = \{\beta_1^{w_x}, \beta_2^{w_x}, \beta_3^{w_x}, ..., \beta_i^{w_x}\}$$

Each of the above-mentioned approaches will be described in the remainder of this section.

4.3.1. Cardinality

The first weight assignment approach is characterized by the assignment of a unified value - specifically, the value of one - to each element within the belief base.

This simplistic yet crucial step sets the foundation by enabling the calculation of the cardinality of kernels and remainders. The cardinality approach allows for either assigning every element of the belief base B a weight w of 1 or simply counting the number of elements in a hitting set.

In this thesis, instead of assigning every element of B a weight w of 1, we count the number of elements β in the hitting set HS to obtain the cardinality of the latter. The value ν of the hitting set HS using the cardinality approach can be defined as follows:

Definition 15. For each hitting set HS_i in the collection $\mathcal{C}(B)$, the value $\nu_c(HS_i)$ is defined as the cardinality of HS_i, i.e., $\nu_c(HS_i) = |HS_i|$.

Example 9. Consider the belief base B from Example 8. The values $\nu_c(HS_i)$ of the hitting sets HS_i of the collection $\mathcal{C}(B)$ can be obtained as shown below:

$$\begin{aligned}
\mathcal{C}(B) = \quad & |\{A1 \wedge \neg A2, A0\}| && \rightarrow \nu_c(HS_1) = |HS_1| = 2 \\
& |\{A1 \wedge \neg A2, A0, \neg A0 \vee A1\}| && \rightarrow \nu_c(HS_2) = |HS_2| = 3 \\
& |\{A1 \wedge \neg A2, \neg A0 \wedge A1, \neg A0 \vee A1\}| && \rightarrow \nu_c(HS_3) = |HS_3| = 3
\end{aligned}$$

In the cardinality approach, the first hitting set HS in the collection $\mathcal{C}$, namely $\{A1 \wedge \neg A2, A0\}$, can be determined as an optimal solution since this set only has two elements. This means that contracting B by this hitting set HS_1 ($B \setminus HS_1$) could lead to a lower loss of information, compared to $B \setminus HS_2$ or $B \setminus HS_3$.

4.3.2. Random Value

Progressing to the next weight assignment approach, a layer of randomness is introduced, where each element β of the belief base B is assigned a random measure r represented by a numeric value. This strategy aims to explore various potential configurations of the belief base B by evaluating the elements β in a stochastic manner. Random measures help identify patterns or solutions that may not be immediately apparent, facilitating the discovery of an optimal solution by assessing how different combinations of elements contribute to the overall random value $\nu_r(HS)$ of the hitting set HS.

In this thesis, the random approach is defined by assigning a unique random measure r in the range between one and the number of elements β in the belief base B, leading to $1 \leq r \leq |B|$. This helps to set a lower and upper limit that does not deviate too much, making this approach comparable to the inconsistency measure approach (see **??**). We will see that the computed inconsistency measures of the knowledge bases we used in this thesis do not deviate much or by a high number between the elements of the belief base B (also see [KT21]). Each random measure r is assigned only once, ensuring uniqueness and achieving better deviation of results.

Definition 16. For each element $\beta_i \in B$, obtain a unique positive random measure $r(i)$ from $r(i) \in \mathbb{R}^+$, where i is the number of the element β_i, such that $r(i)$ is randomly selected within the range from 1 to $|B|$, where $|B|$ is the cardinality of the

belief base B. Each random weight $w_r(\beta_i)$ corresponds to one of the unique positive random measures $r(i)$, i.e. $w_r(\beta_i) = r(i)$. The random value $\nu_r(HS_i)$ is then defined as the sum of the random weights $w_r(\beta_i)$, i.e.,

$$\nu_r(HS_i) = \sum_{i=1}^{|HS_i|} w_r(\beta_i)$$

Example 10. Consider the belief base B from Example 8 with the following unique random measures $r(i)$: $r(1) = 4$, $r(2) = 2$, $r(3) = 1$, $r(4) = 6$, $r(5) = 3$, $r(6) = 5$. Applying the random measures $r(i)$ to the belief base B leads to the following elements β_i with its assigned random weights $w_r(\beta_i)$:

$$B = \{(A0)^4, (\neg A0 \vee A1)^2, (\neg A1 \vee A2)^1, (\neg A0 \wedge \neg A1)^6, (A1 \wedge \neg A2)^3, (\neg A0 \vee \neg A1)^5\}$$

With the assigned random weights w_r, the random value ν_r of a hitting set HS will be calculated by summing the weights $w_r(\beta_i)$ of all elements β_i in the hitting sets HS_i, which leads to the following values $\nu_r(HS_i)$:

$$\begin{aligned}
\mathcal{C}(B) = \quad & \{(A1 \wedge \neg A2)^3, (A0)^4\} && \rightarrow \nu_r(HS_1) = 7 \\
& \{(A1 \wedge \neg A2)^3, (A0)^4, (\neg A0 \vee A1)^2\} && \rightarrow \nu_r(HS_2) = 9 \\
& \{(A1 \wedge \neg A2)^3, (\neg A0 \wedge A1)^6, (\neg A0 \vee A1)^2\} && \rightarrow \nu_r(HS_3) = 11
\end{aligned}$$

The optimal solution set can be found depending on the definition criteria of the optimal solution. If we define the optimal solution to meet a maximality criterion, such that ν_r has to be maximal, meaning that there is no HS_j in the collection $\mathcal{C}(B)$ for which $\nu_r(HS_j) > \nu_r(HS_i)$, then the hitting set HS_i with the highest random value ν_r is determined as the optimal solution. In this case, the third hitting set HS_3 in the collection $\mathcal{C}(B)$, namely $\{(A1 \wedge \neg A2)^3, (\neg A0 \wedge A1)^6, (\neg A0 \vee A1)^2\}$, can be determined as the optimal solution since it has the highest random value, as $\nu_r(HS_1) < \nu_r(HS_2) < \nu_r(HS_3)$.

In contrast, if the optimal solution is defined to meet a minimality criterion, such that ν_r has to be minimal, meaning that there is no HS_j in the collection $\mathcal{C}(B)$ for which $\nu_r(HS_j) < \nu_r(HS_i)$, then the hitting set HS_i with the lowest random value ν_r is determined as the optimal solution. In this case, the first hitting set HS_1 in the collection $\mathcal{C}(B)$, namely $\{(A1 \wedge \neg A2)^3, (A0)^4\}$, can be determined as the optimal solution since it has the lowest random value, as $\nu_r(HS_1) < \nu_r(HS_2) < \nu_r(HS_3)$.

4.3.3. Inconsistency Value

In the third stage, a nuanced approach is adopted through the integration of an inconsistency measure I. This measure assigns a distinct inconsistency measure $I(\beta_i)$ to every element β within the belief base B, enhancing the value-based framework. Utilizing these inconsistency measures $I(\beta_i)$, a refined and precise computation of the optimal solution is achieved, ensuring alignment with the inconsistency

attributes of kernels and remainders. The goal is to identify the hitting set HS_i that contributes the most to the inconsistency of the belief base B.

Inconsistency measurement is a well-explored field of research with a variety of applications. Generally, an inconsistency measure I is a function $I : \mathbb{K} \to \mathbb{R}^{\infty}_{\geq 0}$ [Thi19]. The intuition behind such inconsistency measures I is that a higher value indicates a more severe inconsistency than a lower one. The minimal value 0 represents the absence of inconsistency, i.e., consistency.

The downside of computing inconsistency measures I is that they are generally considered computationally hard [TW19]. However, some research has been conducted in the field of computing contension-based inconsistency measures I_c [GH11, KT21, KGLT23, NKTJ23].

A contension-based inconsistency measure I_c quantifies the degree of inconsistency within a belief base B or knowledge base by evaluating the conflicts among its elements β_i. This approach is grounded in logical frameworks and systematically identifies and measures the extent of contradictory information. The contension-based inconsistency measure I_c can be defined as follows [NKTJ23]:

Definition 17. Let $\tau_3 : \mathit{Var} \to \{T, F, B\}$ be a three-valued assignment following Priest's three-valued logic [Pri79], where T and F correspond to the classical truth values true and false, respectively, and B corresponds to a third, paradoxical truth value, denoted both. The truth order $\prec$ is defined via $F \prec B \prec T$. An assignment τ_3 is extended to arbitrary formulas via:

$$\tau_3(\varphi_1 \wedge \varphi_2) = \min_{\prec}(\tau_3(\varphi_1), \tau_3(\varphi_2)),$$

$$\tau_3(\varphi_1 \vee \varphi_2) = \max_{\prec}(\tau_3(\varphi_1), \tau_3(\varphi_2)),$$

$$\tau_3(\neg T) = F, \quad \tau_3(\neg F) = T, \quad \tau_3(\neg B) = B.$$

An assignment τ_3 satisfies a formula φ, denoted by $\tau_3 \models_3 \varphi$, if either $\tau_3(\varphi) = T$ or $\tau_3(\varphi) = B$.

The contension-based inconsistency measure [GH11] $I_c : \mathbb{K} \to \mathbb{R}^{\infty}_{\geq 0}$ is defined as:

$$I_c(\mathcal{K}) = \min\{|\tau_3^{-1}(B)| \mid \tau_3 \models_3 \mathcal{K}\}.$$

In other words, the contension-based inconsistency measure I_c is the smallest number of atoms $x \in \mathrm{At}(\mathcal{K})$ that need to be set to B to render $\mathcal{K}$ consistent under Priest's three-valued logic.

In this thesis, we used the contension-based inconsistency measure I_c as it provided the best and fastest results in the SAT-solver implementation by Niskanen et al. [NKTJ23], which we used to compute the inconsistency measure I. The inconsistency weight w_I assignment can be defined as follows:

Definition 18. For each element $\beta_i \in B$, assign an inconsistency measure I to an inconsistency weight w_I, where $I \in \mathbb{R}^+$. The inconsistency value $\nu_I(HS_i)$ of a hitting set HS_i is defined as:

$$\nu_I(HS_i) = \sum_{\beta_i \in HS_i} w_I(\beta_i).$$

To obtain the inconsistency weight $w_I(\beta_i)$ for each element β_i in the belief base B, we calculate the initial inconsistency measure $I(B)$ for the entire belief base B using the SAT solver mentioned. Then, for each element β_i in the belief base B, the inconsistency measure $I(B \setminus \beta_i)$ will be calculated, representing the inconsistency measure after removing a specific element β_i. The difference between the initial inconsistency measure $I(B)$ and the calculated inconsistency measure $I(B \setminus \beta_i)$ is computed to determine the impact of each element β_i on the overall inconsistency. The calculation can be defined as follows:

Definition 19. For each element $\beta_i \in B$, the inconsistency weight $w_I(\beta_i)$ will be calculated by subtracting the Contension Inconsistency Measure $I_c(B \setminus \beta_i)$ of $B \setminus \beta_i$ from the initial Contension Inconsistency Measure $I_c(B)$ of B. The contension-based inconsistency weight w_{I_c} is defined as:

$$w_{I_c}(\beta_i) = I_c(B) - I_c(B \setminus \beta_i)$$

In summary, this method systematically evaluates the contribution of each element β in the belief base B to the overall inconsistency, providing a detailed understanding of the dynamics of inconsistency within B.

Example 11. Given the previous Example 8, the initial contension Inconsistency Measure $I_c(B) = 2$ and the calculated contension-based inconsistency weights $w_{I_c}(\beta_i)$ of the elements β_i of B are as follows: $w_{I_c}(\beta_1) = 1$, $w_{I_c}(\beta_2) = 0$, $w_{I_c}(\beta_3) = 0$, $w_{I_c}(\beta_4) = 1$, $w_{I_c}(\beta_5) = 1$, $w_{I_c}(\beta_6) = 0$. The belief base B with its assigned inconsistency weights $w_{I_c}(\beta_i)$ can be visualized as follows:

$$B = \{(A0)^1, (\neg A0 \vee A1)^0, (\neg A1 \vee A2)^0, (\neg A0 \wedge \neg A1)^1, (A1 \wedge \neg A2)^1, (\neg A0 \vee \neg A1)^0\}$$

Given the inconsistency weights $w_{I_c}(\beta_i)$ assigned to the belief base B, the value $\nu_{I_c}(HS_i)$ of the hitting set HS_i is calculated by summing the weights $w_{I_c}(\beta_i)$ of all elements β_i in the hitting set HS_i, which leads to the following values:

$$\begin{aligned}
\mathcal{C}(B) = \quad & \{(A1 \wedge \neg A2)^1, (A0)^1\} && \rightarrow \nu_{I_c}(HS_1) = 2 \\
& \{(A1 \wedge \neg A2)^1, (A0)^1, (\neg A0 \vee A1)^0\} && \rightarrow \nu_{I_c}(HS_2) = 2 \\
& \{(A1 \wedge \neg A2)^1, (\neg A0 \wedge A1)^1, (\neg A0 \vee A1)^0\} && \rightarrow \nu_{I_c}(HS_3) = 2
\end{aligned}$$

In the inconsistency approach, Example 11 provides the same inconsistency values $\nu_{I_c}(HS_i)$ for all hitting sets HS_i. Therefore, each hitting set HS_i can be determined as an optimal solution. As can be seen from the inconsistency weight assignments, this approach is highly dependent on the inconsistency measure I. In Section 4.6.3 we will show an approach that uses the inconsistency values $\nu_{I_c}(HS_i)$ with the cardinality of the hitting sets HS_i to find the optimal solution.

In the literature, various methods are proposed to calculate the random measure of an element in a belief base, such as using the Shapley value [HK10]. Although we considered and implemented the Shapley value calculation, it involves forming coalitions of all elements in the belief base B, which makes it extremely time- and

resource-intensive. Consequently, we have left this approach for further investigation and future work.

To illustrate the dependency on the inconsistency measure I, we have calculated the inconsistency weights $w_I(\beta_i)$ using the so-called problematic inconsistency measure I_p as proposed by Grant and Hunter [GH11] using the implementaion to compute the problematic inconsistency measure I_p from Niskanen et al. [NKTJ23]. The problematic inconsistency measure I_p computes the union of all minimal inconsistent subsets. It contains all formulae in B that are part of at least one inconsistency. Using the problematic inconsistency measure I_p leads to the following problematic-based inconsistency weights $w_{I_p}(\beta_i)$ and values $v(HS_1)$:

Example 12. Consider the belief base B from Example 8 comprise the following elements β and problematic-based inconsistency weights $w_{I_p}(\beta_i)$:

$$B = \{(A0)^1, (\neg A0 \vee A1)^0, (\neg A1 \vee A2)^0, (\neg A0 \wedge \neg A1)^1, (A1 \wedge \neg A2)^1, (\neg A0 \vee \neg A1)^0\}$$

with an initial inconsistency measure $I_p(B) = 6$.

$$\begin{array}{rll} \mathcal{C}(B) = & \{(A1 \wedge \neg A2)^2, (A0)^3\} & \rightarrow \quad \nu_{I_p}(HS_1) = 5 \\ & \{(A1 \wedge \neg A2)^2, (A0)^3, (\neg A0 \vee A1)^1\} & \rightarrow \quad \nu_{I_p}(HS_2) = 6 \\ & \{(A1 \wedge \neg A2)^2, (\neg A0 \wedge A1)^1, (\neg A0 \vee A1)^1\} & \rightarrow \quad \nu_{I_p}(HS_3) = 3 \end{array}$$

In this example, the second hitting set HS_2 in the collection $\mathcal{C}(B)$, namely $\{(A1 \wedge \neg A2)^2, (A0)^3, (\neg A0 \vee A1)^1\}$, can be determined as an optimal solution since this set has the highest value ν based on the sum of the problematic-based inconsistency weights $w_{I_p}(\beta)$ of its elements β.

In Example 10, we can determine the second hitting set HS_2 as the optimal solution, even though this hitting set is not a minimal hitting set. As previously mentioned, the approach of this thesis is to assign weights w_i to the elements of the belief base B and to find an optimal solution based on the value ν_x of the hitting sets. The advantage of this approach is that the weights w_x can be used to implement a B&B strategy that updates its boundary in dependence on a computed hitting set. We assume that with the B&B strategy, we do not have to compute all hitting sets, because this strategy allows us to prune all branches as soon as the optimal solution is found, making the determination of minimal hitting sets unnecessary. The B&B strategy and the underlying logic will be explained in the next section.

4.4. Branch-and-Bound Framework

The weight assignment described in the previous Section 4.1 allows us to implement a B&B strategy that can be used when constructing the hitting set tree to limit the search space and find the optimal solution faster. In this section, we will describe the B&B approach and search strategies used to span the tree. We will see that developing or utilizing an improved search strategy can be beneficial for setting the boundary and pruning branches more efficiently.

4.4.1. Theoretical Background

As we have briefly described in Section 3.5, the B&B framework is a fundamental methodology for solving NP-hard optimization problems that was defined by Morrison et al. [MJSS16] as follows:

Definition 20. Let $\mathcal{P} = (\mathcal{H}, g)$ be an optimization problem, where $\mathcal{C}$ is the collection of hitting sets HS, and $g : \mathcal{H} \to \mathbb{R}$ is the objective function. The goal is to find an optimal hitting set $HS^* \in \arg\min_{HS \in \mathcal{H}} g(HS)$. To solve $\mathcal{P}$, the B&B algorithm iteratively builds a hitting set tree HST composed of subproblems, which are subsets of the hitting set HS. Let $\hat{HS} \in \mathcal{H}$ be a feasible hitting set, known as the possible solution, which can be globally stored.

After a B&B algorithm found a possible solution $\hat{HS}$, it proceeds with the next iteration and selects a new subproblem $S \subseteq \mathcal{H}$ from a list L of unexplored subproblems. If a possible solution $\hat{HS}' \in S$ is found with a better value $\nu(\hat{HS})$ than $\hat{HS}$ (i.e., $g(\hat{HS}') < g(\hat{HS})$), the possible solution $\hat{HS}$ is updated. If it can be proven that no hitting set in S has a better value $\nu(\hat{HS})$ than $\hat{HS}$ (i.e., $\forall HS \in S, g(HS) \geq g(\hat{HS})$), the subproblem S is pruned and considered terminal. Otherwise, child subproblems are generated by partitioning S into an exhaustive (but not necessarily mutually exclusive) set of subproblems $S_1, S_2, \ldots, S_r$, which are then inserted into HST.

The process continues until no unexplored subproblems remain in L. The best possible solution $\hat{HS}$ is then returned. Since subproblems are only pruned if they contain no solution better than $\hat{HS}$, it must be the case that $\hat{HS} \in \arg\min_{HS \in \mathcal{H}} g(HS)$, meaning that the last found solution can be determined as the optimal solution.

In this thesis, we defined the objective function g in different ways, depending on the formulation of the optimal solution. In the cardinality approach the objective function g was set to $g \to \nu(HS)$. In the random and inconsistency approach, the objective function was set to $g \to \nu(B) - \nu(HS)$, where $\nu(B)$ represents the value of the belief base B. Both approaches will be explained in the following sections.

4.4.2. Phases of B&B Algorithm

According to [MJSS16], three components significantly impact B&B performance: the search strategy (order of exploring subproblems), the branching strategy (how the solution space is partitioned), and the pruning rules (rules to prevent exploring suboptimal regions). The search strategies used in this thesis will be described in the following Section 4.5.

The research of this thesis focused on improving these components to enhance B&B performance. This section surveys modern advances in B&B theory, especially concerning hitting set trees. Additionally, it highlights three research directions:

1. Developing new search strategies for finding optimal solutions faster,
2. Analyzing and innovating branching/kernel computing strategies,

3. Creating rules for pruning branches and boundary management.

The branching strategy used in a hitting set tree HST depends on the computation of the kernels k_i. In our approach, we branch the hitting set tree HST based on the number of elements in a kernel k, as described in Section 3.3. The pruning rules will be covered in the subsequent sections and in the description of the implementation in Section 5.

In any B&B algorithm, there are two crucial phases. The first is the search phase, where the algorithm has not yet found an optimal solution $\hat{HS}$. The second phase is the verification phase, where a potential solution is considered optimal, but there are still unexplored subproblems that have not been pruned. It's important to note that a potential solution cannot be confirmed as optimal until all subproblems have been explored or specific criteria are met that establish the solution's optimality. For example, a solution can be deemed optimal if it reaches a minimum value, even if higher values may exist in the unexplored search space. This definition of optimality could be outlined as: meeting a minimum value in the minimum amount of time. Thus, the potential solution can be marked as optimal. Furthermore, the transition from the search phase to the verification phase is only known upon the algorithm's termination. In this thesis, we verify the optimal solution found by our B&B algorithm by comparing it to the optimal solution obtained without using B&B or any pruning strategies. When no B&B strategy is used, the optimal solution is determined by comparing the values ν of all found hitting sets.

The three key components of B&B algorithms, namely search strategy, branching strategy, and pruning rules, each play distinct roles in these phases. Figure Fig. 3 illustrates the transition from the search phase to the verification phase, highlighting the role of search strategy, branching strategy, and pruning rules.

The choice of search strategy primarily affects the search phase of the B&B algorithm. During this phase, the algorithm actively explores subproblems to find an optimal solution. For instance, if pruning rules depend on the value of a possible solution (e.g., comparing the upper bound to the hitting set value), the search strategy must eventually explore the same set of subproblems once an optimal solution is identified.

While the search strategy is crucial during the initial search for the optimal solution, its significance diminishes once the solution is found. In our approach, after identifying an optimal solution HS^* with our B&B algorithm, we run the algorithm without B&B to explore all subproblems. Notably, the B&B algorithm could verify its result in the same run by saving pruned branches in a queue and exploring this queue of unexplored subproblems once the optimal solution is found. However, we chose to separate these runs for a clearer distinction and because we needed to run the algorithm without B&B for performance evaluation. As previously mentioned, the search strategy does not impact the algorithm's performance when all subproblems must be explored. Therefore, we did not implement different search strategies for our verification.

Pruning rules often target the verification phase, especially when objective-based

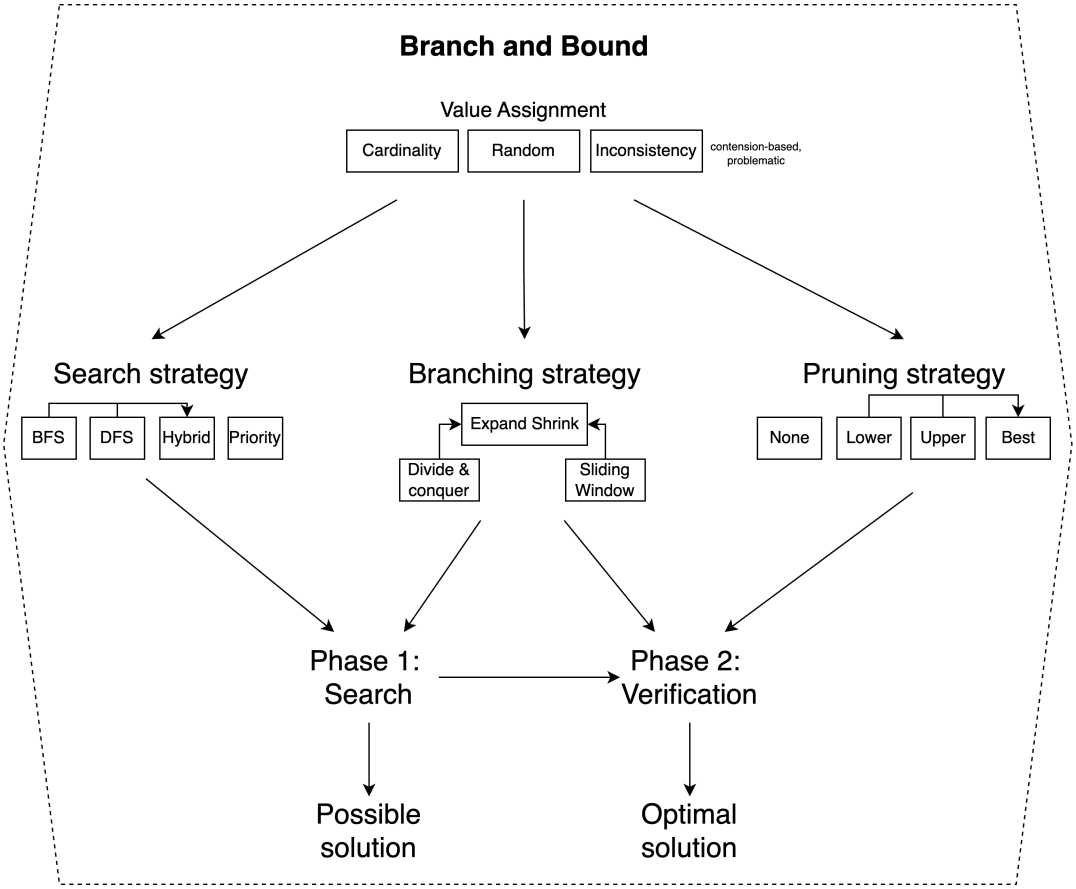


Figure 3: Phases of the Branch-and-Bound Algorithm

bounding is used. These bounds can be weak before finding an optimal (or near-optimal) solution, making early pruning ineffective if the possible solution's objective value is poor. However, pruning rules can also contribute to the search phase, where the pruning rules could incorporate the tree level or number of branches.

The branching strategy impacts both phases significantly. During the search phase, it guides the algorithm towards optimal solutions. In the verification phase, it helps limit branching decisions, preventing unnecessary work to find the optimal solution.

Improving B&B performance during the search phase is crucial for two reasons. First, if the algorithm terminates without finding an optimal solution (e.g., no hitting set meets the requirements), it can still return a potential solution, which can be verified afterwards. This is particularly useful in scenarios where no weight assignment was done, and therefore, no pruning occurs.

Secondly, identifying an optimal solution early in the search phase significantly reduces the size of the search tree and the time required to find the optimal solution, as further nodes with bounds greater or lower than the optimal solution do not

need to be explored. This rationale underpins Dechter and Pearl's [DP85] finding that best-first search (priority) explores the fewest subproblems compared to other search strategies. We will verify this in Section 6, where we compare the mentioned search algorithms in terms of execution time and the number of pruned branches.

However, recent literature contains relatively few studies examining the impacts of search and branching strategies on the performance of B&B algorithms. Instead, most research focuses on pruning rules, which are particularly beneficial during the verification phase. Consequently, the development of more advanced search strategies, branching strategies and a framework for efficient pruning are identified as two critical research directions of this thesis. In the next section, we will explore the implemented search strategies as illustrated in Figure 3.

4.5. Search strategies

In B&B algorithms, search strategies play a critical role in determining the order in which subproblems are explored. These strategies significantly impact the efficiency of the algorithm by influencing the pruning process and the path to the optimal solution. Here, we discuss four search strategies: Breadth-First Search (BFS), Depth-First Search (DFS), Hybrid Search (HYS) and Priority-Based Search (PBS), including their strengths and weaknesses. Figure 4 illustrates a small search tree and the order in which nodes are explored under different search strategies. The (blue) elements of B indicate the edges with their respective weight. The numbers outside the nodes on the left side indicate exploration order. To provide a concrete example, we assume the following belief base B comprise the following elements and inconsistency weights w_i:

$$B = \{(A0)^1, (\neg A0 \vee A1)^4, (\neg A1 \vee A2)^0, (\neg A0 \wedge \neg A1)^1, (A1 \wedge \neg A2)^1, (\neg A0 \vee \neg A1)^0\}$$

4.5.1. Breadth-First Search

Breadth-First Search (BFS) explores all subproblems at the current depth before moving to the next level. This strategy uses a queue to manage unexplored subproblems, ensuring that the shallowest nodes are explored first. BFS is particularly effective in finding solutions close to the root of the tree and works well with unbalanced trees.

However, BFS has high memory requirements because it needs to store all subproblems at each level of the tree. This makes it less practical for B&B algorithms where memory efficiency is crucial. BFS is rarely used in B&B contexts except in specific scenarios, such as when dominance relations can prune subproblems effectively or when a good heuristic solution is available early on.

Additionally, the B&B implementation in this thesis updates the boundary only when a leaf node, representing a potential solution, is found. When using BFS, this

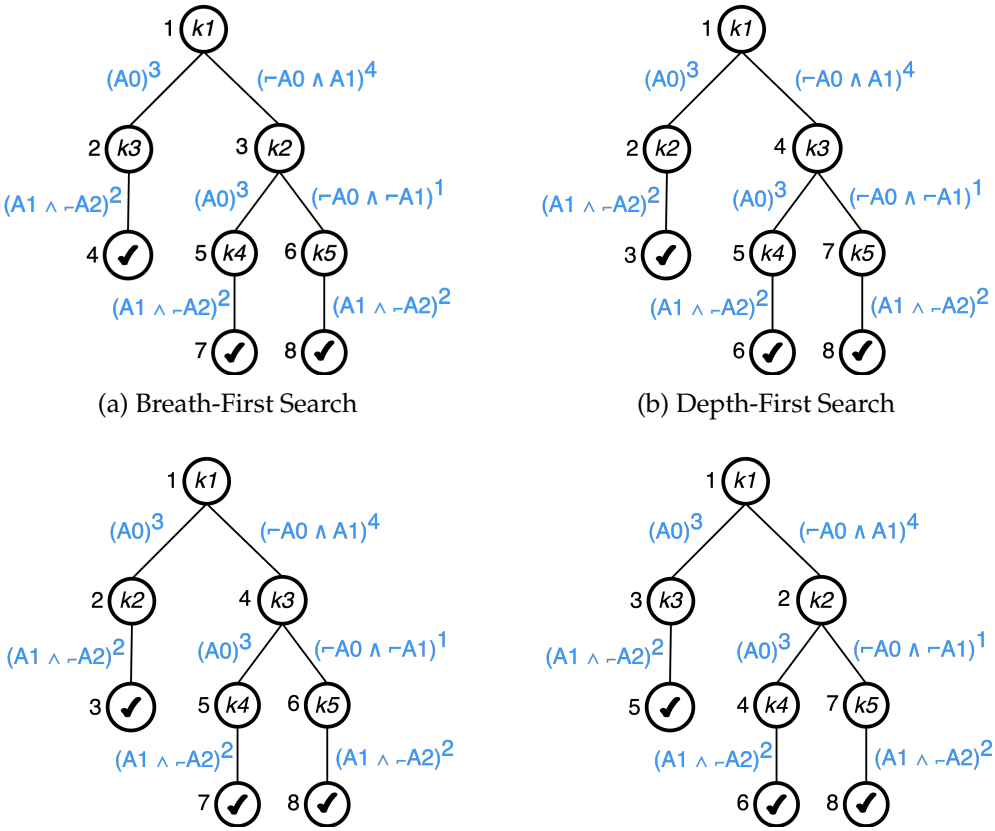


Figure 4: Comparison of different search strategies using random measures

can result in a delayed boundary update, leading to the expansion of many unnecessary branches. These branches might have been pruned earlier if the boundary had been updated sooner.

4.5.2. Depth-First Search

Depth-First Search (DFS) is a strategy that explores as far down a branch of the search tree as possible before backtracking. This method can be implemented by maintaining a stack of unexplored subproblems. At each step, the algorithm selects the subproblem at the top of the stack, explores it, and then pushes its children onto the stack. This means the next subproblem explored is always the most recently generated one.

One significant advantage of DFS is its low memory requirement. Since it only needs to store the current path from the root to the current node and the indices of the last explored children, it avoids the need to keep the entire list of unexplored subproblems in memory. This can be particularly beneficial when the search space

is vast.

However, DFS can encounter issues such as thrashing and inefficiency in unbalanced trees. Thrashing occurs when different regions of the search space fail for similar reasons, causing the algorithm to waste time. In unbalanced trees, DFS might explore many long, unproductive paths before finding an optimal solution.

While the strategy of quickly expanding to every leaf node can be beneficial for updating the boundary once a leaf is found, it can also result in a significant number of unnecessary expanded branches in the case of an unbalanced tree.

4.5.3. Hybrid Search

A hybrid search (HYS) strategy combines elements of both Depth-First Search (DFS) and Breadth-First Search (BFS) to leverage the advantages of both methods. This approach begins with DFS to quickly reach a leaf node and potentially find a possible solution early, to set a boundary that can be used in the pruning logic when expanding the following branches. Once a leaf node is reached, the strategy switches to BFS to systematically explore the remaining subproblems level by level.

The hybrid strategy starts with DFS, which is implemented by maintaining a stack of unexplored subproblems. The algorithm repeatedly selects the subproblem at the top of the stack, explores it, and pushes its children onto the stack. This continues until a leaf node is found. The main benefit of starting with DFS is to quickly find a possible solution by exploring a path down to a leaf node, resulting in a fast update of the boundary.

Once a leaf node is reached and an initial solution is found, the strategy switches to BFS to explore the remaining subproblems more systematically. This phase involves maintaining a queue of unexplored subproblems. The algorithm then explores all subproblems at the current depth level before moving to the next level. The advantages of switching to BFS include:

- **Systematic Exploration**: BFS ensures that all subproblems at the same depth are explored before deeper levels (except the first found leaf via DFS), which can help in finding the optimal solution more efficiently in an unbalanced tree.
- **Better Pruning Opportunities**: By exploring level by level, BFS can make better use of pruning rules, especially when compared against the current best solution found during the DFS phase.
- **Avoiding Deep Unproductive Paths**: BFS prevents the algorithm from getting stuck in long, unproductive paths that might occur with DFS alone.

By combining the depth-first and breadth-first approaches, this hybrid strategy aims to balance early solution discovery with systematic exploration, optimizing both memory usage and search efficiency.

4.5.4. Priority-Based Search

In this thesis, a Priority-Based Search (PBS) strategy was developed, which implements a Best-First Search approach. The PBS strategy selects the next subproblem according to a path value from a node to the root, calculated as the sum of the assigned weights w of the elements β along the branches. The path value can be defined as follows:

Definition 21. Let $\mathcal{S}$ be the set of all paths P from the root node v_{root} of the hitting set tree HST to any node v. The associated weight of an element β_i is denoted w as $w(\beta_i)$. The path value, denoted as $\nu(P)$ is defined as the sum of the weights w of all elements along the path P:

$$\nu(P) = \sum_{\beta_i \in P} w(\beta_i)$$

where β_i are the elements in the path P and $w(\beta_i)$ is the weight of the element β_i.

The priority-based search strategy uses this path value $\nu(P)$ to prioritize nodes that appear most promising for further exploration. This prioritization ensures that nodes appearing most promising are explored first. As a result, the PBS strategy allows for efficient exploration of the search space, often identifying good solutions earlier in the search process.

The PBS strategy operates by using a heap data structure, where the path values $\nu(P)$ serve as keys. This enables the algorithm to always expand the subproblem with the highest priority first. By prioritizing branches based on their path value $\nu(P)$, PBS can effectively direct the search towards more promising regions of the search space, improving the chances of finding an optimal solution quickly.

One of the primary advantages of the PBS strategy is its ability to dynamically prioritize branches, thus not being confined to one specific branch of the tree. This flexibility allows the algorithm to adaptively explore different areas of the search space based on the highest values serving as the best estimates of their potential, leading to more efficient pruning of suboptimal branches. The advantages of the PBS include:

- **Early Detection of Good Solutions**: By focusing on the most promising branches first, PBS often identifies high-quality solutions early in the search process, which can be used to update the boundary and prune less promising branches.
- **Adaptive Exploration**: PBS dynamically adjusts its focus based on the current state of the search, allowing it to efficiently navigate through the search space and avoid spending excessive time on less promising areas.
- **Efficient Use of Resources**: Using a heap to manage subproblems ensures that the algorithm operates efficiently, both in terms of time and memory. The

priority-based selection helps in reducing the number of subproblems that need to be explored.

- **Improved Pruning**: By updating the boundary with better solutions found early, PBS enhances the pruning process, eliminating many suboptimal branches and reducing the overall search space.

As previously discussed, the search strategy is closely intertwined with boundary management and pruning logic. These elements, which will be detailed in the following section, facilitate faster exploration of promising subproblems, thereby enabling more efficient attainment of the optimal solution.

4.6. Boundary Management and Pruning

In the context of B&B algorithms, effective boundary management and pruning are crucial for optimizing the search process and enhancing algorithm efficiency. A key strategy involves strategically setting or updating bounds to achieve the desired goal.

The boundary is calculated by determining the path value $\nu(P(v))$ from the root v_{root} to a given node v. In our implementation (see Section 5) we calculate the path value $\nu(P(v))$ using backtracking from the node v to the root v_{root}. The path value $\nu(P(v))$ is a critical metric used to evaluate subproblems. The cumulative path value $\nu(P(v))$ represents the cost or boundary of the subproblem that can be used to set an upper bound. The bound is defined as follows:

Definition 22. Let $\mathcal{B}$ be the bound that a subproblem at node v in the hitting set tree needs to meet. This bound can either be an upper bound or a lower bound. The value of the bound, denoted $\nu(P(v))$, is determined by the sum of the weights w of the elements along the path $P(v)$ from the root v_{root} to node v. Whether the subproblem needs to satisfy or not exceed this bound depends on the specific criteria defined for upper and lower bounds.

In a B&B algorithm, the boundary acts as a critical threshold or benchmark value that subproblems must meet or exceed to be considered for further exploration. It serves as a reference point against which potential solutions are compared, helping to identify and eliminate subproblems that cannot yield a better solution than the best one found so far.

The comparison of the subproblems to the boundary depends on the objective function g that calculates a potential value $\nu_P(P(v))$ defined as follows:

Definition 23. Let ν_P be the potential value of a path $P(v)$ for node v. The potential value ν_P is calculated as follows:

$$\nu_P(P(v)) = \nu(P(v)) + g(v),$$

where

$$g(v) = \sum_{\beta_i \in B \setminus P(v)} w(\beta_i),$$

and $w(\beta_i)$ represents the weight of the element β_i in the set B.

In other words, the potential value ν_P is calculated by adding the sum of all weights w of elements $\beta \notin P(v)$ from node v to the root node v_{root} to the path value $\nu(P(v))$.

Using our weight assignment approach, which assigns a weight w to every element β of the belief base B, we can calculate various values to achieve efficient pruning. This weight assignment also enables the computation of the value of the belief base B, allowing us to determine maximum values $\nu_{\max}(P(v))$ and minimum values $\nu_{\min}(P(v))$ that are complementary. These values are defined as follows:

$$\nu_{\max}(P(v)) = \nu(P(v)) + \nu(B \setminus P(v))$$
$$\nu_{\min}(P(v)) = \nu(B) - \nu(P(v))$$

Both values $\nu_{\max}(P(v))$ and $\nu_{\min}(P(v))$ can be compared to an upper bound or a lower bound. Due to their complementary nature, we are able to treat the upper bound as a lower bound and vice versa. We will also demonstrate that a combined approach can be beneficial in finding optimal solutions, particularly when using inconsistency weights w_I.

This thesis employs three distinct pruning approaches to efficiently eliminate suboptimal subproblems from the search space. These approaches include an upper pruner, a lower pruner, and a best pruner, all of which will be described in the following sections.

4.6.1. Upper Pruner

The upper pruning employs a lower bound $\mathcal{B}_L$, so that we are identifying subproblems in the search space $\mathcal{S}$ that do not meet a certain threshold. A lower bound $\mathcal{B}_L$ represents the lowest value of the objective function g found so far during the search and can be applied if the optimal solution includes a minimal criterion, leading to a minimum threshold to be reached for each possible solution within the search space. The lower bound $\mathcal{B}_L$ can be defined as:

Definition 24. A lower bound $\mathcal{B}_L$ for a problem within the search space $\mathcal{S}$ is defined as the minimum value that any subproblem must not exceed. Formally, for a given subproblem $P \in \mathcal{S}$, if the evaluation of P results in a value greater than $\mathcal{B}_L$, then P and all its subsequent branches can be pruned from the search process.

By setting this lower bound $\mathcal{B}_L$, we can prune the search process by eliminating the computation of any subproblems along branches that do not meet the specified threshold. This effectively reduces the computational effort by focusing only on

branches that have the potential to meet or be lower than the threshold. Initially, the lower bound $\mathcal{B}_L$ is set to a value that ensures that all subproblems are explored until a leaf node v_{leaf} is found. The lower bound $\mathcal{B}_L$ is updated if a hitting set HS is found by reaching a leaf node v_{leaf}. The boundary is updated to the potential value $\nu_P(v_{\text{leaf}})$ of the leaf node $\nu_P(v_{\text{leaf}})$ if it is less than the existing lower bound $\mathcal{B}_L$.

Formally, the lower bound $\mathcal{B}_L$ is updated as follows:

$$\mathcal{B}_L = \begin{cases} \nu_P(P(v)) & \text{if } \nu_P(P(v)) \leq \mathcal{B}_L \\ \mathcal{B}_L & \text{otherwise} \end{cases}$$

where $\nu(P(v))$ represents the cumulative path value from the root v_{root} to node v, and $\mathcal{B}_L$ represents the current boundary.

Therefore, the upper pruner prunes all subproblems that exceed the lower bound $\mathcal{B}_L$, leading to finding an optimal solution where $\nu(P(v))$ is minimal. This means that there is no other $\nu(P(v')) < \nu(P(v))$.

4.6.2. Lower Pruner

This section covers the fundamentals of lower pruning, introducing an upper bound $\mathcal{B}_\mathcal{U}$, which identifies subproblems in the search space $\mathcal{S}$ that do not meet or exceed a certain threshold. An upper bound $\mathcal{B}_U$ represents the highest value of the objective function g found so far during the search and can be applied if the optimal solution includes a maximal criterion, leading to a maximum threshold to be reached for each possible solution within the search space. The upper bound $\mathcal{B}_U$ can be defined as:

Definition 25. An upper bound $\mathcal{B}_U$ for a problem within the search space $\mathcal{S}$ is defined as the maximum value that any subproblem must not exceed. Formally, for a given subproblem $P \in \mathcal{S}$, if the evaluation of P results in a value lower than $\mathcal{B}_U$, then P and all its subsequent branches can be pruned from the search process.

Initially, the upper bound $\mathcal{B}_\mathcal{U}$ is set to a value that ensures that all subproblems are explored until a leaf node is found. This value can be a maximum value, e.g., $\mathcal{B}_U = \max(\nu_P(P(v)))$ or, as part of a straightforward implementation, $\mathcal{B}_U = \infty$. The upper bound $\mathcal{B}_U$ is updated if a hitting set HS is found by reaching a leaf node v_{leaf}. The upper bound $\mathcal{B}_\mathcal{U}$ is updated to the potential value $\nu_P(v_{\text{leaf}})$ of the leaf node $\nu_P(v_{\text{leaf}})$ if it is greater than the existing upper bound $\mathcal{B}_U$. This can be formalized as follows:

$$\mathcal{B}_\mathcal{U} = \begin{cases} \nu_P(P(v)) & \text{if } \nu_P(P(v)) \geq \mathcal{B}_\mathcal{U} \\ \mathcal{B}_\mathcal{U} & \text{otherwise} \end{cases}$$

where $\nu(P(v))$ represents the cumulative path value $\nu(P(v))$ from the root v_{root} to node v, and $\mathcal{B}_\mathcal{U}$ represents the current boundary.

Therefore, the lower pruner prunes all subproblems that do not meet the upper bound $\mathcal{B}_U$, leading to finding an optimal solution where $\nu(P(v))$ is minimal. This means that there is no other $\nu(P(v')) < \nu(P(v))$. The computation of an optimal solution HS^* can be very hard, as in the worst case the whole search space has to be expanded. This is because the lower pruner might prune branches of the tree that would have contained a subproblem with $\nu(\hat{HS'}) > \nu(\hat{HS})$. This is because the search algorithm has no information about the unexplored subproblems.

This challenge can be faced by a heuristic to compute the possible value of the subproblems. In this thesis we have implemented a computation of the possible solution of unexplored subproblems based on a remainder value.

Remainder Value Computation This approach finds a remainder R of an unexplored node v' and computes the potential value $\nu_P(P(v'))$. As a result a part of improving the calculation of the potential value $\nu_P(P(v))$, we have developed a heuristic to calculate the potential value $\nu_P(P(v))$ based on the value of a remainder R of the subproblem. The computation of remainders will be covered in Section 5. With this approach we are calculating the potential value ν_P as follows:

$$\nu_P(P(v)) = \nu(B(v)) - \nu(R),$$

wherein $B(v)$ represents the dataset at node v, that can be determined by $B(v) = B \setminus P(v)$.

The value $\nu(R)$ of the remainder represents the value of a hitting set HS of a node v, since $B \setminus R = HS$. Figure 5 illustrates an example of this approach, where a hitting set tree HST is illustrated with a four kernels and an optimal hitting set HS^* with value $\nu(HS^*) = 9$.

The HST is search using the Priority-Based Search such that the algorithm finds the optimal hitting set HS^* with the first computed leaf node v_{leaf} at node v_4. For the remainder of this example we denote v_i as a node with the index i that represents an order in which the nodes v_i are explored. The path $P(v_4)$ to node v_4 is marked in green. Now the algorithm continues to search for better solutions, as it has no information whether the possible solution $\hat{HS}$ is the optimal solution HS^*. When the search arrives at node v_4 with kernel $k4$ (comprising three elements), the pruner calculates the objective function $g(v)$ by the sum of the weights w of all elements $\beta \in B \setminus P(v)$, therefore $g(v_7) = 14$. As $g(v_7) > \nu(HS^*)$, the lower pruner decides not to prune the branch and to expand the children of the node v_7.

Now with the remainder value computation, the algorithm first computes a remainder $R(v_4)$ of $B(v_7) \setminus \hat{HS}$ and find $R(B(v_7) \setminus \hat{HS}) = \{A1 \wedge A2, A2 \vee \neg A1, A0, A1 \wedge A2\}$ with a remainder value $\nu(R) = 10$. If the algorithm now computes the potential value $\nu_P(P(v_7)) = \nu(B(v_7) \setminus \nu(R_7)) = 4$, the pruner can prune the branch at node v_7, because the subproblem cannot yield to a better solution, hence $\nu(\hat{HS}') \leq \hat{HS}$ holds.

With this approach we are eliminating the subproblems that have the biggest remainder R. This approach is at least worth exploring, especially for situations where

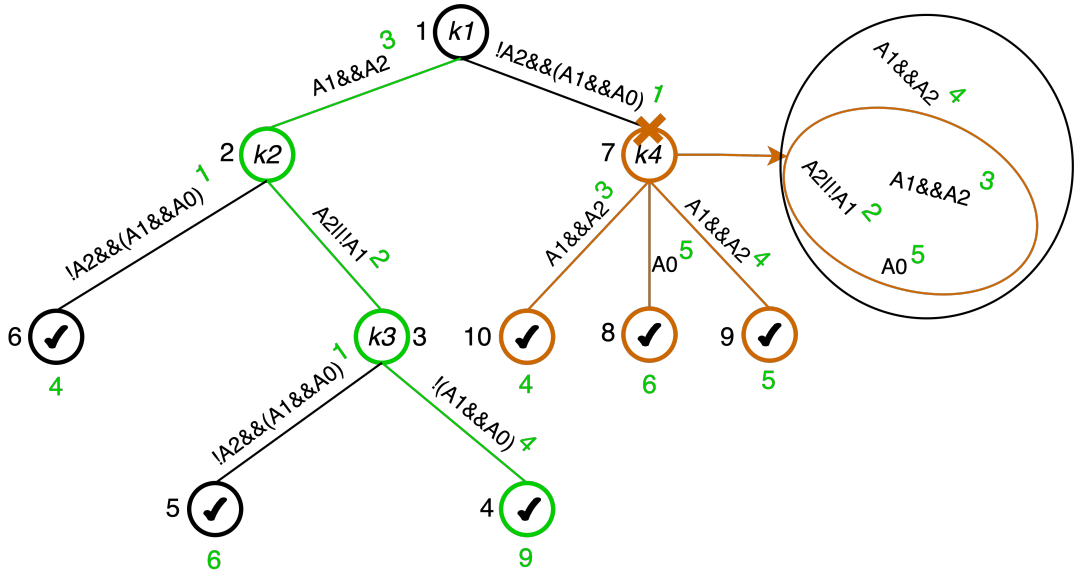


Figure 5: Exemplary Search Tree with Best Pruner

the elements β of B have even weights w. This heuristic is based on the entailment check of alpha α for maximal consistent subsets (remainders R) instead of computing maximal inconsistent subsets (kernels k). We assume that this approach can be faster than computing the search tree solely based on the computation of kernels, as Resina et al. [RRW14] showed that computing remainders R can outperform the computation of kernels k.

It is important to note that this approach only computes one remainder R, although there might be another remainder R' at node v_i. Nonetheless, this approach adds another layer to the computation of the objective function $g(v)$ that might yield to a higher number of pruned branches and therefore to less computation time. We will evaluate this approach in Section 6.4.1. In the next section, we are going to describe another pruning approach, the best pruner.

4.6.3. Best Pruner

The best pruner approach implements a best bound $\mathcal{B}_B$ that combines the principles of the upper bound $\mathcal{B}_U$ as described in Section 4.6.2 and the lower bound $\mathcal{B}_L$ as described in Section 4.6.1. The best pruner differs in two key aspects from the upper pruner and lower pruner. Firstly, its main contribution is the calculation of a unique potential value ν_P that is compared to each bound separately, resulting in a lower potential value ν_{P_L} and an upper potential value ν_{P_U}. Secondly, it incorporates a sequential order of checking both bounds $\mathcal{B}_U$ and $\mathcal{B}_L$.

The pruning logic of the best bound $\mathcal{B}_B$ can be formalized as follows:

$$\mathcal{B}_\mathcal{B} = \begin{cases} \nu_{P_L} & \text{if } \nu_P(P(v)) \leq \mathcal{B}_L \\ \nu_{P_U} & \text{if } |P(v)| \geq \mathcal{B}_U \\ \mathcal{B}_\mathcal{B} & \text{otherwise} \end{cases}$$

where ν_{P_L} represents the cumulative path value from the root v_{root} to node v, $|P(v)|$ represents the cardinality of $P(v)$, and $\mathcal{B}_B$ represents the current best bound.

The best bound $\mathcal{B}_B$ allows us to find the optimal hitting set HS^*, defined as having the highest value $\nu(HS^*)$, such that no $\nu(HS)$ exceeds $\nu(HS^*)$, and the lowest cardinality $|HS^*|$, such that no $|HS|$ is smaller than $|HS^*|$. This approach is particularly beneficial when using inconsistency weights w_I, as it includes elements β with $w_I = 0$. This enables us to find the optimal hitting set HS^* that maximizes participation in the inconsistency of the belief base B, while retaining the highest possible degree of information when repairing the belief base B. Hence, $I(B \setminus HS^*) = 0$ and $|B \setminus HS^*|$ is maximal.

The best pruner demonstrates the main benefit of our weight assignment strategy, especially when using inconsistency weights w_I. With the assigned inconsistency weights w_I, we assume that the optimal hitting set HS^* cannot have a higher inconsistency weight w_I compared to the sum of all inconsistency weights w_I. Therefore, the following applies:

$$\nu_I(HS^*) \leq \sum_{HS_i \in B} \nu_I(HS_i)$$

Although it might be simpler to summarize all weights w_I, it is also possible to use the initial inconsistency measure $I(B)$. This approach allows us to determine the maximum value of the optimal solution before expanding the search space. It enhances computation time, as the best pruner eliminates all subproblems that already meet $\mathcal{B}_L = I(B)$. This method provides a valid solution to compute the optimal hitting set HS^* with the highest value, thus solving a maximality problem in a very short time.

Figure 6 shows an exemplary tree with five hitting sets HS_i. In the tree of Figure 6 we have applied the weight assignment strategy based on inconsistency weights w_I, so that we know that $\nu_I(HS^*) = 3$. With this information, we are exploring the search tree and finding two hitting sets at the leaf nodes at the kernels $k3$ and $k6$ with $\nu_I(HS) = 2$. Knowing that there might be a hitting set HS with a higher $\nu_I(HS)$, the search continues and finds another hitting set at kernel $k9$. Knowing that we have reached the maximal value ν_{max}, the best pruner can now prune all branches that have a higher cardinality than the hitting set HS at kernel $k9$. This allows us to prune the next branch prior to computing kernel $k10$, which ends the search. With this approach, we needed to compute nine kernels k to find the optimal hitting set HS^*, as opposed to computing 14 kernels. In Figure 6, the hitting sets HS with the highest value $\nu(HS)$ are marked with a green checkmark. In this example,

the four marked hitting sets all have a value $\nu_U(HS) = 3$. With the best pruner, we can navigate the tree along the green path to find the optimal solution HS^* after finding two other possible solutions $\hat{HS}$ that might not meet the criteria.

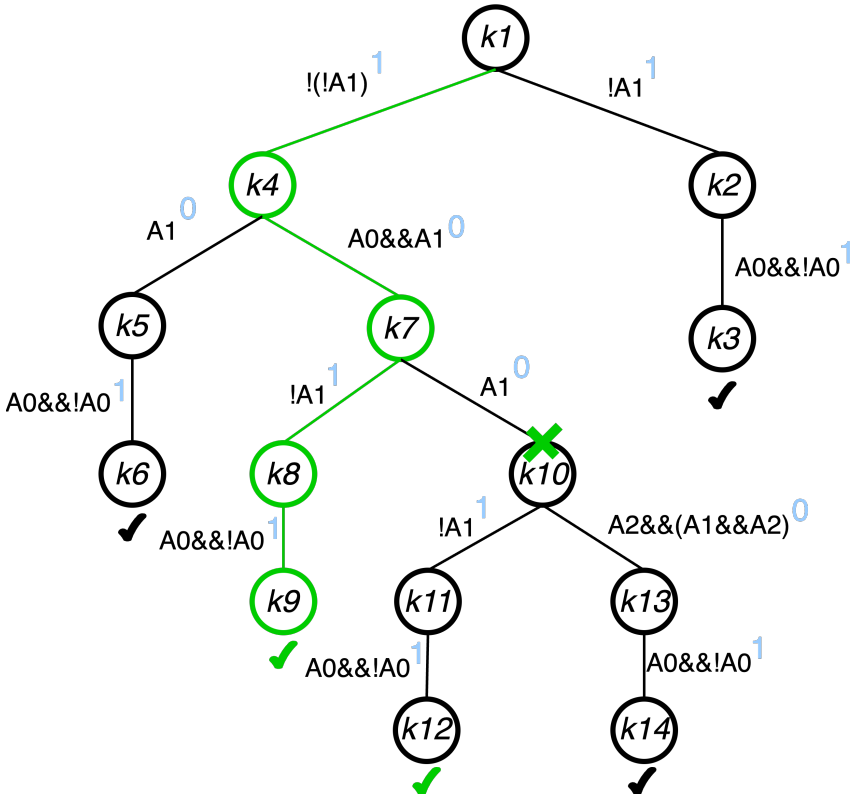


Figure 6: Exemplary Search Tree with Best Pruner

Using an upper bound $\mathcal{B}_U$ and a lower bound $\mathcal{B}_L$ in the Branch-and-Bound (B&B) approach offers several advantages:

- **Efficient Pruning**: By continually updating the upper bound $\mathcal{B}_U$ and lower bound $\mathcal{B}_L$ with better solutions, the algorithm can prune a larger number of suboptimal subproblems, thereby reducing the search space and computational effort.
- **Focused Search**: The boundaries direct the search towards more promising regions of the search space, improving the likelihood of finding the optimal hitting set HS^* more quickly.
- **Memory Optimization**: Pruning subproblems early based on the upper bound $\mathcal{B}_U$ and lower bound $\mathcal{B}_L$ reduces the number of subproblems stored in memory, optimizing resource usage.

- **Faster Convergence**: Updating the boundaries with better solutions found early in the search helps in pruning more subproblems, leading to faster convergence to the optimal hitting set HS^*.

In summary, the combination of boundary management and pruning rules based on the potential value ν_P calculation is integral to the effectiveness of the B&B strategy implemented in this thesis. By dynamically adjusting the boundary and aggressively pruning suboptimal branches, this approach efficiently narrows down the search space, enhancing both speed and accuracy in finding the optimal hitting set HS^*.

5. Implementation

In this chapter, we provide a thorough description of the implementation details of our application, including a weight assignment strategy, an advanced belief change algorithm, and an advanced hitting set tree algorithm with different search strategies. In particular, we describe the application's architecture, its main algorithm schemes and a heuristic to find and compute an optimal solution for belief base contraction.

5.1. Application Architecture

Building on the approach and strategies discussed in Section 4, we present a detailed system architecture, outlining the necessary components and their interconnections.

Our application for computing a hitting set tree is implemented in Python and leverages the SAT solver sat4im[1] provided by Niskanen et al. [NKTJ23], along with the MiniSat SAT solver. The installation of the MiniSat SAT solver is a prerequisite for running the application on any given system. For conducting the Tseitin transformation, we utilize the implementation provided by Niskanen et al. [NKTJ23] in `sat4im/src/core.py`. The application's source code is publicly available on GitHub[2]. Links to the download pages of the required solvers can be found in the repository's README file.

The application is started with a set of mandatory and optional input parameters. Table 2 provides an overview of these input parameters, where "M" denotes mandatory parameters and "O" denotes optional parameters.

	Parameter	**Input**	**Description**
M	{filepath}	{'STR'}	Path to the dataset file as string
M	--vp	{1,2,3}	Strategy parameter value 1, 2, 3
M	--ss	{BFS,DFS,HYB,PBS}	Search strategy to use
M	--alpha	{'STR'}	A string value for α as string
O	--expand	{div-conq} or {sw-size} {INT}	Divide and conquer or sliding window with window size
O	--shrink	{div-conq} or {sw-size} {INT}	Strategy for shrink sliding window with window size
O	-res-db		Save results to database
O	-no-log		Disable logging
O	-path-db		Flag to call file from db

Table 2: Input Parameters

In general, the parameters are designed such that a double dash (--) indicates a parameter that requires additional input, while a single dash (-) signifies a flag

[1] https://bitbucket.org/coreo-group/sat4im/src/master/
[2] https://github.com/SebastianMueller41/Masterthesis

that toggles a function on or off. Parameters enclosed in braces require input, which could be a file path, an integer value or one of the specified options.

The application supports four distinct strategy run modes, specified by the `--vp` input:

- **1**: Assignment of cardinality weights w_c, as detailed in Section 4.3.1.
- **2**: Assignment of random weights w_r, described in Section 4.3.2.
- **3**: Assignment of calculated inconsistency measures w_I, as outlined in Section **??**.

The mandatory input parameter `--ss` indicating the search strategy supports the following four inputs for selecting the search strategy that will be used to span the hitting set tree:

- **BFS**: Breadth-First Search, as detailed in Section 4.5.1.
- **DFS**: Depth-First Search, as detailed in Section 4.5.2.
- **HYS**: Hybrid Search, as detailed in Section 4.5.3.
- **PBS**: Priority-Based Search, as detailed in Section 4.5.4..

The mandatory parameter `--alpha` specifies the element to be checked for entailment. It is important to note that a contradiction can be used as alpha, like `--alpha '(A0&&!A0)'` to force the algorithm to search for minimal inconsistent subsets of the belief base B. By using the contradiction $(A0 \wedge \neg A0)$, the algorithm is compelled to find kernels, which are minimal subsets of B that lead to a contradiction. This process continues until a minimal inconsistent subset is identified. The remaining optional parameters will be described in the subsequent sections.

The optional parameter `--pruner` activates the computation of the optimal solution using the B&B strategy with optional pruning of suboptimal subproblems according to the following logic:

- **NONE**: The whole search space will be explored and the optimal solution retrieved.
- **UPPER**: Searching for the optimal solution with the maximum value ν, while pruning all subproblems not exceeding a lower bound, as outlined in Section 4.6.1.
- **LOWER**: Searching for the optimal solution with the minimum value ν, while pruning all subproblems exceeding an upper bound, as outlined in Section 4.6.2.

- **BEST**: Searching for the optimal solution with the maximum value ν and the minimum cardinality $|HS|$, while pruning all subproblems exceeding a lower bound, once the maximum value ν is reached, as outlined in Section 4.6.3.

Following the explanations above, an example application call could be:

```
python main.py {file} --vp 3 --alpha '(A0&&!A0)'
--ss PBS --pruner BEST --shrink-sw-size 10 -path-db
--expand-div-conq -no-log
```

With this call, the application will generate a hitting set tree based on kernels that do not entail $\{A0 \wedge \neg A0\}$, also known as α-kernels (`--alpha '(A0&&!A0)'`). The hitting set tree will be expanded using the B&B strategy with the assigned inconsistency weights (`--vp 3`). The algorithm will use divide-and-conquer technique (`--expand-div-conq`) for the expand phase and a window size of 10 (`--shrink-sw-size 10`) for the shrink phase. Furthermore, the hitting set tree will be constructed using the Priority-Based Search (`--ss PBS`). The application will retrieve the dataset from the database (`-path-db`) and not create log files (`-no-log`), although it will save critical logging and verification information.

The applications architecture is depicted in Figure 7. This figure illustrates the complete workflow from loading the datasets to the final output generation. The application contains two separate models, a computation model that computes an optimal solution using the B&B algorithm and the advanced kernel finding strategies, and a verification model that computes all hitting sets by expanding the whole search space of a tree, to obtain the optimal solution that helps verify the result of the computation model.

In the following paragraphs, we provide a detailed description of the computation of the weights and the loading of the datasets into the MySQL database, followed by the three main steps: Input validation, Kernel processing, and Generation of the Hitting Set Tree.

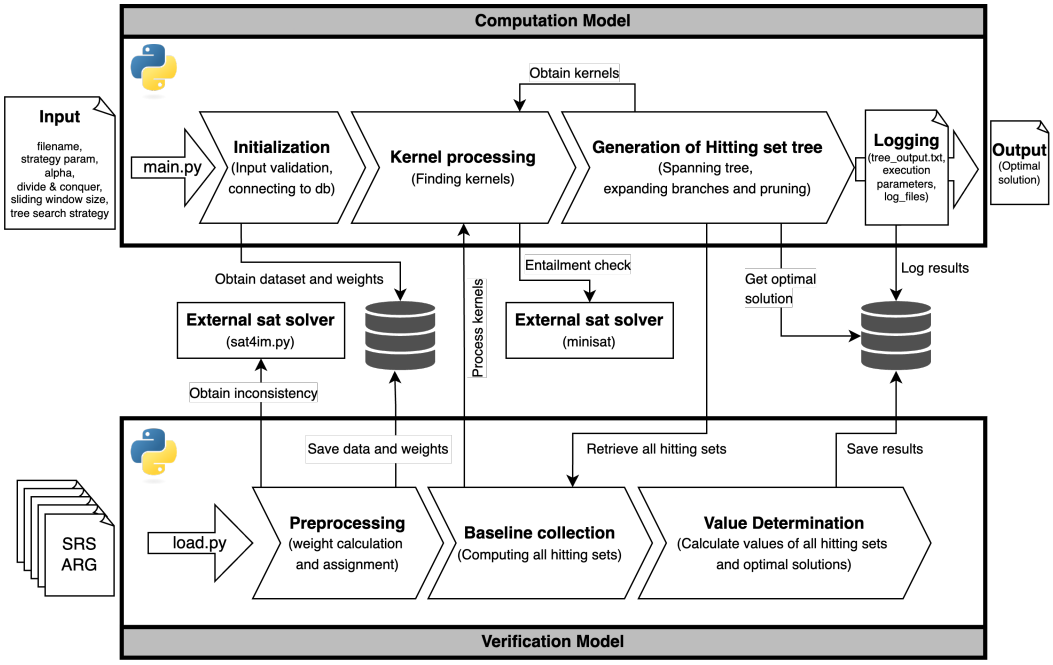


Figure 7: Application architecture

The setup of the application was carried out in two distinct phases. Phase 1 involved loading the SRS and ARG datasets into a database and calculating and assigning weights to the elements of these datasets. Once the datasets were loaded into the database, the verification model was able to calculate all hitting sets for verification. Phase 2 encompassed the actual computation of the hitting set tree, utilizing the datasets and their assigned weights retrieved from the database. The application outputs an optimal solution that was verified by comparing its values to the values generated by the verification model.

Weights for the dataset elements are computed based on either a random measure (see Section 4.3.2) or their inconsistency measures (see Section **??**). For the cardinality approach (see Section 4.3.1), no weight assignment was performed, as the cardinality is determined by simply retrieving the length of the hitting sets.

The datasets, along with their associated weights, are first processed and stored in a MySQL database using the script `load.py`. This script calculates the weights for each element and stores both the data and the calculated weights in the database. This structured storage facilitates efficient retrieval and manipulation of the data during each run of the application. It also allows for an easy verification of the computed solution as the computation model can easily obtain the optimal values and verify its output. Moreover, the MySQL database allows for tracking the results of the application's execution by storing computation-based measures for each dataset and parameter combination. The computation-based measures are further described in Section 6.

In the next sections, we will describe the Verification Model and the Computation Model as depicted in Figure 7.

5.2. Verification Model

To ensure that a computed possible solution $\hat{HS}$ is the optimal solution HS^*, we use the Verification Model, which sets a framework for verifying the computed solutions. The Verification Model comprises three phases: preprocessing, baseline collection, and value determination. Below, we briefly describe the content of these three phases.

5.2.1. Preprocessing

The preprocessing step sets up the application structure, enabling the execution of the application and the reproduction of results explained in Section 6.

As illustrated in Figure 7, preprocessing involves setting up the application and processing the belief bases B stored in the **SRS** and **ARG** datasets. This process includes calculating and assigning weights w_x to each element of B and storing these elements along with their assigned weights. Before storing the weights and belief bases, the system establishes a connection to the database. As explained in Section 4.1, the value calculation may involve calling an external SAT solver. The preprocessing can be initiated by running the Python program `load.py`, which iterates through every dataset and element, stores them in the database, and calculates the weights for the random weight assignment (Section 4.3.2) and the inconsistency weight assignment (Section 4.3.3). For a detailed description of this program, refer to the annex.

The preprocessing is performed only once to set up the application. After successfully executing `load.py`, the Verification Model proceeds to the baseline collection phase.

5.2.2. Baseline Collection

In the second phase of the Verification Model, the application calls the kernel processing and the generation of the hitting set tree, which will be described later. The verification includes computing all possible hitting sets, expanding the entire search space of a hitting set tree HST. This is achieved by storing all found leaf nodes v_{leaf} with their path values $\nu(P(v))$ and other measures, such as the value of the remaining dataset $B(v)$ or the value of the remaining dataset $\nu(B(v))$ for further calculations, as described in Section 4.6.2. The baseline collection stores the leaf nodes with their weights as tuples to keep track of the path values $P(v)$.

Once the baseline collection receives all hitting sets from the generated hitting set tree HST, the Verification Model continues with the value determination phase.

5.2.3. Value Determination

Lastly, the Verification Model determines the values (ν) of the hitting sets for each value assignment and pruning approach to identify the optimal solution (HS^*). Given that three different pruning approaches define the optimal solution differently, the value determination phase calculates all values (ν) and stores them in the database to ensure that a computed optimal solution can be verified afterwards.

After establishing the application's foundation by executing the Verification Model, the computation of optimal solutions with the Computation Model can begin. The Computation Model also consists of three phases, which will be covered in the following sections.

5.3. Computation Model

The application can be run by executing the Python program `main.py`. In the following sections, we will briefly describe the three phases of the Computation Model: Initialization, Kernel Processing, and Generation of the Hitting Set Tree, providing a comprehensive overview of the entire workflow.

5.3.1. Initialization

The initialization phase initializes the program and includes checking the input for valid entries. This involves verifying the correct usage of the input parameters shown in Table 2. Next, the program sets up a connection to the database. In this thesis, the MySQL database was run on a server, so the connection was established via an SSH tunnel. After a successful database connection, the initialization phase loads the belief base from the database and stores it in a data structure (`DataSet()`). The exact data structure is detailed in the UML diagrams in the annex.

After completing the initialization, including input validation and database connection, the application proceeds with the computation of kernels using the selected techniques, depending on the input parameters. This occurs in the next phase, kernel processing, which is described in the following section.

5.3.2. Kernel Processing

As previously mentioned, the application developed in this thesis computes α-kernels of a given belief base (B). The α-kernels are calculated using a straightforward approach as described by Ribeiro [Rib13]. It is important to note that α-kernels can also be computed by first determining a remainder (R), a maximal subset of B that does not entail α ($\alpha \notin Cn(R \subseteq B)$). The remainder can then be subtracted from B ($B \setminus R$) to obtain a minimal subset of B that does entail α ($\alpha \in Cn(B \setminus R)$), considered a hitting set HS. The α-kernels can then be computed using the obtained hitting set by exploring a subset of B that includes all elements from B lying between two elements being subtracted from B in the shrink phase. This method is

not covered in this thesis and left for future work.

A detailed description of the algorithms for computing kernels is provided in Section 5.4. The computed α-kernel will be used to span a hitting set tree, where the first kernel serves as the root node. The generation of the hitting set tree is described in the following section.

5.3.3. Generation of Hitting Set Tree

As described in Section 3.4, each node has as many child nodes as its kernel contains elements. This means that a kernel with three elements contains three child nodes, where each element of the kernel is assigned to one of the three branches. The exploration of the branches depends on the search strategy, explained in Section 4.5. If the application is called with `-ss PBS` and `-vp 3`, the hitting set tree explores its branches based on the priority of the child nodes, where the child nodes are prioritized according to the path value obtained by backtracking and adding the inconsistency weights (w_I) of the edge elements (see Section 18). It is important to note that we assign the weight (w_x) of the edges as the node value because, at the time of branching, the nodes are not yet explored, meaning that the kernels are not yet computed. This is because the computation of the kernel of the respective node is done without the element assigned to the edge of the respective node.

Once the priority-based search algorithm determines the next node to be explored, it backtracks all edges from the node to the root and calls the kernel computation algorithm with the belief base excluding the backtracked edges ($B \setminus \{\beta_1, \beta_2, ..., \beta_i\}$). A detailed algorithm of the hitting set tree generation will be described in accordance with the search strategy algorithms in Section 5.6.

After the hitting set tree has been computed, the application outputs the result and logs it to the database, if the result logging flag `-res-db` was set.

5.3.4. Logging

The logging system in the application is designed to capture and record critical results and metrics during the execution of the application. The key metrics and events logged during the execution include:

- **Execution Time**: The total time taken to execute the program.
- **Strategy Parameters**: The strategy parameters used, such as `sp`, `ss`, sliding window size, and the divide and conquer flag.
- **Kernel Information**: The number of kernels computed, the branches explored, the pruned branches and the depth of the hitting set tree.
- **Alpha**: The specific alpha value used in the computation.
- **Optimal Hitting Set**: The optimal hitting set computed for the solution with its value.

- **Repair and proof**: Saving the repaired dataset in CNF format for validation.

In addition to the result logging, that will be stored in the database, if the result logging flag `-res-db` is set, the application logs information for debugging purposes to log file, wherein each folder generates its own log-file enabling a separation of execution information. This is done unless logging is not disabled via `-no-log`. This comprehensive logging ensures that all critical events and metrics are recorded for analysis and future reference.

In addition to the aforementioned logging, the application saves some of its results in a file `./log/Results.out`. These results represent the validation of the result by validating the computed hitting set with the values generated by the validation model. Furthermore, the application stores all explored hitting sets along with their associated values to ensure that the best solution was computed. For further validation the application generates a repaired dataset file `Results/repair.cnf` that include a repaired dataset, which is obtained by subtracting the optimal solution from B. The repaired dataset is provided in CNF format using the DIMACS standard for validation purposes.

5.4. Advanced Expand Shrink Algorithm

This section covers the implementation of the kernel finding strategy developed to compute kernels from a belief base B. The implementation involves a strategy interface that serves as an abstract base class for all kernel finding strategies. It defines the methods that any concrete strategy class must implement. This design allows for the easy extension of new strategies without modifying the existing codebase. The strategy pattern is utilized to encapsulate the different algorithms for finding kernels within a belief base B. The primary classes involved in this design are `KernelStrategy`, `ShrinkExpand`, and `ExpandShrink`.

The diagram in Figure 28 illustrates the relationships between the `KernelStrategy` interface and its concrete implementations `ShrinkExpand` and `ExpandShrink`. In the diagram, `KernelStrategy` is the abstract base class with two concrete subclasses: `ExpandShrink` and `ShrinkExpand`. The `find_kernel()` method is abstract, which means that any subclass of `KernelStrategy` must provide an implementation for this method. This method is responsible for finding the kernel of a given dataset with respect to a specified element α.

Each subclass implements the `find_kernel()` method, and `ShrinkExpand` provides additional methods such as `find_remainder()`, `shrink()`, `expand()`, and `divide_and_conquer()` to support its strategy. This design allows for flexibility and extensibility in the kernel finding process, enabling the easy addition of new strategies as needed.

The `ExpandShrink` class initially expands the dataset by adding elements until the target element α is entailed, followed by shrinking the dataset to remove redundant elements. The implemented algorithms will be described in Section 5.4.1.

The `ShrinkExpand` class, in contrast, follows a strategy where the dataset is first shrunk by removing elements until the target element α is no longer entailed. The dataset is then expanded by reintroducing elements to find a maximal subset that does not entail α. The implemented algorithms will be described in Section 5.5.

The following section explains the detailed implementation of the kernel finding algorithm proposed by Ribeiro [Rib13] with the implemented improvements using a sliding window and a divide-and-conquer technique.

5.4.1. Advanced Finding Kernel Algorithm

In this section, we present an advanced version of Algorithm 1, originally proposed by Ribeiro [Rib13]. We begin by introducing Algorithm 2, a global and simplified version of the Find Kernel Algorithm. The subsequent section will elaborate on the advancements, specifically the expand phase using either the sliding window or divide-and-conquer methods, and the shrink phase employing similar techniques.

Algorithm 2: Find Kernel Algorithm

Input: B, α
Output: Found α-kernel

```
Function find_kernel(B, α):
    if α ∈ Cn(B) then
        return expand(B, α)
    else
        return None
Procedure expand(B, α):
    B' ← ∅
    for β ∈ B do
        B' ← B' ∪ {β}
        if α ∈ Cn(B') then
            return shrink(B', α)
    return B'
Procedure shrink(B, α):
    for β ∈ B do
        if α ∈ Cn(B \ {β}) then
            B ← B \ {β}
        return B
```

In this section, we present an advanced version of Algorithm 1, originally proposed by Ribeiro [Rib13]. This refined approach includes Algorithm 2, which is a global and simplified version of the Find Kernel Algorithm. The advance-

ments, specifically the expand phase using either the sliding window or divide-and-conquer methods, and the shrink phase employing similar techniques, will be detailed in the subsequent section.

In line 2, Algorithm 1 first checks if α is an entailment of the input belief base B. If it is, the algorithm continues with the expand phase by passing B to `expand()`. Otherwise, it returns `None`, which will be handled as an empty kernel.

In the expand phase (lines 6 to 11), the procedure initializes an empty set B' and adds elements from B to B'. In this simplified version, elements are added one by one. However, in the remainder of this thesis, we will demonstrate the use of the sliding window technique during the expand phase. In line 10, the algorithm checks if α is an entailment of the expanded belief base B'. If so, the algorithm has found a set of elements that entail α, and it proceeds to the shrink phase (lines 12 to 19) by calling the `shrink()` procedure. To check if $\alpha \in Cn(B')$, we are calling the MiniSat SAT solver. For an explanation of the logic behind entailment and satisfiability, refer to Section 3.3.

In the shrink phase (lines 12 to 19), the procedure iterates through each element β in B. For each element, it checks if α is still an entailment of B without β. If it is, β is removed from B. The procedure continues this process until no more elements can be removed without losing the entailment of α. Finally, the reduced set B is returned as the α-kernel.

Algorithm 2, as the global version of the implementation in this thesis, is very similar to Algorithm 1, although Algorithm 2 includes an entailment check of α before the expand phase, ensuring that empty kernels are identified immediately. This step is crucial because, when spanning a hitting set tree, Algorithm 2 will be called multiple times without the elements appearing on the edge of the branches of the tree, resulting in a decreased dataset B.

5.4.2. Expand phase with sliding window

In this section, we will discuss an advanced version of the expand phase using a sliding window technique. According to Table 1, this technique was used by Cobe and Wassermann [CW15] in the shrink phase of an algorithm for computing remainders. We will describe the sliding window technique that differs from the implementation of Cobe and Wassermann [CW15], as presented in Algorithm 3.

In line 2, the procedure initializes an empty set B'. In line 3, the elements of the input belief base B are assigned to the variable `elements`. The procedure then enters a loop (line 4) that iterates from the start index to the length of `elements`, incrementing by the size of the sliding window specified by `window_size`. Within this loop, the procedure calculates the end of the current window (line 5) to ensure it does not exceed the bounds of `elements`. It then extracts the elements within the current window (line 6) and adds them to B' (lines 7 to 9). After expanding B' with the current window of elements, the procedure checks if α is an entailment of B' (line 10). If α is entailed by B', the procedure proceeds to the shrink phase

Algorithm 3: Expand with sliding window

Input: $B, \alpha,$ `window_size`
Output: Expanded dataset B'

```
1  Procedure expand(B, α, window_size):
2      B' ← ∅
3      elements ← B
4      counter ← 0
5      for start ← 0 to |elements| by window_size do
6          window_end ← min(start + window_size, |elements|)
7          window_elements ← elements[start:window_end]
8          for β ∈ window_elements do
9              B' ← B' ∪ {β}
10         if α ∈ Cn(B') then
11             return shrink(B', α)
12     return B'
```

by calling the `shrink()` function with B' and α as arguments (line 11). If α is not entailed, the loop continues with the next window of elements until all elements are processed or α is found to be entailed.

Example 13. Let the belief base B comprise the following elements:

$$B = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \neg A5 \wedge A2, A2, \neg A4 \wedge \neg A2, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$$

With `window_size` set to 3, Algorithm 3 would iterate over B as follows:

$$B = \{\boxed{A0, \neg A0 \vee A1, \neg A1 \vee A2}, \neg A5 \wedge A2, A2, \neg A4 \wedge \neg A2, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$$
$$B = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \boxed{\neg A5 \wedge A2, A2, \neg A4 \wedge \neg A2}, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$$
$$B = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \neg A5 \wedge A2, A2, \neg A4 \wedge \neg A2, \boxed{\neg A1, \neg A0 \vee \neg A1, \neg A3}\}$$

As the elements of the window are added to B', the window size is used as the step size. In contrast, Cobe and Wassermann [CW15] use a step size of 1 for the sliding window technique in the shrink phase. The sliding window technique can be applied in various ways for the expand phase. Besides the implementation used in this thesis, an alternative approach could involve sliding the window over the elements of B and checking if α is entailed by the elements within the window, rather than adding the elements of the window and checking the set of added elements. This alternative approach is a potential subject for future work. Additionally, the sliding window technique could be applied to both the expand and shrink phases. However, in this thesis, we use the divide-and-conquer approach for the shrink phase, which will be described in the next section.

5.4.3. Expand phase with divide-and-conquer

In this section, we will discuss an advanced version of the expand phase using a divide and conquer technique. According to Table 1, this technique was used by Cobe and Wassermann [CW15]. The advanced expand phase is illustrated in Algorithm 4.

Algorithm 4: Expand with divide-and-conquer

Input: B, α
Output: Expanded dataset B

```
1  Function expand_divide_and_conquer(B, α):
2      if |B| ≤ 1 then
3          return expand(B, α)
4      B1, B2 ← B.split()
5      cn_B1 ← Cn(B1, α)
6      cn_B2 ← Cn(B2, α)
7      if ¬cn_B1 and ¬cn_B2 then
8          return expand_divide_and_conquer(B, α)
9      else
10         if cn_B2 then
11             return expand_divide_and_conquer(B2, α)
12         else
13             B ← B1 ∪ B2
14             return expand(B, α)
```

Algorithm 4 describes a divide-and-conquer method for expanding a dataset B using a given parameter α. The algorithm works as follows:

The main function `expand_divide_and_conquer()` is defined with inputs B and α. In line 2, the function checks if the size of B is less than or equal to 1. If this condition is true, it directly calls the `expand()` function (line 3) and returns its result.

In line 4, the function splits B into two subsets, $B1$ and $B2$. It then checks whether α is entailed by $B1$ and $B2$ using the `Cn()` function (lines 5-6).

If α is not entailed by both $B1$ and $B2$ (`cn_B1` and `cn_B2` are both false), the function recursively calls `expand_divide_and_conquer()` on the original set B (line 7).

Otherwise, if α is entailed by $B2$ (`cn_B2` is true), the function recursively calls `expand_divide_and_conquer()` on $B2$ (line 9).

If α is not entailed by $B2$ (`cn_B2` is false), the function combines $B1$ and $B2$ into B and calls the `expand()` function with B and α, then returns its result (lines 11-12).

This method ensures that the dataset B is expanded using the divide-and-conquer

strategy with the given parameter α.

For a detailed analysis of Algorithm 4 we refer to [CW15]. In the next section, we will briefly describe the implementation of an algorithm to compute a hitting set from a remainder using Shrink-Expand.

5.4.4. Shrink phase with sliding window

In this section, we will describe the advanced version of the shrink phase using a sliding window and the advanced expand phase using a divide and conquer technique as proposed by Cobe and Wassermann in [CW15].

Algorithm 5: Shrink with sliding window

Input: B, α
Output: Shrunk dataset B and removed elements

```
1 Function shrink(B, α):
2     removed_elements ← ∅
3     for start ← 0 to |B| do
4         window ← B[start:start + window_size]
5         if α ∈ Cn(B \ window) then
6             B \ window
7             removed_elements ∪ window
8     return B, removed_elements
```

Algorithm 5 shrinks a given dataset B using a sliding window approach while ensuring that a specified element α remains a consequence of the dataset. The algorithm operates as follows:

In line 1, the main function `shrink()` is defined with inputs B and α. The function initializes an empty set `removed_elements` to keep track of the elements that are removed from B (line 2).

The algorithm then iterates over the dataset B in windows of size `window_size` (line 3). For each window, a subset of elements from B is selected (line 4). The algorithm checks if α is still entailed by B after removing the elements in the current window (line 5). If α remains entailed, the elements in the window are removed from B (line 6), and they are added to the `removed_elements` set (line 7).

The iteration continues until all elements in B have been processed. Finally, the function returns the shrunk dataset B and the set of removed elements (line 8). Again, we refer to [CW15]. The next section describes the shrink phase with a divide-and-conquer strategy.

5.4.5. Shrink phase with divide-and-conquer

In this section, we will discuss an advanced version of the shrink phase using a divide and conquer technique. According to Table 1, this technique was used by Cobe and Wassermann [CW15] in the expand phase of an algorithm for computing remainders. We will describe the divide and conquer technique that differs from the implementation of Cobe and Wassermann [CW15], as presented in Algorithm 6.

Algorithm 6: Shrink with divide-and-conquer

Input: B, α
Output: Reduced dataset

```
1  Function divide_and_conquer(B, α):
2      if |B| ≤ 1 then
3          return B if Cn(B, α) else {}
4      B1, B2 ← B.split()
5      cn_B1 ← Cn(B1, α)
6      cn_B2 ← Cn(B2, α)
7      if ¬cn_B1 and ¬cn_B2 then
8          return shrink(B, α)
9      if cn_B1 then
10         return divide_and_conquer(B1, α)
11     if cn_B2 then
12         return divide_and_conquer(B2, α)
```

In line 1, the function `divide_and_conquer` is defined with inputs `B` and α. In line 2, the function checks if the size of B is less than or equal to 1. If true, it returns B if `Cn(`B`, `α`)` is satisfied; otherwise, it returns an empty dataset (line 3). In line 4, the dataset B is split into two subsets, $B1$ and $B2$. The condition α is checked on both subsets (lines 5 and 6), with the results stored in `cn_B1` and `cn_B2`. In line 7, if neither subset satisfies the condition, the function calls `shrink` on the original dataset B and returns the result. If $B1$ satisfies the condition (line 8), the function recursively calls itself on $B1$. Similarly, if $B2$ satisfies the condition (line 9), the function recursively calls itself on $B2$.

It is important to note that Algorithm 6 employs a straightforward approach where it passes both parts $B1$ and $B2$ of B to the `shrink()` function from Algorithm 1 if neither half entails α. An alternative approach could involve splitting B into more than two parts and recombining them in various ways to see if any new combination entails α. While this method can be more computationally expensive, it may be more effective in identifying a valid subset. Another alternative is to combine parts of the dataset randomly or based on certain heuristics to explore different combinations of elements that might entail α. These alternative strategies are potential subjects for future research.

Example 14. In this example, we are using the belief base B from Example 13:

$$B = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \neg A5 \wedge A2, A2, \neg A4 \wedge \neg A2, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$$

Consider $\alpha = A1$. Using the divide-and-conquer technique described in Algorithm 6, the dataset B is split and processed as follows:

In the first iteration, B is split into two halves $B1_1$ and $B2_1$ (indicating the recursion step by subscripted numbers):

$$B1_1 = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \neg A5 \wedge A2\}$$
$$B2_1 = \{A2, \neg A4 \wedge \neg A2, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$$

In the following step, the algorithm checks the entailment of α for both halves $B1_1$ and $B2_1$ and assigns the result to the variables $\texttt{cn\_B1}_1$ and $\texttt{cn\_B2}_1$:

$$\texttt{cn\_B1}_1 = \text{Cn}(B1_1, \alpha) = \textit{true}$$
$$\texttt{cn\_B2}_1 = \text{Cn}(B2_1, \alpha) = \textit{false}$$

Since $B1_1$ entails α, the algorithm recursively calls `divide_and_conquer()` with $B1_1$ and α as arguments. In the second iteration, $B1_1$ is split into two halves $B1_2$ and $B2_2$:

$$B1_2 = \{A0, \neg A0 \vee A1\}$$
$$B2_2 = \{\neg A1 \vee A2, \neg A5 \wedge A2\}$$

Now both halves $B1_2$ and $B2_2$ are checked for the entailment of α and assigned to the variables $\texttt{cn\_B1}_2$ and $\texttt{cn\_B2}_2$:

$$\texttt{cn\_B1}_2 = \text{Cn}(B1_2, \alpha) = \textit{true}$$
$$\texttt{cn\_B2}_2 = \text{Cn}(B2_2, \alpha) = \textit{false}$$

Since $B1_2$ entails α, the algorithm again recursively calls `divide_and_conquer()` with $B1_2$ and α as arguments. In the third iteration, $B1_2$ is split into two halves $B1_3$ and $B2_3$:

$$B1_3 = \{A0\}$$
$$B2_3 = \{\neg A0 \vee A1\}$$

Now both halves $B1_3$ and $B2_3$ are checked for the entailment of α and assigned to the variables $\texttt{cn\_B1}_3$ and $\texttt{cn\_B2}_3$:

$$\texttt{cn\_B1}_3 = \text{Cn}(B1_3, \alpha) = \textit{false}$$
$$\texttt{cn\_B2}_3 = \text{Cn}(B2_3, \alpha) = \textit{false}$$

Since neither half $B1_3$ nor $B2_3$ entails α, the algorithm combines both halves and calls the `shrink()` function with the combined halves $B1_3$ and $B2_3$.

Step	Splits $B1_x$	Entailment Check
1	$B1_1 = \{A0, \neg A0 \vee A1, \neg A1 \vee A2, \neg A5 \wedge A2\}$	$\texttt{cn\_B1}_1 =$ true
	$B2_1 = \{A2, \neg A4 \wedge \neg A2, \neg A1, \neg A0 \vee \neg A1, \neg A3\}$	$\texttt{cn\_B1}_1 =$ false
2	$B1_2 = \{A0, \neg A0 \vee A1\}$	$\texttt{cn\_B1}_2 =$ true
	$B2_2 = \{\neg A1 \vee A2, \neg A5 \wedge A2\}$	$\texttt{cn\_B1}_2 =$ false
3	$B1_3 = \{A0\}$	$\texttt{cn\_B1}_3 =$ false
	$B2_3 = \{\neg A0 \vee A1\}$	$\texttt{cn\_B2}_3 =$ false

Table 3: Recursive Steps of the divide-and-conquer Algorithm

Thus, the divide-and-conquer technique iteratively splits, processes, and recombines the dataset B to reduce it while checking the entailment of α.

The recursion steps with the respective variables and its assignments are displayed in Table 3

In the next section we will breifly describe the implementation of an algorithm to compute a hitting set value from a remainder using Shrink-Expand as proposed by Ribeiro [Rib13].

5.5. Finding Remainder Algorithm

In this section, we describe the implementation of an algorithm to compute a remainder that will be used to obtain a value $\nu(P(v))$ using the objective function describesd in Section 4.6.2.

Algorihtm 7 aims to determine a remainder of the dataset B with respect to α, meaning to find a subset B', such that $\alpha \notin B' \subseteq B$. The primary function, `find_remainder`, first shrinks the dataset by removing elements that do not affect the entailment of α through the `shrink` function (line 2). The `shrink` function iterates over each element in B, removes it temporarily, and checks if α is still entailed; if it is, the element β is added to a set of removed elements (lines 7 to 10). The `find_remainder` procedure then calls the `expand` function in line 3, which tries to add back the removed elements one by one, ensuring that α is not entailed in the process.

For this algorithm we also implemented the sliding window and divide-and-conquer methods described previously. For a detailed analysis of Algorithm 4 we refer to [CW15]. The following section details the advanced hitting set algorithm including the spanning of the tree and the different pruning strategies.

5.6. Hitting Set Tree Algorithm

In this section, we explain the implementation of the hitting set tree algorithm employing search strategies and pruners to find new kernels and span the hitting set tree. The search implementation is illustrated in Figure 30.

Algorithm 7: Find Remainder Algorithm

Input: B, α
Output: Kernel based on found remainder

```
1  Function find_remainder(B, α):
2      remainder_dataset, removed_elements ← shrink(B, α)
3      return expand(remainder_dataset, removed_elements, α)
4  Procedure shrink(B, α):
5      B' ← B
6      removed_elements ← ∅
7      for β ∈ B' do
8          if α ∈ Cn(B' \ {β}) then
9              B' ← B' ∪ {β}
10             removed_elements ← {β}
11     return B', removed_elements
12 Procedure expand(B, removed_elements, α):
13     for β ∈ removed_elements do
14         if α ∉ Cn(B ∪ {β} then
15             B ← B ∪ {β}
16     return B
```

The hitting set tree algorithm utilizes various search strategies to efficiently explore the solution space. Each search strategy, including Breadth-First Search (BFS), Depth-First Search (DFS), Hybrid Search (HYS), and Priority-Based Search (PBS), offers unique advantages and challenges. The following sections provide a detailed description of each strategy and its implementation. In the following, we will describe the Priority-Based Search (PBS) in more detail. For the remaining search strategies (BFS, DFS, HYB) we will focus on the differences in terms of the utilized data structures.

5.6.1. Priority-Based Search Algorithm

The Priority Search (PBS) algorithm priorities nodes based on a specific criterion, often related to the kernel's properties or search depth. The algorithm creates the initial node, adds it to the priority queue, and processes nodes based on their priority. The implementation of the PBS is explained in Algorithm 8.

The Priority Search (PBS) algorithm is designed to find the kernel of a given dataset. The process begins in the `find_kernels` function (lines 1 to 6), which initializes a sorted priority queue and creates the root node using the `create_initial_node` function (lines 7 to 12). If the root node is not `None`, the algorithm proceeds to the `priority_search` function (line 5).

Algorithm 8: Priority Search (PBS)

Input: Dataset `B`, α, ExpShrink, Brancher, Pruner

```
Function find_kernels(B, α):
    queue ← SortedList()
    v_root ← create_initial_node(B, α)
    if v_root ≠ None then
        return priority_search(v_root)
    return None
Function create_initial_node(B, α):
    result ← ExpShrink.find_kernel(B, α)
    if result = None then
        return None
    v_root ← new(v)
    return v_root
Function priority_search(v_root):
    queue.add(v_root)
    while queue ≠ ∅ do
        element ← queue.pop()
        v ← element
        if Pruner.should_prune(v) then
            prune(v)
            continue
        if v_kernel = ∅ then
            result ← ExpShrink.find_kernel(v.dataset, α)
            if result ≠ None then
                v.kernel ← result
                v_children ← Brancher.expand_children(v)
                foreach v_child ∈ v_children do
                    queue.add(v_child)
            else
                v_leaf ← v
                Pruner.update_boundary(v_leaf)
        else
            Brancher.expand_children(v)
            foreach v_child ∈ v_children do
                queue.add(v_child)
```

The `create_initial_node` function (lines 7 to 12) calls the `ExpShrink.find_kernel` method to identify the initial kernel for the dataset. If no kernel is found, it returns `None`; otherwise, it creates and returns a new node with the found kernel.

The core of the algorithm is the `priority_search` function (lines 13 to 34). This function manages the priority queue, which starts with the root node (line 14). It processes each node by popping the highest priority element from the queue (line 16) and checking if it should be pruned using the `Pruner.should_prune` method (lines 18 to 20) of the pruner that will be descibed in the following sections.

If the node does not have a kernel (line 21), the algorithm attempts to find one using `ExpShrink.find_kernel` (line 22) using the assigned dataset of the node v. In general we are assingning each node v a dataset that contains all elements B except the elements of its path $P(v)$. If a kernel is found, it sets the kernel and expands the node's children (lines 23 to 27), pushing each child onto the queue (lines 26 to 27). If no kernel is found, it updates the boundary with the leaf node (lines 29-30). If the node already has a kernel, it simply expands its children and adds them to the queue (lines 32 to 34).

In summary, the PBS algorithm systematically searches through the dataset, using a sorted priority queue that automatically sorts the nodes v to manage a priorization of the nodes v, and employs strategies to find and expand kernels while pruning unnecessary nodes.

The process of Priority-Based Search can be summarized as follows:

1. **Initialization:** Start with the initial node representing the entire problem. Calculate path values of the branches and add the nodes to the priority queue.

2. **Exploration:** Continuously extract the subproblem with the highest priority from the sorted queue. If the current subproblem can be expanded, generate its children, calculate their path value, and add them to the queue.

3. **Boundary Update:** If a leaf node (complete solution) is found, update the boundary if this solution is better than the current best. This helps in pruning other subproblems more effectively.

4. **Pruning:** Evaluate whether to prune a subproblem based on the current boundary. If the subproblem cannot yield a better solution than the current best, it is pruned and not further explored.

5. **Termination:** The process continues until there are no more subproblems to explore in the queue. The best solution found during the search is returned as the optimal solution.

By integrating the strengths of the Priority-Based Search and dynamic prioritization, the Priority-Based Search strategy developed in this thesis provides an effective means of navigating large and complex search spaces, making it well-suited for solving optimization problems within the B&B framework. Nonetheless, we have

implemented other search strategies for evaluation. It is especially how the following search strategies perform when using an aggressive pruning approach, which will be described following the search strategy implementation.

5.6.2. BFS, DFS, and HYS

Here we describe the main differences in queuing and node handling for BFS, DFS, and HYS algorithms compared to PBS.

BFS In BFS, nodes are processed level by level, meaning all nodes at the current depth are processed before moving to the next depth level. Algorithm 9 shows a brief implementation of this search strategy.

Algorithm 9: Breadth-First Search (BFS)

```
   Input: Dataset B, α, ExpShrink, Brancher, Pruner
   Output: Kernel of the dataset
1  Function bfs(B, α):
2      v_root ← new(v)
3      queue ← {v_root}
4      while queue ≠ empty do
5          v ← queue.pop()
6          result ← ExpShrink.find_kernel(v.dataset, α)
7          if result ≠ None then
8              v.set_kernel(result)
9              v_children ← Brancher.expand_children(v)
10             foreach child ∈ v_children do
11                 queue.append(child)
```

The Breadth-First Search (BFS) algorithm explores nodes level by level. Its `find_kernels` function initiates BFS on dataset `B` with α. The `bfs` function initializes an empty queue and finds the initial kernel. If a kernel is found, it creates a root node v_{root}, sets it as the tree root, and adds it to the queue. The algorithm processes nodes level by level, pruning nodes as needed. For each node, it creates a reduced dataset and a child node for each element in the kernel. If a kernel is found in the reduced dataset, the child node is added to the queue; otherwise, it is marked as a leaf. The tree structure is logged at the end. For a detailed implementation the reader is referred to the code available on GitHub [3].

DFS In DFS, nodes are processed by exploring as far as possible along each branch before backtracking. A simplified implementation of the DFS is explained in Algo-

[3] https://github.com/SebastianMueller41/Masterthesis

rithm 10.

Algorithm 10: Depth-First Search (DFS)

Input: Dataset `B`, α, ExpShrink, Brancher, Pruner
Output: Kernel of the dataset

```
1 Function dfs(B, α, v):
2     for element ∈ v.kernel do
3         v_child ← new(v)
4         if ExpShrink.find_kernel(v.dataset, α) ≠ None then
5             queue.push(v_child)
6     while queue ≠ empty do
7         v_next ← queue.popBack()
8         dfs(v_next.dataset, α, v_next)
```

The Depth-First Search (DFS) algorithm explores as far as possible along each branch before backtracking. The `find_kernels` function initiates the DFS process on the dataset `B` with respect to α and then prints and logs the tree structure. The `dfs` function begins by finding the initial kernel if the `parent` node is `None`. If a kernel is found, a root node v_{root} is created and set as the tree root, and the DFS function is called recursively with the root's dataset and the root node. If a `parent` node is provided, the algorithm checks if the node should be pruned. If not, it iterates through each element in the parent's kernel, creates a reduced dataset, and calculates the bounding box value. A child node is created for each reduced dataset, and if a kernel is found, the child node's kernel is set, and the DFS function is called recursively on the child node. If no kernel is found, the child node is marked as a leaf. The tree structure is logged at the end.

HYS The Hybrid Search (HYS) algorithm combines DFS and BFS strategies. Initially, it performs DFS until the first leaf is found, then switches to BFS. A simplified implementation of the HYS is explained in Algorithm 11, which basically uses both implementations from BFS and DFS.

The Hybrid Search (HYS) algorithm combines both depth-first and breadth-first search strategies. The `find_kernels` function initiates the process based on whether the first leaf has been found. If `first_leaf_found` is `True`, it logs a message indicating the switch to BFS, sets the tree of the BFS search to the tree of the DFS search, and then runs the BFS search to find kernels. Finally, it sets the tree of the DFS search to the tree of the BFS search to synchronize the tree structures. If `first_leaf_found` is `False`, it runs the DFS search to find kernels and sets the tree of the DFS search to the current tree.

The following section covers the branching strategy that is responsible for expanding the children of a node.

Algorithm 11: Hybrid Search (HYS)

Input: Dataset `B`, α, Kernel Strategy, Brancher, Pruner
Output: Kernel of the dataset

```
Function find_kernels(B, α):
    if first_leaf_found then
        bfs.tree ← dfs.tree
        bfs.find_kernels()
        dfs.tree ← bfs.tree
    else
        dfs.find_kernels()
        tree ← dfs.tree
```

5.7. Branching Implementation

The branching and pruning serve as two phases of the B&B (see 4.4) that together with the search strategy complete the B&B framework. The following sections cover the implementation of the brancher and the developed pruners.

The implemented `Brancher` class is a crucial component of the B&B framework, responsible for expanding the search tree by generating child nodes from the current node. The branching strategy highly depends on the computation of kernels. For each element of a kernel of a node, one branch will be generated, effectively spanning hitting set trees.

Algorithm 12: Brancher Pseudoalgorithm

Data: ExpShrink, Dataset
Result: List of child nodes

```
Function expand_children(v):
    if v.kernel = ∅ or v = v_leaf then
        return []
    children ← []
    for β in v.kernel do
        reduced_dataset ← v.dataset
        reduced_dataset.remove(β)
        v_child ← new(v)
        ν(P(v_child)) ← ν(P(v)) + w(β)
        v_children.append(v_child)
        children.append(v_child)
    return children
```

The `Brancher` uses the specified kernel strategy and the dataset. Its primary

function, `expand_children`, is responsible for generating the child nodes of the current node based on its kernel k. The `expand_children` method is the core function of the `Brancher`. It first checks if the current node's kernel $v.kernel$ is `None` or if the current node v is a leaf v_{leaf} (lines 2 to 3). If either condition is true, it returns an empty list, indicating no further expansion is possible. For each element β in the current node's kernel $v.kernel$ (line 5), the method removes the element β from the current node's dataset to create a reduced dataset (lines 6 to 7). It then calculates the child path value $\nu(P(v_{child}))$ by adding the weight w of the element β to the path value $\nu(P(v))$ of the node v (line 9). A new child node (`new(v)`) is created with the path value, linking it to the current node v as its parent (line 10). The newly created child node v_{child} is then added to the current node's children $v_{children}$ (line 11). Finally, the method returns the list of generated child nodes (line 19).

When spanning hitting set trees, the branching strategy is heavily dependent on the computation of kernels k. Each element β in a kernel k of a node v generates a new branch, expanding the search tree. This systematic expansion ensures that all potential solutions are explored efficiently, with the tree structure dynamically adapting based on the kernel computations at each node v.

This branching method, combined with effective kernel computation and pruning strategies, forms a robust framework for efficiently solving the belief base B contraction problem, as further evaluated in subsequent sections. Therefore, we will now describe the implementation of the Pruner.

5.8. Pruner Implementation

In this sections, we present the implementation of three different pruners: the Upper Pruner, Lower Pruner, and Best Pruner as described in Section 4.6. These pruners are designed to optimize the search process by eliminating nodes that cannot lead to a better solution, thereby improving the efficiency of the B&B framework. Figure 30 illustrates the class implementation of the pruners. All of the implemented pruners (UPPER, LOWER and BEST) extend a global pruner called `BasePruner`. The `BasePruner` not only serves to handle shared functions of the pruners, but also the base case where no pruning is selected.

5.8.1. Base Pruner

The Base Pruner provides the foundational structure for all pruners. It includes methods for calculating potential values $\nu_P(P(v))$, finding remainders R in datasets, and computing potential bounds for nodes v. The Base Pruner will be explained in Algorithm 13

In Algorithm 13 the `calculate_subproblem` function (lines 1 to 4) calculates the value $\nu(P(v))$ as the sum of the weights w_i of the elements β_i in the dataset of a node v and assigns $\nu(P(v))$ to the node's sub_value, which it then returns. The `find_remainder_in_dataset` function (line 5) returns the remainder R found in

Algorithm 13: Base Pruner Pseudoalgorithm

Data: Kernel Strategy, Tree
Result: Subproblem value or boolean for pruning decision

```
1  Function calculate_subproblem(v):
2      ν(P(v)) ← sum of elements in v.dataset
3      v.sub_value ← ν(P(v))
4      return ν(P(v))
5  Function find_remainder_in_dataset(dataset):
6      return remainder.find_remainder(dataset)
7  Function calculate_potential_bound(v):
8      if remainder_flag then
9          R ← find_remainder(v.dataset)
10         if R ≠ ∅ then
11             foreach β_i ∈ R do
12                 ν(R) ← ν(R) + w_i(β_i)
13         else
14             ν(R) ← 0
15     ν_P(P(v)) ← ν(P(v)) − ν(R)
16     return ν_P(P(v))
17 Function should_prune(v):
18     return False
```

the given dataset. The `calculate_potential_bound` function (lines 7 to 16) calculates the potential value $\nu_P(P(v))$ of a node v. If the remainder flag is set, it finds the remainder R of the dataset and summarizes the weights w_i of all elements β of the found remainder R. Next, it subtracts the remainder value $\nu(R)$ from the path value $\nu(P(v))$ to get the potential path value $\nu_P(P(v))$ (line 15) which is returned in line 16. If the function does not find a remainder R, the potential path value $\nu_P(P(v))$ is simply the path value $\nu(P(v))$ of node v. Finally, the `should_prune` function (lines 17 to 18) always returns `False`, indicating no pruning decision is made at this stage. This algorithm provides essential functionalities for more specialized pruners that build upon it, like the upper pruner and lower pruner, that will be described in the following sections.

5.8.2. Upper Pruner

The Upper Pruner is responsible for pruning subproblems that do not meet or exceed the lower Bound $\mathcal{B}_L$ as explained in Section 4.6.1. A simplified implementation is illustrated in Algorithm 14. The pruner updates the boundary using leaf nodes

v_{leaf} and determines whether a node v should be pruned based on its path value $\nu(P(v))$. As explained in Section 4.6 we are using the complementary path value $\nu_{\min}(P(v)) = \nu(B) - \nu(P(v))$ to have a consistent implementation for the Upper Pruner and Lower Pruner, which will be explained in the remainder of this section.

Algorithm 14: Upper Pruner Implementation

```
   Data: Hitting Set Tree
   Result: Boolean indicating whether to prune the node
1  Function update_boundary_with_leaf(v_leaf):
2      if ν_min(P(v_leaf)) > tree.lowerBound then
3          tree.lowerBound ← ν_min(P(v_leaf))
4  Function should_prune(v):
5      return ν_min(P(v)) ≤ tree.lowerBound
```

The Algorithm 14 incorporates a `update_boundary_with_leaf` function (lines 1 to 3) that is responsible for updating the tree's lower bound $\mathcal{B}_L$ when a leaf node v_{leaf} is encountered. It compares the complementary path value $\nu_{\min}(P(v_{\text{leaf}}))$ of the leaf node v_{leaf} with the current lower bound $\mathcal{B}_L$ of the tree, and if the complementary path value $\nu_{\min}(P(v_{\text{leaf}}))$ is greater, it updates the lower bound $\mathcal{B}_L$ to this value. The `should_prune` function (lines 4 to 5) determines whether a node v should be pruned by comparing the complementary path value $\nu_{\min}(P(v))$ of the node v with the tree's lower bound $\mathcal{B}_L$. If the complementary path value $\nu_{\min}(P(v))$ is less than or equal to the lower bound $\mathcal{B}_L$, the function returns `True`, indicating that the node v should be pruned. Otherwise, it returns `False`. This approach ensures that the search space is efficiently reduced by eliminating nodes v that cannot lead to an optimal solution, which is deemed to represent the minimal path value $\nu(P(v))$.

5.8.3. Lower Pruner

The Lower Pruner class also extends the `BasePruner` and prunes nodes based on a upper bound criterion. It updates the upper bound $\mathcal{B}_U$ using leaf nodes v_{leaf} and checks if a node v should be pruned by calculating its potential value $\nu_P(P(v))$ as described in accordance with the Base Pruner.

The `Lower Pruner` Algorithm 15 also consists of the two main functions `update_boundary_with_leaf` and `should_prune`. The `update_boundary_with_leaf` function (lines 1 to 3) is responsible for updating the tree's upper bound when a leaf node is encountered. It compares the subproblem value $\nu(P(v_{\text{leaf}}))$ of the leaf node with the current upper bound of the tree, and if the subproblem value is less than the upper bound, it updates the upper bound to this value.

The `should_prune` function (lines 4 to 6) determines whether a node

Algorithm 15: Lower Pruner Implementation

Data: Kernel Strategy, Tree
Result: Boolean indicating whether to prune the node

```
1 Function update_boundary_with_leaf(v_leaf):
2     if ν(P(v_leaf)) < tree.upperBound then
3         tree.upperBound ← ν(P(v_leaf))
4 Function should_prune(v):
5     ν_P P(v)) ← calculate_potential_bound(v)
6     return ν_P(P(v)) ≤ tree.upperBound
```

should be pruned by calculating its potential bound $\nu_P(P(v))$ using the `calculate_potential_bound` method. If the potential bound is less than or equal to the tree's upper bound, the function returns `True`, indicating that the node should be pruned. Otherwise, it returns `False`. This method ensures that nodes which cannot improve the current best solution are effectively pruned from the search space, thus optimizing the search process.

5.8.4. Best Pruner

The Best Pruner class extends the `BasePruner` and incorporates both the upper and lower bound criteria for pruning. It also performs an initial check for potential duplicate sums to determine if the dataset qualifies for a specialized pruning approach.

The `Best Pruner` algorithm is designed to prune nodes v in a search tree based on both upper bound and lower bound criteria, incorporating an initial check for potential duplicate sums to determine if this specialized pruning approach can be applied. Algorithm 16 consists of three main functions: `update_boundary_with_leaf`, `should_prune`, and `has_potential_duplicate_sums`. The `update_boundary_with_leaf` function (lines 1 to 9) updates the tree's lower bound $\mathcal{B}_L$ and upper bound $\mathcal{B}_U$ when a leaf node v_{leaf} is encountered. It first calculates the value $\nu(B)$ (lines 2 to 3). If the path value $\nu(P(v_{\text{leaf}}))$ of the leaf node equals the value $\nu(B)$, it updates the tree's lower bound $\mathcal{B}_L$ to this path value $\nu(P(v))$ and the upper bound $\mathcal{B}_U$ to the cardinality of the path P to the node v ($|P(v)|$) (lines 5 to 6). If the path value $\nu(P(v_{\text{leaf}}))$ is greater than the current lower bound $\mathcal{B}_L$, it updates the lower bound $\mathcal{B}_L$ to the path value $\nu(P(v))$ (line 9). The `should_prune` function (lines 10 to 16) determines whether a node v should be pruned by checking if the tree's lower bound $\mathcal{B}_L$ is zero (line 11) or if the cardinality of the node path $|P(v)|$ is greater than or equal to the upper bound $\mathcal{B}_U$ when the maximum is reached (lines 13 to 15). Returns `True` if the node should be pruned; otherwise, it compares the node's path value to the tree's lower bound (line 16). The `has_potential_duplicate_sums` function (lines 17

Algorithm 16: Best Pruner Implementation

Data: Kernel Strategy, Tree
Result: Boolean indicating whether to prune the node

Function `update_boundary_with_leaf(`v_{leaf}`)`**:**
  foreach $\beta_i \in B$ **do**
    $\nu(B) \leftarrow \nu(B) + w(\beta_i)$
  if $\nu(P(v_{\text{leaf}})) = \nu(B)$ **then**
    tree.lowerBound $\leftarrow \nu(P(v_{\text{leaf}}))$
    tree.upperBound $\leftarrow |P(v_{\text{leaf}})|$
    `max_reached` $\leftarrow$ `True`
  else if $\nu(P(v_{\text{leaf}})) >$ tree.lowerBound **then**
    tree.lowerBound $\leftarrow \nu(P(v_{\text{leaf}}))$
Function `should_prune(`v`)`**:**
  if tree.lowerBound $= 0$ **then**
    return `False`
  if max_reached **then**
    $\nu(P(v)) \leftarrow |P(v_{\text{leaf}})|$
    return $\nu(P(v)) \geq$ tree.upperBound
  return $\nu(P(v)) <$ tree.lowerBound
Function `has_potential_duplicate_sums()`**:**
  return `True` **if if potential duplicate** ν **exist, else** `False`

to 18) checks if the dataset contains potential duplicate sums of potential path values $\nu_P(P(v))$, returning `True` if such sums exist. For a detailed implementation of the initial check, we refer to the GitHub repository. The combination of these functions ensures efficient pruning by leveraging both lower and upper bound criteria and optimizing the search process through specialized pruning when applicable.

These pruning methods significantly enhance the efficiency of the B&B framework by systematically eliminating nodes that cannot lead to an optimal solution, thus concentrating the search on the most promising areas of the search tree. In the following section, we will discuss the possible combinations of this thesis' application, highlighting various improvements and configurable parameters that can be set within the code. It is important to note that due to their lack of contribution to the performance or execution time in finding optimal solutions, certain functions such as the computation of remainders R or the potential value $\nu_P(P(v))$ based on the remainder value $\nu(R)$ have not been implemented. Nonetheless, these functions have been evaluated, as will be detailed in Section 6.

5.9. Strategy combinations

In this section, we demonstrated the implementation of the B&B framework, including its various phases: searching, branching, and pruning. We have incorporated several advanced strategies to efficiently compute kernels and remainders, thereby optimizing the search tree. This thesis presents a modular framework for finding an optimal solution for belief base B contraction. Specifically, we provided three different weight assignment approaches, four search strategies, and three pruner implementations (no pruning excluded). Additionally, we can compute kernels using the Expand-Shrink algorithm, employing both a sliding window technique and a divide-and-conquer strategy for both phases, the expand and shrink phase. Additionally, we are able to compute remainders R using a sliding window and divide-and-conquer strategy to determine the potential value $\nu(P(v))$. Figure 8 illustrates the different strategy options available in the current application, with the most promising strategies highlighted in green. These will be validated in the next section, which covers the evaluation.

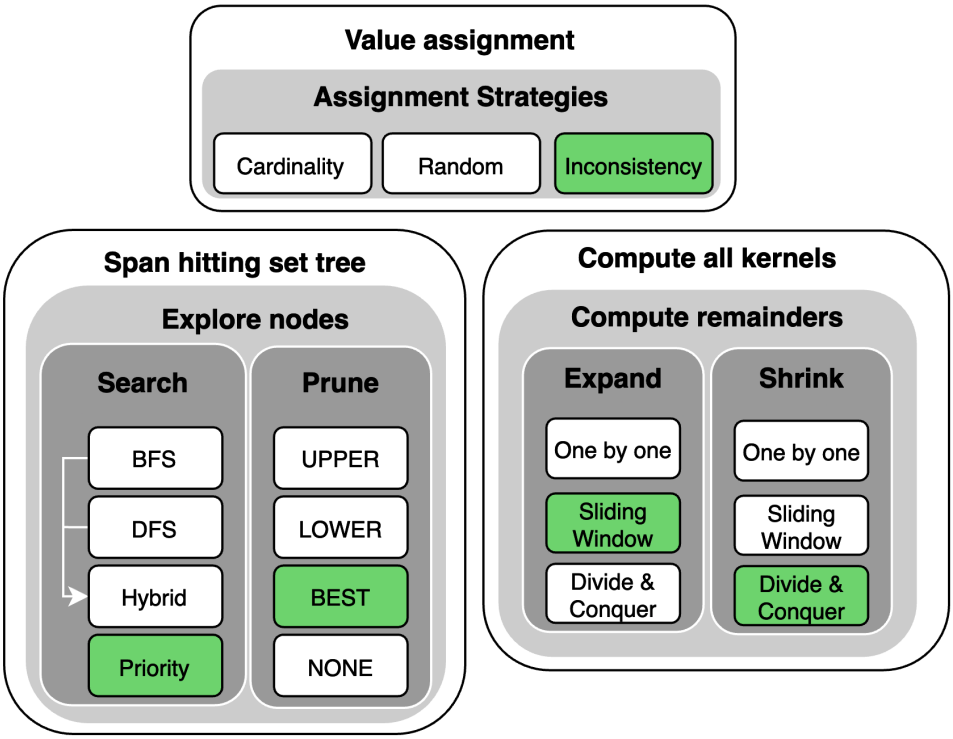


Figure 8: Application strategy combinations

It has to be noted that the pruners are related to either a minimization problem or a maximization problem. Therefore, the results of the lower pruner and the upper or best pruner shall not be compared with each other. As well known, it is a lot easier to find the optimal solution HS^* if it is determined to provide a lowest value, as

this approach does not rely on an estimation of unexplored subproblems. It is much harder to compute an optimal soultion HS^* that provides the maximal value, as the algorithm has to calculate a potential value $\nu_P(P(v))$ without the knowledge of the subproblem. It can only estimate the potential value $\nu_P(P(v))$, but it can never be certain that it reached the maximal value without having searched all subproblems. Therefore, this thesis provides the sophisticated approach to calculating inconsistency weights w_I to find an optimal solution HS^*. This approach is particularly charming, since in general $\nu(B) \neq \sum_{\beta_i \in B} w_I(\beta_i)$. Although we found some datasets where this does not apply and $\nu(B) < \sum_{\beta_i \in B} w_I(\beta_i)$. This is due to our calculation of the inconsistency weights w_I where we calculate $w_I(\beta_i) = I(B) - I(B \setminus \beta_i)$. In this case there can be instances where two elements β_i, β_j participate in the same cause of inconsistency, thereby being assigned same inconsistency weight w_I, although $I(B \setminus \{\beta_i, \beta_j\}) = I(B \setminus \beta_i) = I(B \setminus \beta_j)$.

Figure 31 visualizes the underlying approach by illustrating an exemplary hitting set tree with three assigned weights: the cardinality weight w_c, the random weight w_r, and the inconsistency weight w_I. These weights are displayed on each edge along with the edge element of the branch (or the subsequent node). At the leaf nodes, the hitting set values ν_c, ν_r, and ν_I represent the values of the hitting sets. Depending on the selected search strategy, value assignment strategy, and pruning strategy, the algorithm determines:

- **Leaf node 23:** for random weights w_r, upper pruning, and all search strategies.
- **Leaf node 4:** for cardinality weights w_c, upper pruning, and all search strategies.
- **Leaf node 14:** for inconsistency weights w_I, lower pruning, and all search strategies.

The exemplary tree in Figure 31 demonstrates that the determination of optimal solutions is highly dependent on the branching and searching strategy. The algorithm may decide between two nodes at the same level if they have equal values ν. In such cases, the algorithm employs the first-in-first-out (FIFO) principle. Different branching strategies might influence the selection of the optimal weights.

In summary, this section has detailed the comprehensive implementation of our B&B framework, encompassing weight assignment strategies, belief change algorithms, and advanced hitting set tree algorithms with diverse search strategies. By integrating these components with efficient branching and pruning methods, we have constructed a robust system capable of solving complex belief base contraction problems. The following section will present an evaluation of these implementations, providing empirical results and analysis to validate the effectiveness and performance of the proposed approaches.

6. Evaluation

This section discusses the evaluation of the proposed system. We begin by presenting the hardware setup and experimental parameters. Next, we examine the performance of our algorithms, followed by an analysis of the instances where we were able to find optimal solutions. Finally, we investigate the performance of the improved approach for calculating the potential value $\nu_P(P(v))$ from the remainder value $\nu(R)$.

6.1. Hardware Setup

The results were computed on an Ubuntu 20.04.6 LTS system (Linux kernel 5.4.0-172-generic) with Intel Core Processor (Haswell, no TSX, IBRS) 3.40-GHz CPUs and 32 GB of RAM. To limit the execution time, we used a 1800-second time limit.

6.2. Knowledge Bases

The experimental evaluation of the present B&B algorithm includes both the computation model and the verification model, which verifies if the computed solution was optimal according to the definition of the optimal hitting set. The computation model includes several advanced strategies to find the solution faster, as this is computationally hard.

As benchmarks, we use the datasets **ARG** and **SRS** from previous works by Kuhlmann and Thimm [KT21], also utilized by Niskanen et al. [NKTJ23]. A detailed description of the datasets and the inconsistency measurement algorithm can be found in [KGLT23]. The datasets include two different categories:

- **ARG**: This dataset consists of 326 knowledge bases containing individual CNF clauses of a standard SAT encoding. We set a threshold of 30 formulas for computing these knowledge bases to prevent excessive timeouts, resulting in 54 knowledge bases. We ran the B&B algorithm for all of these 54 datasets.
- **SRS**: This dataset consists of 1800 knowledge bases (KBs) that were randomly generated using *SyntacticRandomSampler* from *TweetyProject*[4]. The knowledge bases' sizes range from 5 to 15 formulas with signature size 3, to 50 to 100 formulas with signature size 30, with an average signature size of 16 and 36 formulas. A subset of 555 of these knowledge bases was evaluated.

The datasets comprise finite belief base CNF formulas involving logical operators such as AND (&&), OR (||), NOT (!), IMPLICATION ($\rightarrow$), and EQUIVALENCE ($\longleftrightarrow$). They include a mix of conjunctive normal form (CNF) and disjunctive normal form (DNF).

Table 6 shows an overview of the evaluated knowledge bases that were used for verifying the results of our application.

[4] https://tweetyproject.org/

Dataset	Signature size	Formulas per knowl. base	Knowledge bases
SRS3	3	5 - 15	200
SRS5	5	15 - 25	200
SRS10	10	15 - 25	200
ARG	10 - 30	10 - 30	28

Table 4: Overview of the sets of computed knowledge bases.

As these datasets only include the belief base without any further information, we needed to compute all hitting sets to verify if the computation model of this thesis' application found the solution that was deemed to be optimal. We refer to these runs as the baseline, as we are comparing our Branch-and-Bound (B&B) algorithm to these results. For the sake of completeness, we attempted to compute all hitting sets for a dataset with a signature size of 20 and 30 formulas, but this computation was terminated after 180 hours of runtime.

Therefore, we focused on the **SRS** dataset with signature sizes 3, 5, and 10. Each set of knowledge bases with a signature size of 3, 5, and 10 includes 200 knowledge bases, amounting to 600 knowledge bases in total. From these 600 knowledge bases, we attempted to compute all hitting sets, successfully retrieving the hitting sets for the knowledge bases with signature sizes 3 and 5 that do not entail $\alpha = (A0 \wedge \neg A0)$. This approach forces the hitting set tree algorithm to compute all possible hitting sets, as α itself is a contradiction providing an inconsistency. For the **SRS** datasets with signature size 10, we calculated all hitting sets with $\alpha = (A0 \vee A1)$, as almost all datasets with a signature size greater than 10 timed out with $\alpha = (A0 \wedge \neg A0)$.

Out of the 54 **ARG** datasets that include less than 30 lines, we ran the B&B algorithm with $\alpha = (\text{arg0} \wedge \neg\text{arg0})$, although the algorithm does not necessarily need a valid α, especially if α is a contradiction. We found that 26 of these datasets were consistent and no dataset timed out when computing the baseline. From the resulting 28 knowledge bases, we were able to compute **1,691** hitting sets.

Out of the 400 knowledge bases with signature size 3 or 5, that ran with $\alpha = (A0 \wedge \neg A0)$, we found that 17 of the datasets are consistent and 149 datasets timed out, allowing us to compute the hitting sets for 234 of these knowledge bases, which in total provided **87,983** hitting sets. Out of the 200 knowledge bases with signature size 10, computed with $\alpha = (A0 \vee A1)$, 46 knowledge bases timed out, allowing us to use 154 knowledge bases to compute **33,079** hitting sets for these knowledge bases.

Table 5 shows the total amount of computed knowledge bases and hitting sets that can be used to verify the instances where our application was able to compute the optimal solution.

On the one hand, we expect the B&B algorithm to solve more knowledge bases than the baseline because with the improvements we expect the algorithm to compute an optimal solution before the timeout. On the other hand, we expect the lower and best pruner to always output the optimal solution, which we will verify for those knowledge bases that provided all hitting sets as shown in Table 5.

Dataset	Timeouts	# KBs	# hitting sets	--alpha
SRS3	27	156	14,493	$(A0\&\&!A0)$
SRS5	122	78	73,490	$(A0\&\&!A0)$
SRS10	46	154	33,079	$(A0\|\|A1)$
ARG	0	28	1,691	$(arg0\&\&!arg0)$
Total	196	416	122,753	

Table 5: Overview of evaluated hitting sets

Figure 9 illustrates the number of timeouts and consistent datasets per knowledge base, showing that for the knowledge bases with signature size 5 over 50% of the runs timed out.

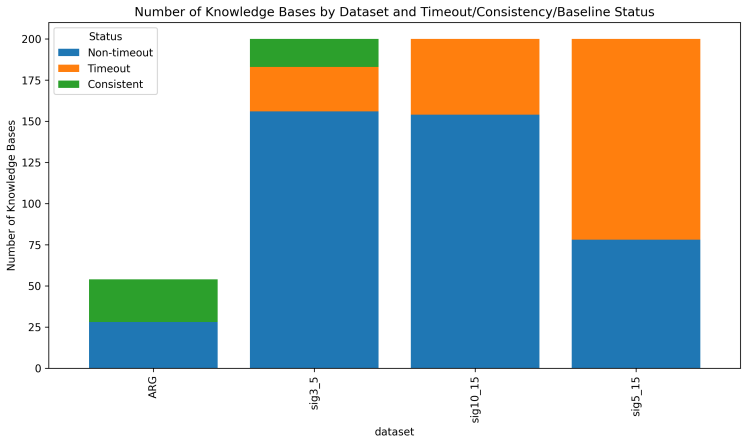


Figure 9: Number of timeouts per knowledge base

In addition to the consistent datasets, we discovered that for 25 of the knowledge bases with a signature size of 3 and 17 of the knowledge bases with a signature size of 5, the sum of the inconsistency weights w_I was 0. This means that each element β_i of these knowledge bases had an assigned inconsistency weight w_I of 0, even though the initial inconsistency value $\nu_I(B)$ was greater than 0. This occurs because none of the formulas in the knowledge base individually participate in the inconsistency, but rather it is a combination of at least two elements that contributes to the inconsistency.

For the remainder of the evaluation, we will separate the results into two phases. In the first phase, we will evaluate the performance of the B&B algorithm, verifying the instances of computed optimal solution to assess whether the algorithm works as intended. We will also compare the computation of the B&B algorithm with the baseline to evaluate the execution time compared to the baseline, where the entire search space was explored. In the second phase, we will evaluate the advancements implemented in this thesis, including the advanced kernel find algorithm (see 2), the

computation of the remainder value (4.6.2) and the implemented search strategies (5.6), to understand how these advanced strategies influence the performance of the B&B algorithm. We will only evaluate these advanced strategies for those baselines that did not time out to prevent excessive computation.

6.3. Branch-and-Bound performance

In this section we will outline the performance of the B&B performance with regards to the execution time, the computed kernels, the number of branches and the number of pruned branches. We expect the execution time to rapidly decrease with a rising number of pruned branches, as the algorithm does not compute the kernels for pruned branches leading to less sat solver calls that should improve the overall performance of the application in terms of the execution time.

Before we evaluate the computation of the optimal solution, we will recall the definition of the optimal solution for the three implemented pruner.

LOWER The lower pruner computes optimal solutions with a maximal value ν. Therefore, we are comparing the computed optimal solution of the lower pruner with the highest value ν with the optimal solution computed by the baseline with the highest value ν. In the cases where both scores are identical, we assume that the lower pruner found the optimal solution.

UPPER The upper pruner computes optimal solutions with a minimal value ν. Therefore, we are comparing the computed optimal solution of the upper pruner with the lowest value ν with the optimal solution computed by the baseline with the lowest value ν. In the cases where both scores are identical, we assume that the upper pruner found the optimal solution.

BEST The best pruner computes optimal solutions with a maximal value ν and minimal cardinality $|HS|$. Therefore, to compare the results from the best pruner to the optimal solution of the baseline, we are comparing the maximal values ν and find the solution with the lowest cardinality over all results with the maximal value ν_{max}.

6.3.1. Computing Optimal Solutions - Verification

In this section we will cover all instances where the optimal solution was found. It is important to include the constraint of the found optimal solution because, as mentioned previously, otherwise the pruner with the aggressive pruning approaches would be deemed as performing the best, but they were not even able to compute the optimal solution.

We evaluated the calculation of the optimal solution with the aforementioned pruners with none of the advancements activated mentioned in Section 5.4. Table 6

shows how many cases the application was able to find the optimal solution for the specified knowledge bases. The columns are specified as follows:

- **Opt. Max:** specifies the cases where $HS^* = max(\nu)$
- **Opt. Min:** specifies the cases where $HS^* = min(\nu)$
- **Opt. Best:** specifies the cases where $HS^* = max(\nu), min(|HS^*|)$
- **c | r | I:** denotes the cardinality (c), random (r), and Inconsistency (I) approach

Dataset	KBs	Opt. Max #			Opt. Min #			Opt. Best #		
		c	r	I	c	r	I	c	r	I
SRS3	156	60	32	131*	156	156	131*	156	156	131**
SRS5	78	64	66	60	78	78	78	78	78	69**
SRS10	154	61	62	67	154	154	154	154	154	154**
ARG	28	28	28	27*	28	28	28	28	28	27**

Table 6: Overview of the sets of knowledge bases and the instances where the optimal solution was found

In Table 6, we marked the instances of the best pruner with * where it was not able to compute all an optimal solution due to the sum of the inconsistency weights $w_I = 0$. This is due to the computation of individual inconsistency weights w that do not provide an inconsistency measure greater than 0. In this case, the assignment of the inconsistency weights w_I assigns the value 0 to each element of B even though $I(B) > 0$. Therefore, the best pruner finds the optimal solution in 100% of the cases, as we do not count the cases where $\sum_{\beta_i \in B} w_I(\beta_i) = 0$ applies.

As one can see from the table, the overall performance of the lower pruner is rather poor. This is because we implemented the lower pruner with a very aggressive pruning approach, where the potential value $\nu_P(P(v))$ is simply the sum of all weights w of the remaining dataset $B(v) = B \setminus P(v)$. This often leads to the pruning of a branch if the first branches of the tree do not contain elements with a high weight w. This problem could be addressed by rearranging the search tree with a different root node v_{root}, where the elements β of the kernel k have higher weights w. Nonetheless, this aggressive pruning allows a better comparison with the best pruner, where the pruning decision is rather conservative, as the best pruner does not prune any branch if the $max(\nu)$ was not reached. In the remainder of this section, we will show how the aggressive pruning approach performs compared to the best pruner. We expect that the lower pruner with its aggressive approach at least outperforms the best pruner in terms of execution time.

Table 7 shows the relative performance of the different approaches and pruners. As we mentioned earlier, for some instances, the inconsistency approach did not calculate an inconsistency weight $w > 0$. We did not compute the optimal solution

for these instances, as this would just be the same as the baseline. This issue could be addressed by calculating different inconsistency weights w, like the Shapley value (see **??**). Since we know that the optimal solution would have been found for the inconsistency approach when $\sum_{\beta_i \in B} w_I = 0$, we treated these instances as if the optimal solution was found.

Dataset	KBs	Opt. Max %			Opt. Min %			Opt. Best %		
		c	r	I	c	r	I	c	r	I
SRS3	156	38	21	77	100	100	100	100	100	100
SRS5	78	86	85	77	100	100	100	100	100	100
SRS10	154	40	40	44	100	100	100	100	100	100
ARG	28	100	100	100	100	100	100	100	100	100

Table 7: Overview of the relative performance of the algorithm in finding the optimal solution

We see that the best pruner finds the optimal solution in 100% of the cases, even if the optimal solution is deemed to have the highest value. It is important to note that this table covers the cases where the entire search space needed to be searched (worst case). We also tried to use a more aggressive pruning approach that implements the logic of the upper lower pruner, but this approach only works if the optimal solution is not characterised by the maximum value and minimum cardinality but solely by the maximum value. Despite this, the best pruner will outperform the baseline, even with the conservative pruning approach. This will be covered in the next section.

6.3.2. Comparative Performance Analysis

In this section, we will evaluate the performance of the algorithm and compare the results to the computed baseline. The baseline is defined as a run where the algorithm did not prune any branches and searched the entire search space of the hitting set tree. This includes instances that timed out, where we cannot estimate the real runtime of the instance. Generally, we would not be able to verify that our approaches computed optimal solutions if the baseline timed out. Nonetheless, in the previous section we saw that the best and upper pruners do compute optimal solutions in every instance, such that these two pruners could be evaluated without further verification. This means that these pruners could be used without the need to compute all hitting sets beforehand. The lower pruner employs a very aggressive pruning approach that prunes every branch that might not yield a solution with a higher value than the current lower bound. Therefore, we will not evaluate the instances of the lower pruner where we could not verify the computed solution to be optimal.

To show the difference in performance in Figure 10 we are displaying all instances, regardless of the verification that the optimal solution was found. Opposed to this

Figure 11 shows the performance of all instances that found the optimal solution compared to the baseline. We do see that when the optimal solution was found, the difference to the baseline decreased.

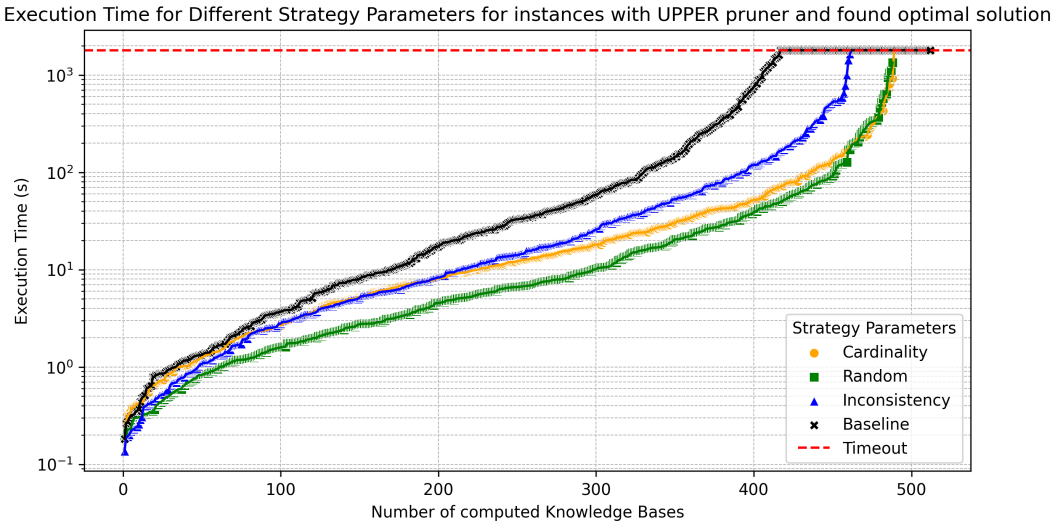


Figure 10: Performance of all pruners without optimal solution constraint

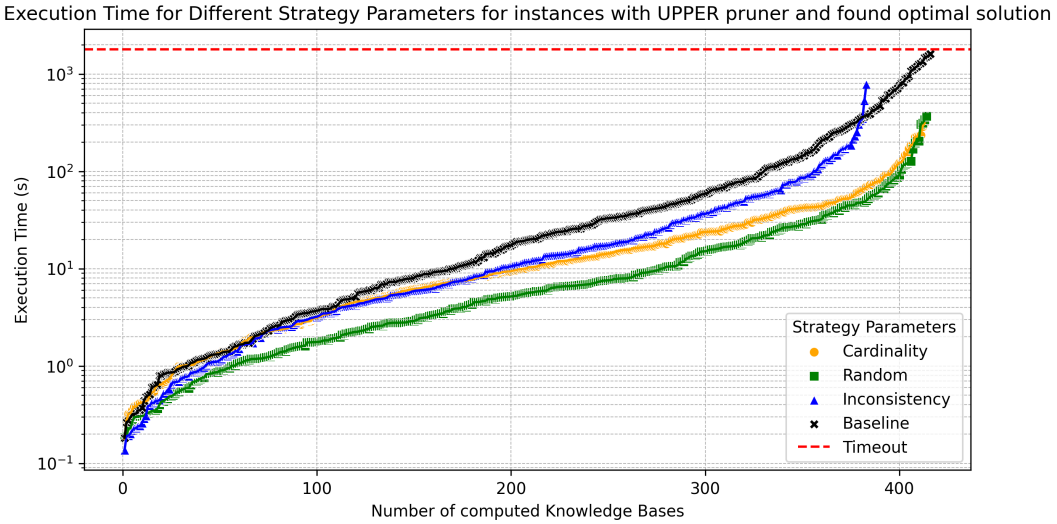


Figure 11: Performance of all pruners with optimal solution constraint

For a more detailed analysis, we will now evaluate the performance of the application for each pruner and weight assignment approach separately. First, we evaluated the performance of the different weight assignment approaches with the three pruner settings. It is important to note that we evaluated all three approaches

for the lower and upper pruner. With regards to the best pruner, we only compare the inconsistency approach to the baseline. We did not evaluate the random and cardinality approach for the best pruner, as these two approaches provide the baseline results in the worst case scenario, where the whole search space needs to be unfolded.

The first takeaway is that we see that the performance of the pruning approaches differ from each other; Figure 12 shows that the lower pruner only outperforms the baseline with the inconsistency approach, as the inconsistency approach was the only approach besides the baseline that found optimal solutions in 100% of the instances. For the cardinality and random approach, the algorithm is less efficient, as it does not find the optimal solution due to the lack of knowledge about the value of the subproblem.

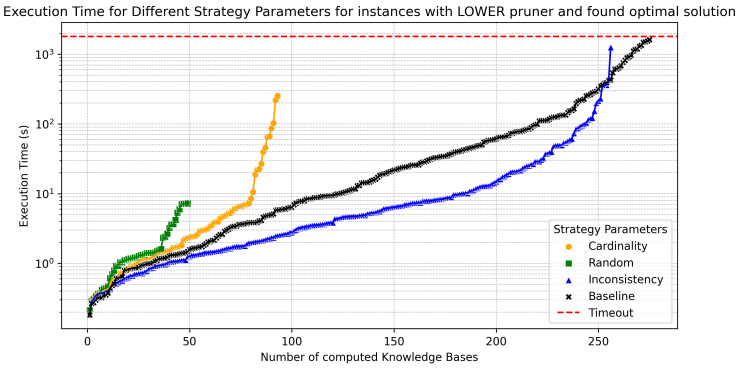


Figure 12: Lower pruner performance for found optimal solutions

However, the algorithm outperforms the baseline for all instances in which the optimal solution was considered to provide the lowest value. Figure 13 meets our expectations and the algorithm was able to find the solution with the lowest value in less time compared to the baseline. In Figure 13 we have included the instances of the baseline that timed out to show the improvements of our approach. We can also see that the random approach and the cardinality approach found the optimal solution in the same number of instances, but the random approach was more efficient. This is not surprising as all runs used the Priority-Based Search that is more efficient the more the weights of the elements differ from each other. Therefore we can see that the random approach offers the best performance. The inconsistency approach was unable to compute the same amount of instances as the baseline or the cardinality and random algorithms, as for the inconsistency approach we did not compute the instances where $\nu(B) = \sum_{\beta_i \in B} w_I(\beta_i) = 0$.

Figure 13 shows the execution time of the upper pruner over the number of instances, wherein in all instances an optimal solution was computed.

The results for the upper pruner are not surprising, as the upper pruner finds the optimal solution with the lowest value $min(\nu)$. This task is not considered hard as the search algorithm can explore the tree based on the lowest path values $\nu(P(v)$,

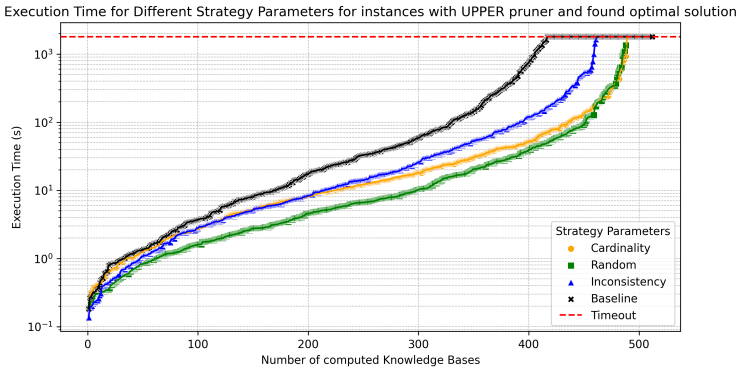


Figure 13: Upper pruner performance for found optimal solutions

such that this approach does not even face the challenge of computing the potential value ν_P of the unexplored subproblems.

Figure 14 shows the performance of the best pruner by comparing the inconsistency approach to the baseline.

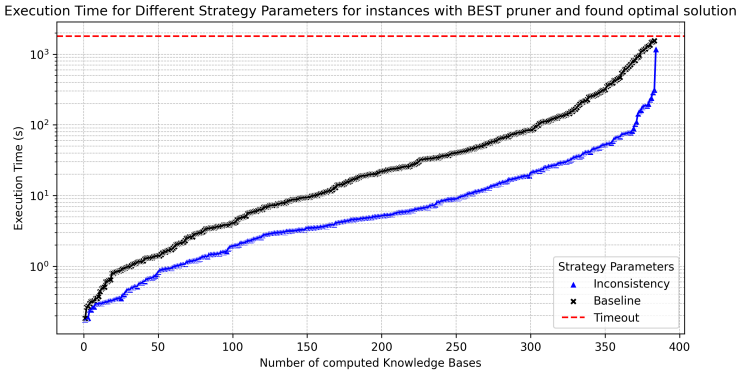


Figure 14: Best pruner performance for found optimal solutions

It is encouraging to see that the B&B algorithm outperforms the baseline with the inconsistency weight assignment, where it was able to find 100% of the optimal solutions in less time. It is important to note that in Figure 14 we limited the baseline to all instances where the inconsistency approach was able to compute an optimal solution, so all instances where $\nu(B) = \sum_{\beta_i \in B} w_I(\beta_i) > 0$. This also allows for a better resolution of the time differences between the two runs.

The time difference is mainly based on the pruned branches and therefore on the lower amount of computed kernels, which is shown in Figure 15. In this graph, we can see that the higher the number of kernels of the baseline, the larger the difference between the number of kernels. This means that our approach performs better with increasing number of kernels and therefore also bigger or more complex knowledge bases.

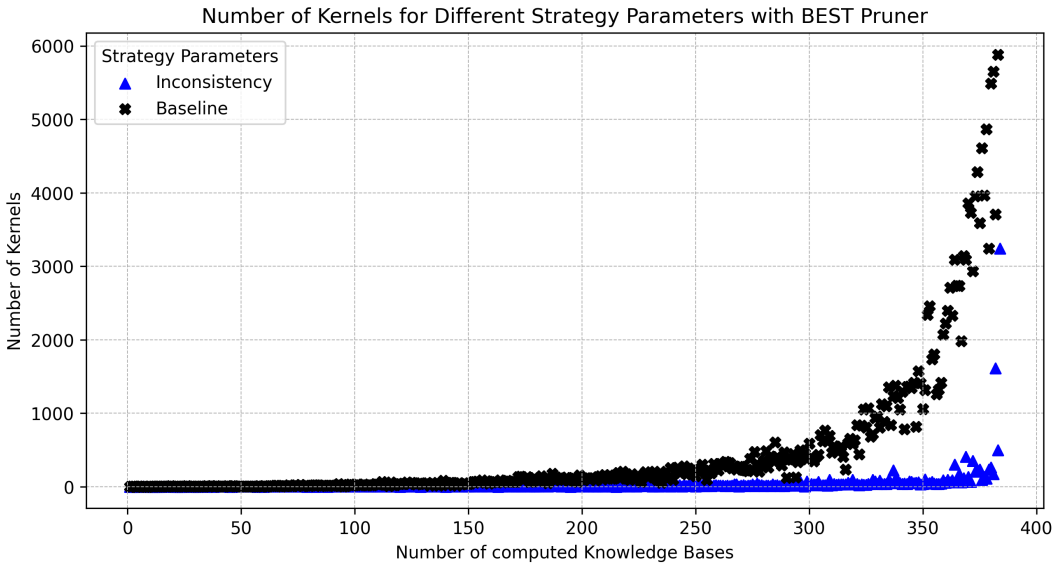


Figure 15: Number of kernels of baseline and inconsistency approach

Figure 16 shows the distribution of the relative time difference between the two approaches. We can see that in almost all cases the relative time difference was > 50%. Instances in which the baseline was faster (indicated by positive time differences) can be linked to knowledge bases with a small number of formulas.

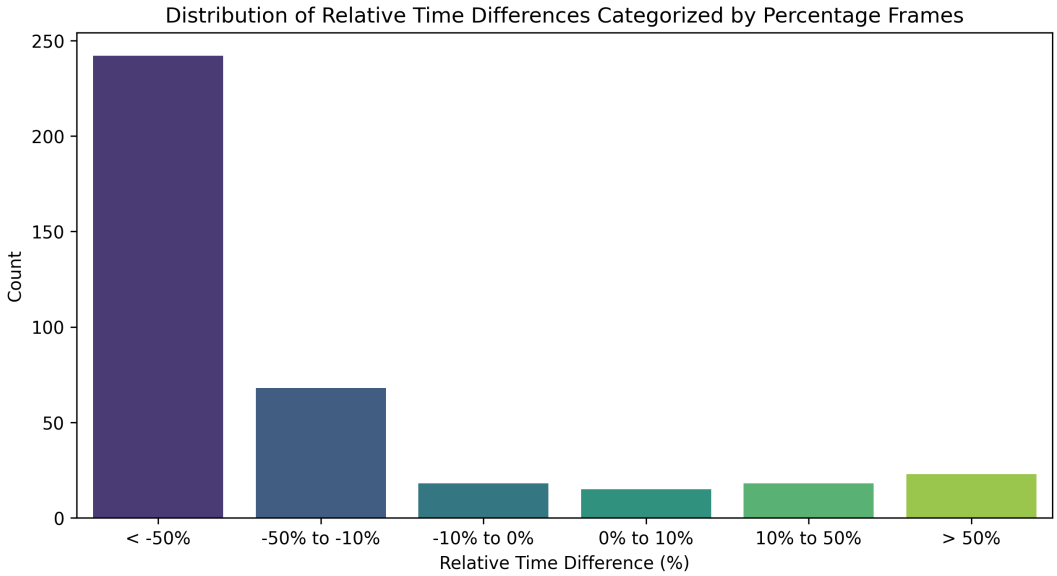


Figure 16: Relative time difference between baseline and inconsistency approach

In the next section, we will evaluate the performance of the alternating strategies used to compute kernels k as described in Section 5.4.1.

6.4. Advanced Find Kernel and Remainder Algorithm

In this section we will evaluate the influence of the Find Kernel Algorithm 2 and the Find Remainder Algorithm 7 on the performance of this application. First, we will evaluate the implemented strategies like divide-and-conquer and sliding window for the expand and shrink phase as described in Section 5.4.1 and see how this affects the performance and especially the runtime of the algorithm.

In general, we have computed the following combinations, where 0 indicates the strategy being deactivated (False) and 1 denotes the strategy being activated (True):

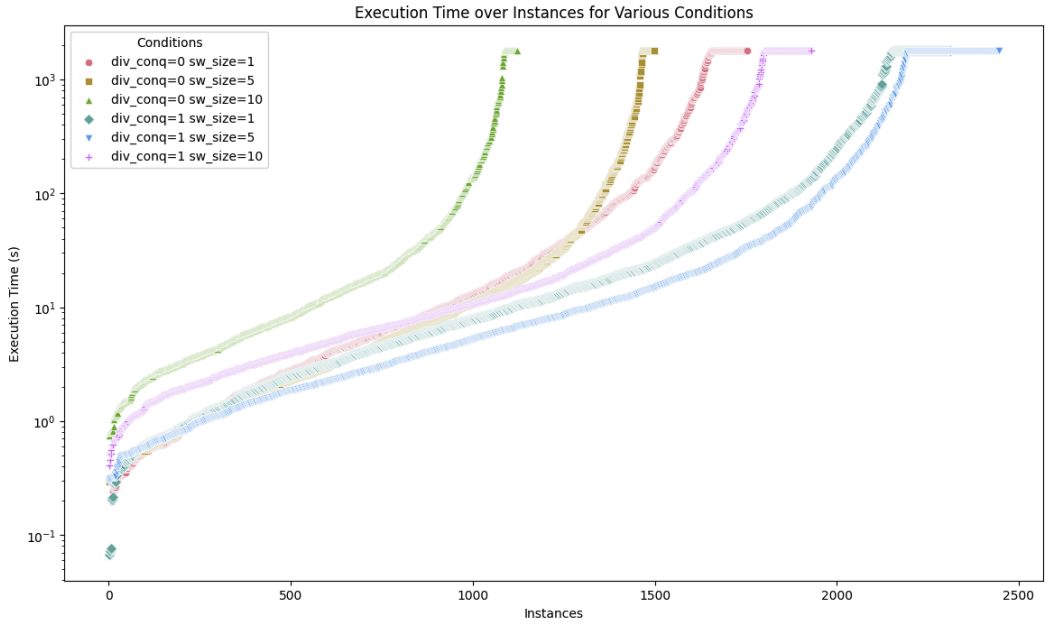


Figure 17: Comparison of execution time for computing kernels strategies

The results used to produce Figure 17 were aggregated and a mean values were calculated that are shown in Table 8.

In this table, we see that we achieved the best results with the sliding window size 5 and without using the divide-and-conquer strategy. Usually, divide-and-conquer is expected to produce better results in combinatorial problems, but we suggest that in the field of finding kernels, where a SAT-solver is called to evaluate the satisfiability of a set, it can be faster to determine the satisfiability of bigger sets. Therefore, it might be the case that divide-and-conquer performs worst as it splits the datasets into two halves such that the SAT-solver had to determine the satisfiability for smaller sets. This assumption will be supported by the following Figures 18 and 19 where we see that with the divide-and-conquer technique we were able to

Condition		Mean Execution Time (s)
div_conq	sw_size	
0	1	153.79
0	5	99.86
0	10	107.25
1	1	159.18
1	5	110.35
1	10	106.06

Table 8: Mean Execution Times for Different divide-and-conquer and sliding window Combinations

compute more instances before timing out. This means that the divide-and-conquer technique shows improved performance for larger datasets, since we assume that the execution time increases with the signature size and number of formulas.

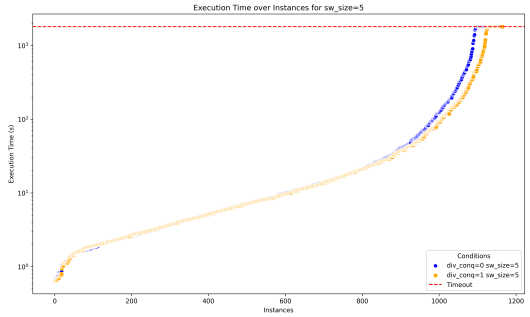


Figure 18: Detailed comparison of divide-and-conquer for sliding window size 5

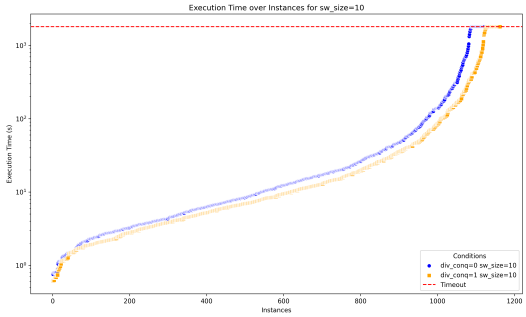


Figure 19: Detailed comparison of divide-and-conquer or sliding window size 10

Conclusively, we showed that the developed advanced strategies, namely divide-and-conquer and sliding window do outperform the baseline, which is represented by expanding and shrinking the knowledge base by one formula at a time. In comparison between all possible combinations we saw that a bigger window size results in up to 50% less execution time. The next section covers the performance of the computation of remainders for the remainder value.

6.4.1. Computing Remainder Value

In this section, we evaluate the computation of the remainder value as described in Section 4.6.2. We ran the sig3 dataset with the lower pruner and the random weight assignment, as this pruner has the most potential for improvement. Figure 20 shows that both approaches perform similarly, although in some instances, the remainder value computation took more time to compute the hitting sets.

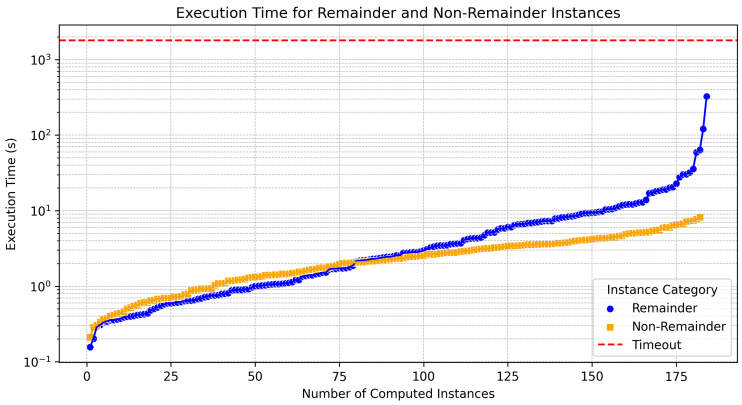


Figure 20: Comparison of execution time for computing hitting sets with and without remainder value

We also compared both approaches in terms of computed kernels (Fig. 21) and the number of pruned branches (Fig. 22). In some instances, the number of computed kernels is higher, which correlates with an increase in pruned branches. This illustrates the variability of this approach, aligning with the evaluation by Resina et al. [RRW14], where the computation of remainders either outperformed or underperformed the computation of kernels. This variability is reflected in our approach by a higher number of pruned branches in some cases and more time-intensive computations in others. It would be interesting to further compare instances where the remainder computation outperforms the kernel computation to understand its impact on the execution time of the algorithm.

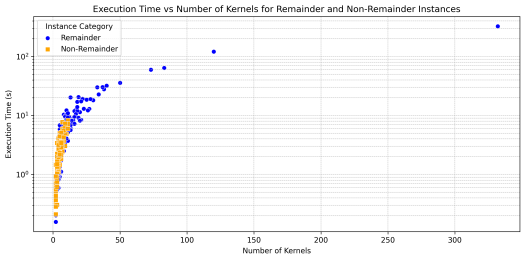


Figure 21: Execution time comparison of remainder computation

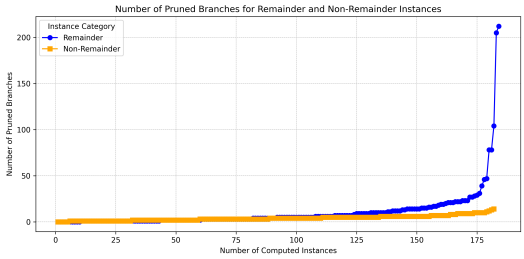


Figure 22: Number of pruned branches in remainder computation

Furthermore, the remainder value computation approach found fewer optimal solutions, as illustrated in Figure 23. This may be due to the higher number of pruned branches, where the remaining value could cause pruning of branches that include subproblems with higher values ν. It is important to note that the lower pruner performed poorly regardless of the value computation approach. The calculation of the remainder value $\nu(R)$ uses a more aggressive pruning approach, increasing the likelihood of pruning branches that contain higher values.

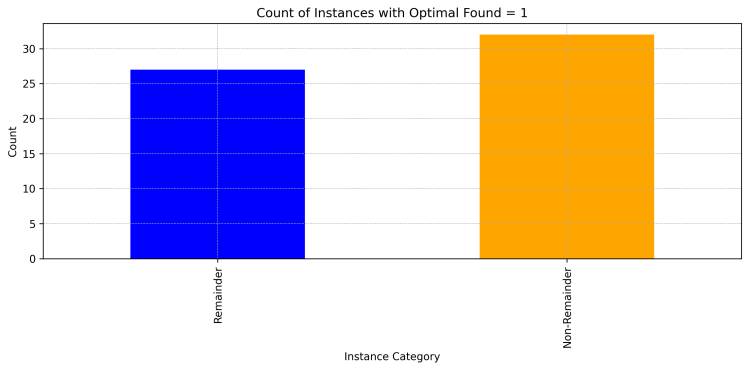


Figure 23: Comparison of optimal solutions found with remainder computation

In summary, we did not find instances where the remainder computation value significantly outperformed the algorithm that counts the value of a node's subproblem as the sum of all values, regardless of their participation in the hitting set of the node's subproblems. This might be because this approach not only computes the kernel of the node but also a remainder, resulting in more calls to the SAT solver. Nonetheless, this approach could be worth exploring in future work, particularly if the computed remainders are used not only to calculate the remainder value $\nu(R)$ but also to span subproblems, such as $HS = B \setminus R$.

6.4.2. Search Strategy Comparison

In this section, we evaluate the impact of the search strategy on the performance of the algorithm in terms of the execution time and the instances where the optimal solution was found. We ran the algorithm on a subset of knowledge bases from the sig3 and sig5 with the inconsistency value ν_I and the three pruners. In this evaluation, the Priority-Based Search serves as the baseline, as we want to find out if one of the other searches can outperform the Priority-Based Search, that is deemed to explore the fewest number of subproblems, according to the literature [MJSS16].

Figures 24, 25, 26 show the execution time over the number of computed knowledge bases for each pruner. The graphs show that the search strategies do perform similarly in terms of the execution time, although it is interesting to see that the Hybrid-Search (HYS) did perform better when used in combination with the lower or upper pruner. It seem as if our assumption was correct that combining both advantages of DFS and BFS can yield to better results. We interpret this result in such a way that the algorithm quickly sets a bound and therefore prunes branches earlier. Another encouraging outcome is that the HYS operated well within a managable range wherein the other searches had instances with longer execution time.

In the next step, we are comparing the instances where the optimal solution was found for each search strategy. This is shown in Figure 27. Again, we do see very similar results, although it appears that the Priority-Based Search is slightly faster

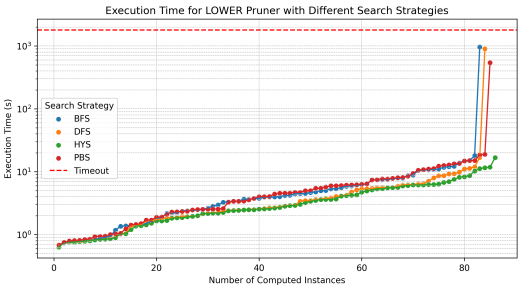


Figure 24: Execution time of lower pruner with all search strategies

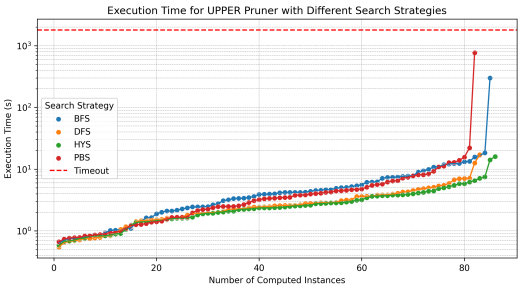


Figure 25: Execution time of upper pruner with all search strategies

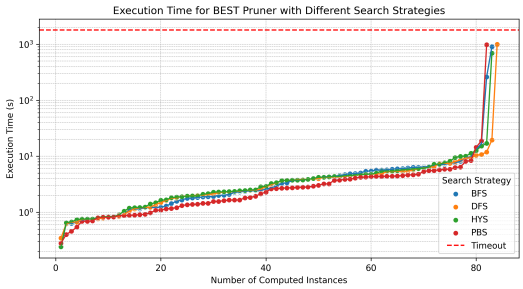


Figure 26: Execution time of best pruner with all search strategies

than the other search strategies when finding the optimal solution with the highest value. It is interesting to see that in our algorithm the BFS did also perform well in terms of finding optimal solutions. For the best pruner the searches almost always found the same number of optimal solutions.

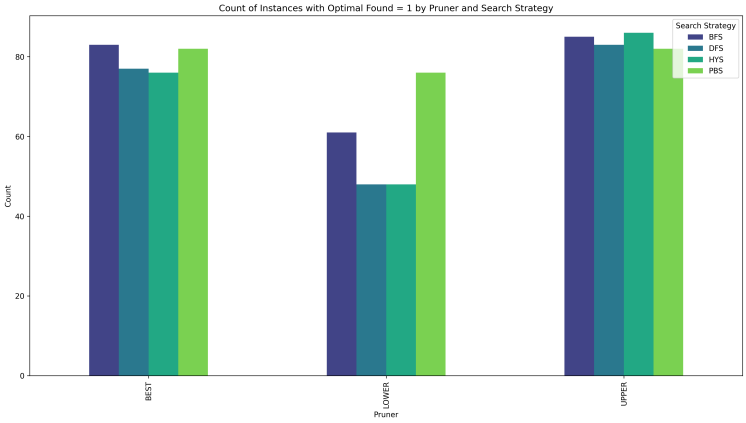


Figure 27: Comparison of optimal solutions found with remainder computation

In summary, we can confirm that the search strategy that implements a best-first approach (PSB) offers the most promising results. In combination with the pruner approach and the value assignment approach the Priority-Based Search performed the best in finding the solution with the highest value (lower pruning). This is an encouraging result, as these instances serve the hardest approach in terms of the pruning decision.

7. Conclusion

With the rapid advancements in artificial intelligence, AI agents increasingly gather vast amounts of information that may contradict existing beliefs, leading to inconsistencies within their belief bases. Therefore, efficient heuristics for belief base revision are crucial. When adapting a belief base to resolve inconsistencies, selecting the right subset of data for contraction is essential. This can be addressed through applications and heuristics that identify optimal solutions for belief base contraction.

Current literature predominantly offers algorithms and advancements for computing maximal inconsistent subsets, known as kernels, or maximal consistent subsets, known as remainders. Most research focuses on Description Logic (DL) or Web Ontology Language (OWL), with limited attention given to Classical Propositional Logic (CPL) (see Section 3). Furthermore, there is a scarcity of research on qualification approaches that add a layer to the algorithms, determining a value or score for potential solutions in belief base contraction. The goal is to not only identify possible solutions but also to find the one that best meets certain criteria, such as maximal inconsistency or minimal disruption to the belief base.

Our findings provide a heuristic to characterize possible solutions for belief base contraction and propose algorithms to find these optimal solutions more efficiently. By introducing the inconsistency value ν_I, we developed a technique for ranking elements of a belief base in terms of inconsistency. This approach is particularly valuable because it can be applied to all inconsistent belief bases where $I(B) \geq 0$.

We also demonstrated that the introduced values ν can be utilized to implement a Branch-and-Bound strategy, pruning branches during the computation of hitting sets HS, which are potential solutions for belief base contraction. Our approach can identify optimal solutions HS^* in less than 50% of the time required compared to exploring the entire search space of a hitting set tree HST.

Additionally, we introduced advancements like the divide-and-conquer and sliding window techniques, enhancing known algorithms for computing kernels and remainders. These techniques allowed us to compute kernels with an average time saving of 50% compared to existing algorithms. We also showed the impact of different search strategies for spanning a hitting set tree, verifying that a best-first approach yields the most effective results.

Lastly, we proposed further advancements for improving known Branch-and-Bound techniques by introducing the computation of a remainder value $\nu(R)$, which helps calculate a potential value ν_P for a node's subproblem, enhancing the performance of the Branch-and-Bound algorithm. Our results indicated that this approach leads to higher pruning rates, although it comes at the cost of increased computation time. Thus, further research is needed to explore these findings in greater detail.

Outlook

For future work, there are four promising research directions:

1. **Improving Weight Assignment Approaches**: We could enhance the weight assignment by incorporating other inconsistency measures, e.g. as provided by Kuhlmann and Thimm [KT21]. Another approach is the Shapley value that was provided by Hunter and Konieczny [HK10], which computes inconsistency measures at the formula level. Recent research on atom-centric inconsistency measures [GH23] presents another promising approach.

2. **Exploring Remainders for Hitting Set Trees**: Further investigation into computing remainders and using them to span hitting set trees, rather than computing kernels for each node, could lead to significant improvements. This area, being less explored, offers high potential for advancements. Resina et al. [RRW14] showed that remainders can outperform kernels in some instances, although predicting these cases beforehand remains challenging.

3. **Enhancing Hitting Set Tree Generation**: The performance of weight assignment approaches and search strategies is significantly influenced by the sequence in which kernels are computed. This is particularly evident in the lower pruner approach, where the optimal solution HS^* may not be found or might even be pruned if the initial branches stemming from the root node v_{root} have low values (as illustrated by the branch $\neg A0$ or the hitting set tree in Figure 31). Consequently, a promising research direction would be to identify an optimal root node v_{root} or to explore rearranging the hitting set tree HST to improve performance.

4. **Enhancing Pruning Rules for Branch-and-Bound**: Our remainder value approach suggests that there is room for improving pruning rules. Future research could focus on enhancing pruning techniques not only for minimization problems [BFSW23] but also for maximality problems. Developing heuristics to determine subproblem values could significantly improve the performance of lower pruners, particularly for computing optimal solutions with maximum values.

References

[AGM85] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *The Journal of Symbolic Logic*, 50(2):510–530, 1985. Publisher: Association for Symbolic Logic.

[AM85] Carlos E. Alchourrón and David Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, 44(4):405–422, December 1985.

[BFSW23] Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. An Efficient Branch-and-Bound Solver for Hitting Set, September 2023.

[BHV21] Handbook of Satisfiability, April 2021.

[CW15] Raphael Cóbe and Renata Wassermann. Ontology repair through partial meet contraction. In *Proceedings of the 2015 International Conference on Defeasible and Ampliative Reasoning - Volume 1423*, DARe'15, pages 9–15, Aachen, DEU, July 2015. CEUR-WS.org.

[Dix94] S. Dixon. Belief Revision: A Computational Approach. 1994.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, July 1985.

[DPF+05] Li Ding, Rong Pan, Tim Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and Ranking Knowledge on the Semantic Web. *Proceedings of the 4th International Semantic Web Conference*, pages 156–170, November 2005.

[DW93] S. Dixon and W. Wobcke. The implementation of a first-order logic AGM belief revision system. In *Proceedings of 1993 IEEE Conference on Tools with Al (TAI-93)*, pages 40–47, November 1993. ISSN: 1063-6730.

[FH11] Eduardo Fermé and Sven Ove Hansson. AGM 25 Years: Twenty-Five Years of Research in Belief Change. *Journal of Philosophical Logic*, 40(2):295–331, 2011. Publisher: Springer.

[FH18] Eduardo Fermé and Sven Ove Hansson. Belief Bases. In Eduardo Fermé and Sven Ove Hansson, editors, *Belief Change: Introduction and Overview*, SpringerBriefs in Intelligent Systems, pages 49–57. Springer International Publishing, Cham, 2018.

[Fuh96] André Fuhrmann. *An Essay on Contraction*. Studies in Logic, Language, and Information. Center for the Study of Language and Information, June 1996.

[GH11] John Grant and Anthony Hunter. Measuring Consistency Gain and Information Loss in Stepwise Inconsistency Resolution. In Weiru Liu, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 6717, pages 362–373. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[GH23] John Grant and Anthony Hunter. Semantic inconsistency measures using 3-valued logics. *International Journal of Approximate Reasoning*, 156:38–60, May 2023.

[Gui20] Ricardo Guimaraes, Ferreira. Modularity in Belief Change of Description Logic Bases. 2020.

[Gä88] Peter Gärdenfors. Knowledge in Flux. Modeling the Dynamics of Epistemic States, 1988.

[Han94] Sven Ove Hansson. Kernel Contraction. *The Journal of Symbolic Logic*, 59(3):845–859, November 1994. Publisher: Association for Symbolic Logic.

[Han99] Sven Ove Hansson. *A Textbook of Belief Dynamics*. 1999.

[HK10] Anthony Hunter and Sébastien Konieczny. On the measure of conflicts: Shapley Inconsistency Values. *Artificial Intelligence*, 174(14):1007–1026, September 2010.

[Hor11] Matthew Horridge. Justification Based Explanation in Ontologies, 2011.

[JBQ19] Qiu Ji, Khaoula Boutouhami, and Guilin Qi. Resolving Logical Contradictions in Description Logic Ontologies Based on Integer Linear Programming. *IEEE Access*, 7:71500–71510, 2019. Conference Name: IEEE Access.

[JQH09] Qiu Ji, Guilin Qi, and Peter Haase. A Relevance-Directed Algorithm for Finding Justifications of DL Entailments. In Asunción Gómez-Pérez, Yong Yu, and Ying Ding, editors, *The Semantic Web*, Lecture Notes in Computer Science, pages 306–320, Berlin, Heidelberg, 2009. Springer.

[Jun01] Ulrich Junker. QuickXPlain: Conflict Detection for Arbitrary Constraint Propagation Algorithms. October 2001.

[Kal06] Aditya Anand Kalyanpur. Debugging and Repair of OWL Ontologies. July 2006.

[KGLT23] Isabelle Kuhlmann, Anna Gessler, Vivien Laszlo, and Matthias Thimm. Comparison of SAT-based and ASP-based Algorithms for Inconsistency Measurement, April 2023. arXiv:2304.14832 [cs].

[KPHS07] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding All Justifications of OWL DL Entailments. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, Lecture Notes in Computer Science, pages 267–280, Berlin, Heidelberg, 2007. Springer.

[KT21] Isabelle Kuhlmann and Matthias Thimm. *Algorithms for Inconsistency Measurement using Answer Set Programming*. November 2021.

[LD60] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[LW66] E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, August 1966.

[MJSS16] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, February 2016.

[Moo10] Kodylan Moodley. DEBUGGING AND REPAIR OF DESCRIPTION LOGIC ONTOLOGIES, December 2010.

[NKTJ23] Andreas Niskanen, Isabelle Kuhlmann, Matthias Thimm, and Matti Järvisalo. Computing MUS-Based Inconsistency Measures. In Sarah Gaggl, Maria Vanina Martinez, and Magdalena Ortiz, editors, *Logics in Artificial Intelligence*, volume 14281, pages 745–755. Springer Nature Switzerland, Cham, 2023. Series Title: Lecture Notes in Computer Science.

[Pri79] Graham Priest. The Logic of Paradox. *Journal of Philosophical Logic*, 8(1):219–241, 1979.

[PSLP03] Chintan Patel, Kaustubh Supekar, Yugyung Lee, and E. K. Park. OntoKhoj: a semantic web portal for ontology searching, ranking and classification. In *Proceedings of the 5th ACM international workshop on Web information and data management*, WIDM '03, pages 58–61, New York, NY, USA, November 2003. Association for Computing Machinery.

[QHH+08] Guilin Qi, Peter Haase, Zhisheng Huang, Qiu Ji, Jeff Pan, and Johanna Völker. A Kernel Revision Operator for Terminologies - Algorithms and Evaluation - VideoLectures.NET, November 2008.

[Ref91] Paul Refenes. Reasoning and revision in hybrid representation systems by Bernhard Nevel, Springer-Verlag, Berlin, 1990, pp 270, DM 42. *The*

Knowledge Engineering Review, 6(2):132–133, June 1991. Publisher: Cambridge University Press.

[Rei87] Raymond Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Rib13] Márcio Moretto Ribeiro. Algorithms for Belief Bases. In Márcio Moretto Ribeiro, editor, *Belief Revision in Non-Classical Logics*, SpringerBriefs in Computer Science, pages 105–113. Springer, London, 2013.

[RRW14] Fillipe Resina, Marcio Ribeiro, and Renata Wassermann. Algorithms for Multiple Contraction and an Application to OWL Ontologies. *Proceedings - 2014 Brazilian Conference on Intelligent Systems, BRACIS 2014*, pages 366–371, December 2014.

[SQJH08] Boontawee Suntisrivaraporn, Guilin Qi, Qiu Ji, and Peter Haase. A Modularization-Based Approach to Finding All Justifications for OWL DL Entailments. In John Domingue and Chutiporn Anutariya, editors, *The Semantic Web*, Lecture Notes in Computer Science, pages 1–15, Berlin, Heidelberg, 2008. Springer.

[Thi19] Matthias Thimm. Inconsistency Measurement. In Nahla Ben Amor, Benjamin Quost, and Martin Theobald, editors, *Scalable Uncertainty Management*, volume 11940, pages 9–23. Springer International Publishing, Cham, 2019.

[Tse83] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[TW19] Matthias Thimm and Johannes P. Wallner. On the complexity of inconsistency measurement. *Artificial Intelligence*, 275:411–456, October 2019.

[Was99] Renata Wassermann. Resource Bounded Belief Revision. *Erkenntnis*, 50(2):429–446, May 1999.

A. Annex

A.1. UML - Advanced Expand Shrink Algorithm Implementation

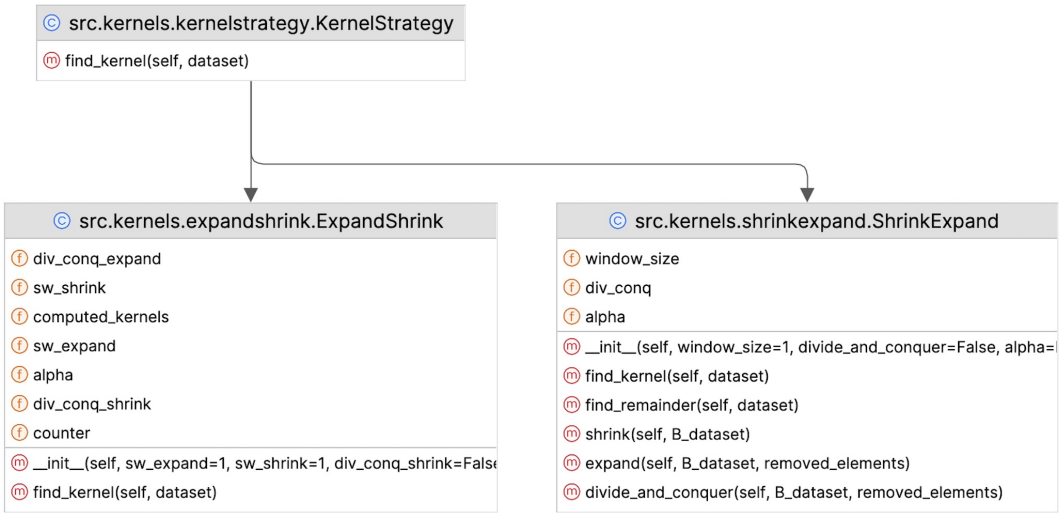


Figure 28: Class implementation of kernel finding strategy

A.2. UML - Seach Strategy Implementation

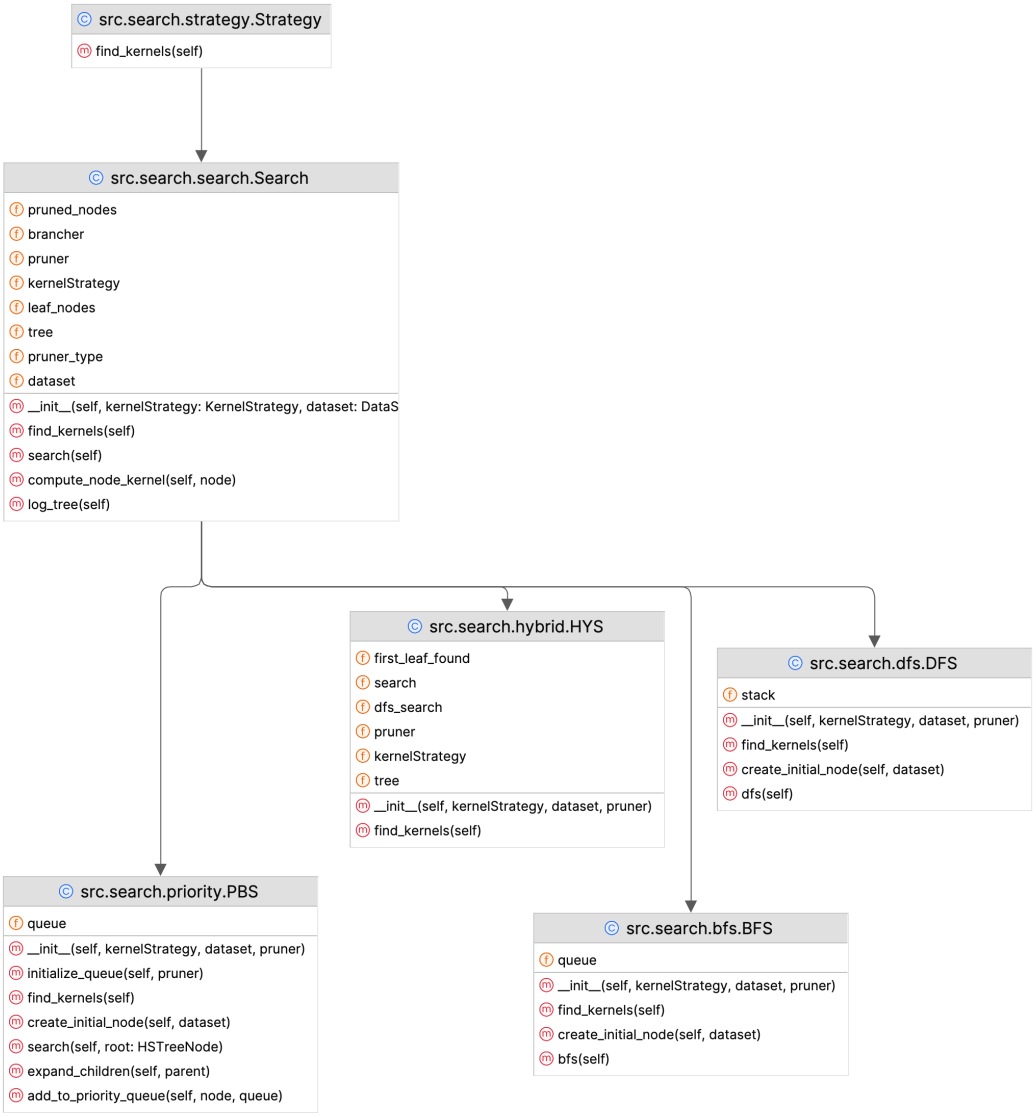


Figure 29: Class implementation of the search strategy

A.3. UML - Pruner Implementation

Figure 30: Class implementation of the Pruners

A.4. Example Hitting Set Tree with Assigned Weights

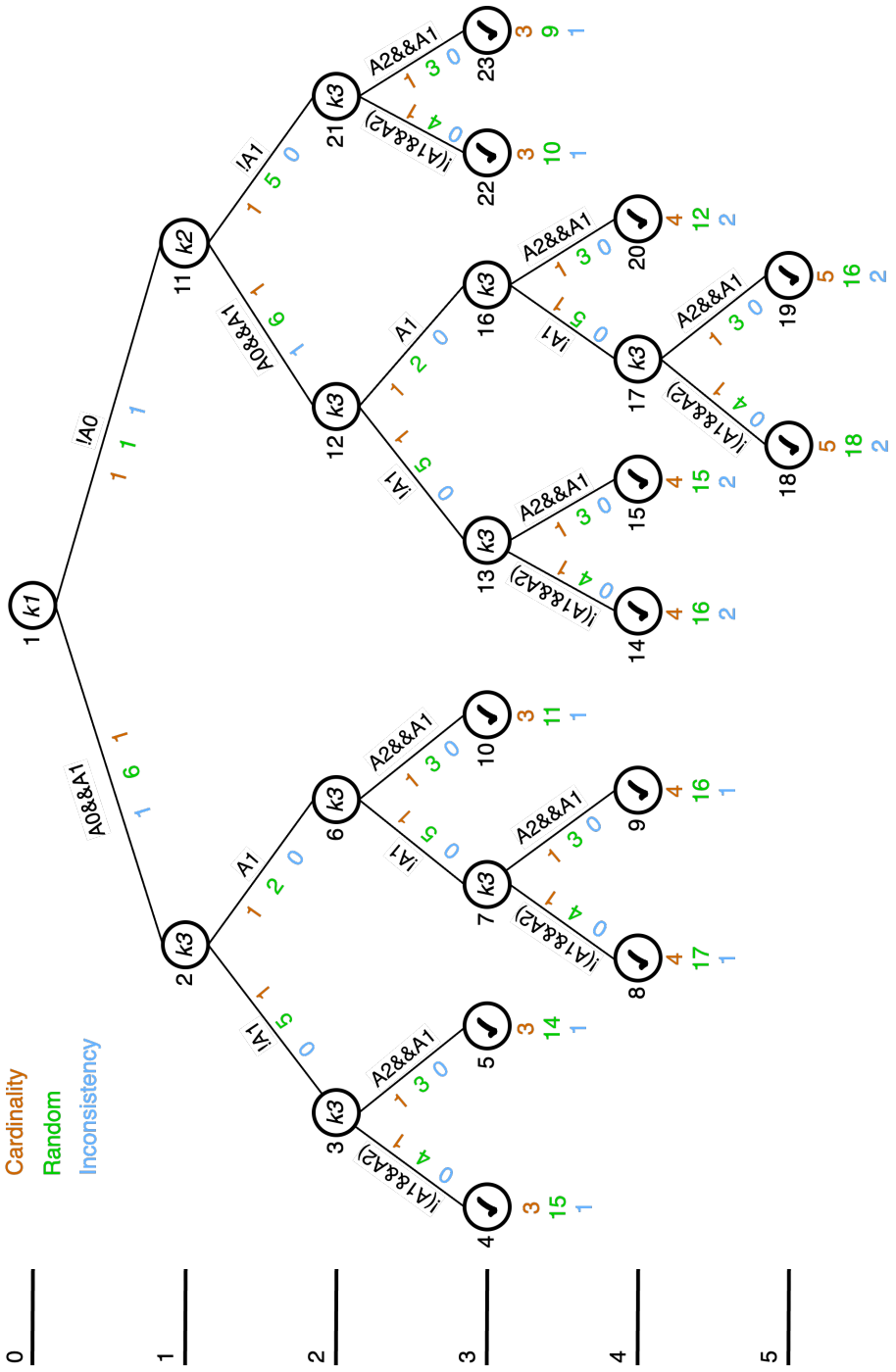


Figure 31: Exemplary HST with edge elements and all three assigned weights