
UNIVERSITY OF CAPE TOWN


DEPARTMENT OF COMPUTER SCIENCE

CS Honours Project Final Paper 2024

Title: Investigating the Optimisation of Rational Closure in Defeasible Reasoning

Author: Maqhobosheane Mohlerepe

Project Abbreviation: EXTRC

Supervisor(s): Tommie Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	15
Experiment Design and Execution	*0*	*20*	10
System Development and Implementation	*0*	*20*	5
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Investigating the Optimisation of Rational Closure in Defeasible Reasoning

Maqhobosheane Mohlerepe
University of Cape Town
Cape Town, South Africa
mhlmaq001@myuct.ac.za

ABSTRACT

Knowledge representation and reasoning is a subfield in artificial intelligence(AI) that aims to formalize the logic of decision-making. Classical reasoning, such as deduction, is a type of reasoning in which conclusions logically follow from premises. Defeasible reasoning offers a way to navigate the uncertainties and exceptions inherent in real-world decision-making. Kraus, Lehmann and Magidor formalized a defeasible reasoning framework known as rational closure, which provides a means to develop algorithms that can accommodate exceptions and uncertainties, bringing us closer to creating machines with reasoning capabilities akin to human cognition. Despite the framework's significance, there has been limited research on its scalability. This paper presents an empirical study exploring how caching and search algorithms can optimize rational closure and evaluates the scalability of the proposed optimized reasoners. Through systematic experiments, our analysis reveals that the degree of optimization in execution times is significantly influenced by factors such as the distribution of statements in the ranked knowledge base, the type of query set, and the search algorithm employed in rational closure. These findings contribute valuable insights into the practical application of rational closure in scalable AI systems.

KEYWORDS

Artificial Intelligence, Knowledge Representation and Reasoning, Defeasible Reasoning, Propositional Logic, Rational Closure, KLM framework

1 INTRODUCTION

The question of how human knowledge can be effectively represented in computer language is a central challenge in artificial intelligence (AI). Classical reasoning, characterized by deductive processes, provides a structured framework for drawing conclusions from premises. However, strict adherence to deductive validity can overlook exceptions and uncertainties in real-world scenarios. For example, classical reasoning would conclude that "penguins can fly" based on the premises "all birds fly" and "penguins are birds." This conclusion, while logically valid, is factually incorrect because penguins are an exceptional type of bird that does not fly. Defeasible reasoning offers a solution by allowing for rational but not necessarily deductively valid inferences. With defeasible reasoning, one might say "birds typically fly," thus leaving room for exceptions like penguins and allowing for the retraction of previous conclusions when new information is presented.This shift from classical logic is crucial for modeling human-like cognition in AI systems, where decision-making often involves navigating complex, uncertain environments. One of the most prominent frameworks for defeasible reasoning is rational closure, as defined by Kraus, Lehmann, and Magidor. While there has been extensive theoretical research on rational closure [2] and various attempts to extend it, fewer efforts have been made to implement defeasible reasoners, particularly in a scalable way. This gap is significant because the effectiveness of automated reasoning in AI, especially when dealing with larger, more complex knowledge bases, hinges on scalability.

This paper aims to optimize the execution time of the rational closure algorithm, thereby improving its scalability through caching techniques. Futhermore, we intend to perform experiments on larger knowledge bases to determine the degree to which binary and ternary search can be used to improve the scalabilty of rational closure. In this paper, all implementations focus on the KLM framework as algorithms based on this framework have been proven to be computable and efficient[13].

This research forms part of a collaborative project alongside the work of Sadiki and Moloi. Sadiki's paper focuses on the generation of defeasible knowledge bases, evaluating parameters used in their creation through Monte Carlo simulations [22]. Moloi's contribution explores both the theoretical foundations and practical implementations of rational closure and lexicographic closure [17].

The rest of this dissertation is structured as follows: Section 2, which is done jointly with Moloi and Sadiki [16], presents the notations, symbols, and technical background. Section 3 details the development of the algorithms, followed by Section 4, which outlines the test cases and evaluation process. Section 5 presents and analyzes the results obtained from evaluation, and Section 6 offers conclusions and recommendations for future research.

2 THEORETICAL BACKGROUND

2.1 Propositional Logic

Propositional logic is a language that formalizes knowledge representation and reasoning. It deals with indivisible statements, known as atomic propositions [10], which can either be true or false. This language utilizes atomic propositions and logical operators ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) to construct propositional formulas[3]. A set of these formulas is formally denoted by $\mathcal{L}$. Valuations are all the logically possible states of the world based on a set of propositional atoms $\mathcal{P}$ in some language $\mathcal{L}$[11]. Formally, a valuation is a function ($u\colon \mathcal{P} \rightarrow \{\mathrm{T}, \mathrm{F}\}$) where T and F are True and False,

respectively. A valuation (u) is said to satisfy a formula (α) in the language ($\mathcal{L}$), formally denoted as (u) $\Vdash \alpha$), if the truth value of (α) under the valuation (u) is true [15, 18, 21].

Any valuation u that satisfies α is a model of α, denoted as Mod(α). A knowledge base (denoted by $\mathcal{K}$) is a finite set of propositional formulas from which conclusions can be drawn. The set of valuations that satisfy all the formulas in a knowledge base $\mathcal{K}$ are the models of $\mathcal{K}$, denoted as Mod($\mathcal{K}$) [15, 18, 21].

Entailment is the notion that using the relationship between two or more statements one can make deductions or conclusions . This is a fundamental concept in reasoning, it formally shows how certain states of the world are logical consequence of others [24]. Suppose we are given propositional statements or formulas, $\alpha, \beta \in \mathcal{L}$, α is said to entail β or β referred to as a logical consequence of α, denoted $\alpha \models \beta$ if and only if the models of α are a subset of the models of β ($Mod(\alpha) \subseteq Mod(\beta)$). The set of models of α , denoted $Mod(\alpha)$, are the set of all valuations that satisfy α [11, 15, 18, 21].

2.2 Defeasible Reasoning

Defeasible reasoning within the KLM-style framework introduces a syntax and semantics that enriches classical propositional logic, allowing for the representation of typicality and exceptionality in logical inference. At the syntactic level, the framework incorporates defeasible implications (DIs), denoted as $\alpha \vdash \beta$, where α typically implies β[5].These defeasible implications do not allow for nesting, maintaining a level of simplicity in their application.

The KLM framework is semantically grounded in preferential interpretations [12, 23], where interpretations are ranked based on their degree of 'typicality.' These interpretations form a hierarchical structure, with the lowest ranks representing the most typical scenarios and the highest ranks representing exceptions or less typical situations. The formalization of this ranking system allows defeasible reasoning to model the subtleties of human cognition, where exceptions to general rules are common.

2.3 KLM: Defeasible Entailment

Defeasible reasoning differs from classical reasoning in that conclusions can change with the introduction of new information, making it non-monotonic. This flexibility allows defeasible reasoning to more accurately reflect real-world scenarios where exceptions to general rules occur [1, 11, 15, 18, 21]. For example, in a knowledge base knowledge base $\mathcal{K}$ with the statements: $\{b \rightarrow f, b \rightarrow w, p \rightarrow b, p \rightarrow \neg f\}$ where b, f, w and p represent birds, fly, wings and penguins, respectively. Classical reasoning would incorrectly conclude that penguins, being birds that cannot fly, should not exist. However, defeasible reasoning, by recognizing the exception that penguins do not fly, prevents this contradiction [15, 18, 21].

To handle such exceptions, the KLM framework introduces the concept of typicality (denoted as $\vdash$), replacing material implications in statements subject to exceptions. In the given example, $b \rightarrow f$ becomes $b \vdash f$, reflecting that while birds typically fly, not all birds do (e.g., penguins). Conclusions drawn from such a knowledge base use defeasible entailment (denoted by $\approx$) instead of classical entailment, ensuring that inferences remain consistent with typicality[11, 24] .

The framework also adheres to LM-rationality, a set of properties proposed by KLM to guide defeasible reasoning. These properties include Reflexivity, Left Logical Equivalence, Right Weakening, And, Or, Cautious Monotonicity, and Rational Monotonicity. All of which ensure that defeasible reasoning operates logically and consistently in the presence of exceptions[4, 8, 11, 12, 14, 15, 18, 21].

Preferential interpretations, as articulated by KLM [12] and Lehmann and Magidor [14], are based on the preferential semantics initially formulated by Shoham [11, 12]. Defeasible reasoning is semantically supported by ranked interpretations, which organize possible scenarios into a hierarchy of typicality.A ranked interpretation is formally defined as a function, denoted as $R : \mathcal{U} \rightarrow \mathbb{N} \cup \infty$, where $\mathcal{U}$ represents the set of interpretations. The most typical interpretations are assigned lower ranks (closer to 0), and the least typical (or impossible) are assigned higher ranks (up to ∞) [5, 15, 18, 21]. A ranked interpretation, R is said to satisfy a defeasible implication $\alpha \vdash \beta$ if, in the lowest rank where α is satisfied, β holds in all corresponding models within that rank[5]. In contrast, classical propositional logic considers a formula true across all interpretations of a given rank, disregarding typicality [15, 18, 21].

2.4 Rational Closure

Rational closure in defeasible reasoning is a crucial process for deriving conclusions from both explicit and implicit statements within a knowledge base. It allows us to conduct rational inference on defeasible knowledge bases [6], ensuring that they are logically sound and consistent. One key aspect of rational closure is to assign ranking to each formula in the knowledge base, thereby extending preferential reasoning [15, 18, 21, 24].

Formally, rational closure involves determining whether defeasible implication is entailed by a knowledge base using a method that adheres to the rationality properties proposed by Lehmann and Magidor. These properties ensure that conclusions drawn from defeasible implications remain consistent and coherent, even when exceptions are present. Rational closure is the most conservative form of defeasible entailment [15, 18, 21]. This means that anything entailed by rational closure will also be entailed by the other common forms of defeasible entailment [19]. The process of rational closure can be defined either semantically or algorithmically, as outlined in [5, 15, 18, 21].

2.4.1 Minimal Ranked Entailment. The semantic foundation of rational closure is built on ranked interpretations, which organize possible worlds (or scenarios) based on their typicality. In this context, the most typical worlds are ranked lower, while exceptions are ranked higher. Among the various ranked interpretations that can be generated from a knowledge base $\mathcal{K}$, there exists a unique minimal ranked interpretation, denoted as $\mathcal{R}^{\mathcal{K}}_{RC}$. If this minimal ranked interpretation entails a defeasible implication $\alpha \vdash \beta$, it is expressed as $\mathcal{K} \approx \alpha \vdash \beta$, indicating that $\alpha \vdash \beta$ is part of the rational closure of $\mathcal{K}$. This form of entailment, derived from ranked interpretations, is known as LM-rational, reflecting its foundation in the work of Lehmann and Magidor [15, 18, 21].

The concept of minimal ranked entailment is directly related to the Base Rank Algorithm. The minimal ranked interpretation $\mathcal{R}^{\mathcal{K}}_{RC}$ that rational closure seeks to approximate through systematic ranking and filtering of contradictions is the goal of the Base Rank Algorithm.

2.4.2 Base Rank Algorithm. The Base Rank Algorithm is the first step in determining whether a defeasible implication is part of the rational closure of a knowledge base $\mathcal{K}$. This algorithm systematically processes the knowledge base to create ranks that reflect the typicality of statements, ultimately leading to the identification of defeasible entailments.

As detailed in [5], the base rank is the first of the two sub-algorithms used for rational closure.

The steps for Base Rank Algorithm are as follows:

(1) Materialization: The knowledge base $\mathcal{K}$ is transformed into its material counterpart $\overrightarrow{\mathcal{K}}$, where defeasible implications are replaced with material implications (i.e., $\vdash$ is replaced with $\rightarrow$). This is denoted as $\mathcal{E}_0$.

(2) Iterative Filtering: A series of materializations $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n, \mathcal{E}_\infty$ are generated. The initial materialization $\mathcal{E}_0$ corresponds to the materialized knowledge base $\overrightarrow{\mathcal{K}}$. Subsequent materializations $\mathcal{E}_i$ are derived by eliminating statements $\alpha \rightarrow \beta$ from the previous materialization $\mathcal{E}_i - 1$ where α can be shown to be false. This process continues until no further eliminations are possible, resulting in the final materialization $\mathcal{E}_\infty$.

(3) Ranking: The ranking is established based on the differences between consecutive materializations. Statements in the original knowledge base $\mathcal{K}$ occupy the bottom rank (infinite rank), while more general statements appear in higher ranks. A statement $\alpha \rightarrow \beta$ is assigned a base rank i if it is present in rank i, indicating that it holds in at least one typical interpretation corresponding to that rank.

Algorithm 1 BaseRank

Input: A knowledge base $\mathcal{K}$
Output: An ordered tuple $(R_0, \ldots, R_{n-1}, R_\infty, n)$
$i := 0$
$E_0 := \overrightarrow{\mathcal{K}}$
repeat
  $E_{i+1} := \{\alpha \rightarrow \beta \in E_i \mid E_i \models \neg\alpha\}$
  $R_i := E_i \setminus E_{i+1}$
  $i := i + 1$
until $E_{i-1} = E_i$
$R_\infty := E_{i-1}$
if $E_{i-1} = \emptyset$ **then**
  $n := i - 1$
else
  $n := i$
end if
return $(R_0, \ldots, R_{n-1}, R_\infty, n)$

2.4.3 Rational Closure Algorithm. After establishing the base ranks using the Base Rank Algorithm, the RationalClosure algorithm checks whether a defeasible implication $\alpha \vdash \beta$ is entailed by the knowledge base $\mathcal{K}$. This is achieved by systematically removing ranks that cause contradictions and determining whether the remaining ranks support the implication.

The steps for the rational closure algorithm are as follows:

(1) Initialize: The algorithm starts by retrieving the ranks generated by the BaseRank algorithm. The initial combined rank set $\mathcal{R}$ includes all ranks from R_0 to $R_n - 1$.

(2) Contradiction Check: The algorithm iteratively checks if the union of the infinite rank R_∞ and the combined ranks R entail the negation of the antecedent $\neg\alpha$. If a contradiction is found (i.e., $(R_\infty \cup R \models \neg\alpha)$, the most typical rank (starting from R_0) is removed, and the process continues until no more contradictions exist or all ranks have been removed.

(3) Final Entailment Check: Once the contradictory ranks are removed, the algorithm checks whether the remaining ranks (including R_∞) entail $\alpha \rightarrow \beta$. If they do, the algorithm returns true, indicating that $\mathcal{K}$ defeasibly entails $\alpha \vdash \beta$; otherwise, it returns false.

Algorithm 2 RationalClosure

Input: A knowledge base $\mathcal{K}$ and a Defeasible Implication $\alpha \vdash \beta$
Output: true, if $\mathcal{K} \mid\approx \alpha \vdash \beta$, and **false**, otherwise
$(R_0, ..., R_{n-1}, R_\infty, n)$:= BaseRank($\mathcal{K}$)
$i := 0$
$R := \bigcup_{i=0}^{j<n} R_j$
while $R_\infty \cup R \models \neg\alpha$ and $R \neq \emptyset$ **do**
  $R := R \setminus R_i$
  $i := i + 1$
end while
return $R_\infty \cup R \models \alpha \rightarrow \beta$

The following example illustrates rational closure.

Suppose we want to know whether a knowledge base $\mathcal{K}$ with the statements penguins are birds($p \rightarrow b$), birds typically fly($b \vdash f$), penguins typically do not fly($p \vdash \neg f$) and penguins typically have wings($p \vdash w$) entails penguins typically fly($p \vdash f$).

Following the base rank algorithm, the materialised knowledge base $\overrightarrow{\mathcal{K}}$ will be ranked as follows:

Rank Number	Formulas
0	$b \rightarrow f$
1	$(p \rightarrow \neg f)$, $(p \rightarrow w)$
∞	$p \rightarrow b$

Table 1: Initial Ranks

The rational closure algorithm is then used to check where the ranked knowledge base entails materialised "penguins typically fly"(i.e., $p \rightarrow f$).

Starting from R_0, we check for contradictions by checking whether $R_\infty \cup R \models \neg p$. We find that this is the case and remove the lowest rank R_0 and are left with the ranks in Table 2.

Rank Number	Formulas
0	$(p \rightarrow \neg f)$, $(p \rightarrow w)$
∞	$p \rightarrow b$

Table 2: Remaining Ranks

We check for contradictions again and find that there are none. We now proceed to the final entailment check in which we verify whether the remaining ranks entail $p \rightarrow f$. They do not, thus the algorithm would output **False**.

3 SYSTEM DEVELOPMENT AND IMPLEMENTATION

In this paper, we developed a total of six defeasible reasoners. Each reasoner determines whether a defeasible implication statement (i.e., a statement of the form $\alpha \vdash \beta$) is entailed by a specific defeasible knowledge base. All implementations described in this section produced correct results when tested against cases corresponding to the KLM Properties, as detailed by Kraus, Lehmann, and Magidor [14]. Consequently, we are confident in the completeness and correctness of our reasoners. These reasoners are built upon the foundational work of Pillay and Hamilton[7, 20], who extensively utilized the TweetyProject library and the built-in classical reasoner tool, which employs the satisfiability(SAT) Sat4j solver. In our implementation, we employ a SAT solver—a tool that checks the satisfiability of Boolean formulas—to determine whether certain ranks in the knowledge base can be removed without causing inconsistencies. This step is crucial in the process of entailment checking.

The first reasoner developed implements rational closure directly, following the approach detailed in [5] as described in algorithm 2. This implementation is referred to as the naive implementation, as it represents the most basic version of rational closure.

3.1 Optimisation Technique 1: Binary and Ternary Entailment Checkers

A second reasoner was developed utilizing binary search to identify the ranks that should be removed. Unlike the naive approach, which performs a linear search through the ranked knowledge base, the binary search method recursively divides the ranked knowledge base into two parts. This process continues until it identifies the rank at which the formula α becomes consistent with the knowledge base, given a defeasible query where $\alpha \vdash \beta$. The algorithm that was used was adopted from [7] and the steps below outline how it is done:

(1) **Initialization:** Begin by setting the min and max values, representing the smallest and largest possible ranks within the ranked knowledge base. If this is the first execution of the algorithm, initialize min as 0 and max as the total number of ranks.

(2) **Midpoint Calculation:** Calculate the midpoint between min and max. This midpoint will be the current rank under consideration, denoted as r.

(3) **Consistency Check - Upper Half:** Evaluate whether removing rank r and all ranks above it results in the antecedent, α becoming consistent with the knowledge base. If it does, proceed to the next step. If it does not, update min to $r + 1$ and repeat the process, as the rank at which α becomes consistent must be in the lower half (i.e., higher rank numbers).

(4) **Consistency Check - Lower Half:** Check if adding rank r back into the knowledge base makes α consistent. If so, update max to r and repeat the process. This indicates that the rank where α becomes consistent lies in the upper half (i.e., lower rank numbers).

(5) **Final Decision:** If neither of the above conditions holds, rank r is identified as the highest rank that needs to be removed. The algorithm will then verify if $\alpha \vdash \beta$ by checking the consistency of the formula with the knowledge base, excluding all ranks up to and including r.

(6) **Return Result:** The algorithm returns true if $\alpha \vdash \beta$ is entailed by the knowledge base after removing the necessary ranks, and false otherwise.

Previous research has indicated that the optimization's effectiveness is influenced not by the number of ranks but by the rank at which each query becomes consistent. We hypothesize that since binary search operates with a time complexity of $O(logn)$ for both its best and worst cases, where n is the number of ranks, improved performance will only be observed if α becomes consistent with the knowledge base at a rank approximately greater than $logn$. The naive reasoner, which iterates linearly, has a time complexity of $O(1)$ in its best case and $O(n)$ in its worst case. Therefore, for the binary search to show improvement, α would need to become consistent at a rank larger than $logn$. Consequently, we anticipate that the results of this experiment will align with previous findings, even with the larger knowledge bases we employed.

The final reasoner developed to enhance the process of entailment checking is the ternary entailment checker. This method, like the binary search approach, divides the ranked knowledge base, but it does so into three sections instead of two. The algorithm adopted from Pillay[20] has been observed to perform significantly better than the binary entailment checker, particularly when the query antecedent becomes consistent with the knowledge base at higher ranks.

We hypothesize that, given our large knowledge bases, our results will align with these findings. The ternary division of the knowledge base allows for potentially fewer searches when the rank at which the query antecedent becomes consistent is high. To investigate the significance of this difference, we plan to use the Wilcoxon signed-rank test to compare the runtimes, rather than simply comparing the actual runtimes. It is important to note, as observed in previous investigations into the optimization of rational closure, that while the number of entailment checks has been optimized, entailment

checking still reduces to the NP-complete Boolean satisfiability problem. Nonetheless, these attempts are expected to yield some improvement in execution times.

The algorithm that was used for implementation was adopted from Pillay and works as follows:

(1) **Initialization:** The algorithm starts by setting a *min* value to 0 and a *max* value to the total number of ranks. This defines the initial search range for the ranks.

(2) **Midpoint Calculation:** Two midpoints, $mid1$ and $mid2$, are calculated based on the current *min* and *max* values:

$$mid1 = \left\lfloor min + \frac{max - min}{3} \right\rfloor$$

$$mid2 = \left\lfloor max - \frac{max - min}{3} \right\rfloor$$

These midpoints divide the search space into thirds for more efficient narrowing of the search range.

(3) **First Consistency Check:** The algorithm checks if removing all ranks above and including $mid1$ makes the negation of the formula consistent with the knowledge base.

- If this condition holds, it means we have removed enough low-priority information to ensure consistency, and the algorithm moves to step 4.
- If not, the algorithm proceeds to step 5.

(4) **Setting "rankRemove":** If removing ranks above and including $mid1$ leads to consistency, the next step is to check whether adding $mid1$ back keeps the negation consistent.

- If adding $mid1$ back does not break the consistency, this means that $rankRemove$ is set to $mid1$, as it is the highest rank that needs to be removed for consistency. The algorithm then proceeds to step 9 to finalize the decision.
- If adding $mid1$ back results in inconsistency, set *max* to $mid1$ and return to step 2 to search in the upper half of the ranks.

(5) **Second Consistency Check (mid2):** The algorithm checks whether $mid2$ is less than the total number of ranks.

- If $mid2$ is within the range of ranks, proceed to step 6.
- If $mid2$ equals the total number of ranks, go to step 7.

(6) **Check Removing Rank - mid2:** The algorithm checks if removing all ranks above and including $mid2$ makes the negation of the formula consistent with the knowledge base.

- If the negation is consistent, proceed to step 8.
- If not, update *min* to $mid2 + 1$ and return to step 2 to search in the lower half of the ranks.

(7) **Boundary Case Check:** If $mid2$ equals the total number of ranks (i.e., $mid2 == n$), the algorithm updates the search range by setting *min* to $mid1 + 1$ and *max* to $mid2$. Then, return to step 2 to continue searching in the upper half of the ranks.

(8) **Check Adding Rank Back - mid2:** The algorithm checks if adding $mid2$ back into the knowledge base (i.e., including all ranks from 'mid2' to $n - 1$) maintains consistency with the negation of the formula.

- If it does, $rankRemove$ is set to $mid2$, and the algorithm moves to the final step (step 9).
- If adding $mid2$ makes the negation inconsistent, the algorithm adjusts the search range by setting *min* to $mid1 + 1$ and *max* to $mid2$, and then returns to step 2.

(9) **Final Decision:** Once the appropriate $rankRemove$ has been identified, the algorithm checks whether the knowledge base, with all ranks from $rankRemove$ and above removed, entails the formula α.

- If the formula is entailed, the algorithm returns **True**.
- Otherwise, it returns **False**.

3.2 Optimisation Technique 2: Caching

We developed a set of three reasoners that implement caching mechanisms, each utilizing different strategies: naive, binary, and ternary entailment checking algorithms. To optimize the process, two distinct caching mechanisms were implemented using hash tables.

3.2.1 Caching Mechanism 1: Antecedent-Based Caching. The first caching mechanism focuses on storing the processed antecedents of queries. If multiple queries within a set share the same antecedent, the system can skip the repeated computation of ranks to be removed for that specific knowledge base. Instead, it can directly proceed to the final entailment check, saving significant computational time.

Suppose the system encounters two queries, $\alpha_1 \sim \beta_1$ and $\alpha_1 \sim \beta_2$. Once the ranks associated with α_1 are processed during the first query, the same ranks are cached. When processing the second query, the system retrieves the cached ranks, bypassing the need to recompute them.

This mechanism can be highly beneficial in scenarios where a large number of queries share common antecedents. By avoiding redundant computations, it speeds up the overall entailment checking process.

3.2.2 Caching Mechanism 2: Full Query and Result Caching. The second caching mechanism stores entire queries along with their results. If a query appears multiple times within a query set, the system does not need to recompute the entailment or the ranks to be removed. Instead, it directly returns the cached result. Consider a query set where $\alpha_1 \sim \beta_1$ appears multiple times. After the first evaluation, the result is cached. Subsequent occurrences of the same query can instantly return the cached result, eliminating the need for any further computation.

While these caching mechanisms are memory-intensive, especially with large knowledge bases, we predict that their implementation

is justified by the significant reduction in computational overhead. The hash tables used for caching can grow large, but this is a necessary trade-off to achieve scalability in large-scale systems.

We predicted that for the naive reasoner, there would be significant optimizations in scenarios with high redundancy in antecedents due to the linear nature of its search process, which benefits greatly from antecedent caching. Futhermore, the binary reasoner would yield moderate to high optimizations across various scenarios, with the best performance seen when queries are spread across ranks rather than clustered at a single rank. Lastly, the ternary reasoner would yield the highest level of optimization, particularly in large knowledge bases with high consistency points and with the support of caching mechanisms that reduce the number of search steps.

4 EXPERIMENT DESIGN

4.1 Test Cases

4.1.1 Knowledge Bases. In our experiments, we utilized knowledge bases generated by Bailey [7] and Sadiki[22], specifically designed to be queried in the context of our research. These knowledge bases were composed of defeasible statements only since our primary focus will be how the ranking interacts with optimisation techniques. To ensure consistency with previous studies, the ranked knowledge bases were varied in terms of the number of ranks and the distribution of statements in the ranks. Additionally, we introduced a new variable of the total number of statements in a knowledge base. In previous studies, only the number of ranks was primarily used to represent the size of the knowledge base. We believe that incorporating this new variable as a measure of size will enable a more comprehensive investigation of the scalability offered by the proposed optimization techniques.

The variations in the knowledge bases were systematically controlled as follows:

(1) Number of Ranks: We generated knowledge bases with 10, 50, and 100 ranks. In all cases, these ranks were populated with an equal number of statements per rank, thereby achieving a uniform distribution of statements across the ranks.Each knowledge base contained a total of 1,000 statements.

(2) Statement Distribution: The generated statements within the knowledge bases followed different distribution patterns,uniform, linear decrease, and linear decline,while maintaining the same number of 10 ranks and 1,000 statements. This allowed us to investigate the impact of statement distribution on the performance of our entailment checking algorithms.

(3) Number of Statements: We generated knowledge bases with 500, 1000 and 2000 statements. In all cases, the knowledge bases had 50 ranks and a uniform distribution(same number of statements in each rank after base rank algorithm).

By employing these variations, we aimed to thoroughly assess the scalability and efficiency of the algorithms under conditions that closely resemble the diversity found in practical applications of knowledge bases.

4.1.2 Query Sets. To evaluate the various factors hypothesized to influence the extent of performance improvement, we designed and utilized a series of manually curated query sets consisting of 100 queries each. These query sets were crafted to test specific aspects of the caching mechanisms and search algorithms under investigation:

(1) **Unique Query Formulas:** This set consisted of entirely distinct query formulas, ensuring no overlap or repetition among the queries.

(2) **Mixed Unique and Repeated Queries:** This set included an equal split between unique queries and repeated queries, allowing us to observe the impact of query repetition on performance.

(3) **Queries with Repeated Antecedents:** This set was composed of queries that shared the same antecedent but differed in their consequents, providing insights into how the caching mechanism handles repeated antecedents.

(4) **Mixed Unique and Repeated Antecedent Queries:** In this set, half of the queries were unique, while the other half had repeated antecedents, enabling a comparison of performance across different types of query repetition.

(5) **Half Repeated Antecedents and Half Repeated Queries:** This query set included a mix of repeated antecedents and repeated queries, designed to test the combined effects of both types of repetition on the caching mechanisms.

(6) **Queries Consistent at Final Rank:** This set contained queries that only became consistent at the final rank, aimed at evaluating the effectiveness of the search algorithms when the solution lies at the extreme end of the rank spectrum.

(7) **Queries Consistent after Removal of Rank 0:** This set consisted of queries that became consistent immediately after the removal of the first rank, testing the search algorithms' efficiency in scenarios where the solution is found early in the process.

Query sets 1 to 5 were specifically developed to assess the efficacy of the caching mechanisms, while query sets 6 and 7 were designed to evaluate the performance of the search algorithms under different consistency conditions.

4.2 Execution

A program was developed to evaluate the performance of six different reasoners when applied to a given knowledge base and query set. All reasoners utilized the same base rank algorithm hence all timing was done on the entailment checking process only.

For each query set, the program executed the entire set of queries against the knowledge base using each reasoner, recording the total execution time required for the reasoner to determine whether the knowledge base entails each query in the set. This process was repeated five times for each reasoner to obtain a reliable measure of performance, with the average execution time across these runs calculated and recorded.

The recorded average execution times for each reasoner were then stored in a CSV file for further analysis. The experiments were conducted on a laptop equipped with 8 GB of RAM, an Intel Core i7 processor with 4 cores and 8 logical processors, providing a consistent hardware environment for all tests.

5 RESULTS AND DISCUSSION

For each of the knowledge bases and query sets, a Wilcoxon signed-rank test was conducted to compare the runtimes of the non-cached implementations of the reasoners, specifically comparing the naïve with the binary, naïve with the ternary, and ternary with the binary implementations. Additionally, comparisons were made between the runtimes of the cached implementations and their respective non-cached counterparts. The obtained p-values were evaluated, where a p-value greater than or equal to 0.05 indicates support for the null hypothesis, suggesting no statistically significant difference between the mean runtimes of the two reasoners being compared. Conversely, a p-value less than 0.05 supports the alternative hypothesis, indicating a significant difference in runtimes between the reasoners. The p-values were computed using an online Wilcoxon signed-rank test calculator[9], and the complete set of p-values is provided in the appendix.

5.1 Binary and Ternary Search Results

5.1.1 Comparison over varying number of statements. The experimental results reveal a distinct and consistent difference in runtimes between the binary and ternary search entailment methods when compared to the naive entailment method. This difference remains statistically significant, with a p-value of 0.01563, as the number of statements increases, suggesting that the optimization achieved by the binary and ternary search methods is not affected by the total number of statements in the knowledge base. Both the binary and ternary search entailment methods consistently outperformed the naive entailment across various query set and knowledge base combinations. This superior performance can be attributed to the binary and ternary search methods' ability to effectively narrow down the search space, as opposed to the linear processing of all ranks performed by the naive entailment method. By efficiently identifying the relevant rank where the antecedent becomes consistent, the binary and ternary search methods maintain their advantage regardless of the overall size of the knowledge base. Moreover, the results provide clear evidence of a lack of significant difference between the ternary and binary entailment checks. This lack of difference remains relatively consistent as the number of statements increases, with an average p-value of 0.1980. Given that the search space is divided by ranks and the statements are uniformly distributed, the number of statements does not appear to influence this observed lack of difference between the binary and ternary methods.

5.1.2 Comparison over varying number of ranks. The query sets evaluated to test the efficacy of incorporating binary and ternary search algorithms into the rational closure entailment checking were those that become consistent at the first rank and those that become consistent at the last rank. These specific query sets will serve as the primary focus in the discussion of the results.

A distinct difference in runtimes was observed between the naive implementation and the binary and ternary implementations, as indicated by a p-value of 0.01563. This difference remained consistent as the number of ranks increased. The experimental results demonstrate that the performance of the binary and ternary search reasoners surpassed that of the naive reasoner, even when the antecedent became consistent with the knowledge base at the first rank. This observation contradicts the initial hypothesis, which asserted that the binary and ternary reasoners would only exhibit improved performance when the antecedent becomes consistent at a rank significantly greater than $\log n$, where n represents the total number of ranks.

Given that $\log n \approx 1.7$ for $n = 50$, it is plausible that the proximity of the consistent rank to $\log 50$ contributed to the observed performance gains of the binary and ternary search reasoners. While the hypothesis predicted that the naive reasoner would be more efficient in scenarios where the consistent rank is low, the experimental results suggest that the binary and ternary search methods can deliver superior performance even when the consistent rank is close to $\log n$.

This outcome implies that the efficiency of the binary and ternary search algorithms may not be as tightly correlated with the rank of consistency as initially hypothesized. Instead, the results suggest that the advantages of the binary and ternary search approaches could extend to a broader range of scenarios, including those where the consistent rank is relatively low. Further investigations should explore how close to $\log n$ the consistent rank must be for the binary and ternary implementations to start performing better, as this may provide a more comprehensive understanding of the conditions under which these search methods offer significant advantages.

There is a large distinct difference between the ternary and binary implementations at 10 ranks with a p-value of 0.01563. However, this changes to a non-significant difference with a p-value of 0.1563 for ranks 50 and 100. At rank 10, the binary entailment is seen to drastically outperform the ternary entailment across all query sets. This may be due to the fact that binary entailment only has to deal with two sections at a time thus for the knowledge bases with fewer ranks, there is far less overhead. Since these sections are not dealt with concurrently, then it means that the ternary dividing is small knowledge base(one with fewer ranks) and processing 3 mid sections one after the other depending on the type of query might take up more time as opposed to just the two sections. Even where the difference is not large enough to reject the null hypothesis that there is a difference in runtimes, wherever there is a difference, the binary entailment is seen to outperform the ternary still and this may be attributed to the previous reason. It would be beneficial for more research to be done on when the ternary starts performing almost as well as the ternary or better therefore an exploration into investigating more ranks and fewer ranks. The current experiment limits the conclusions that can be drawn.

The data shows that there is a large non-significant difference between the two reasoners when given query set which become consistent in the first or last ranks with p-values of 0.5781 and 0.4688 respectively implying that the rank at which the antecedent became

consistent with the knowledge base does not impact degree of optimisation. This could be because both algorithms are of $O(logn)$.

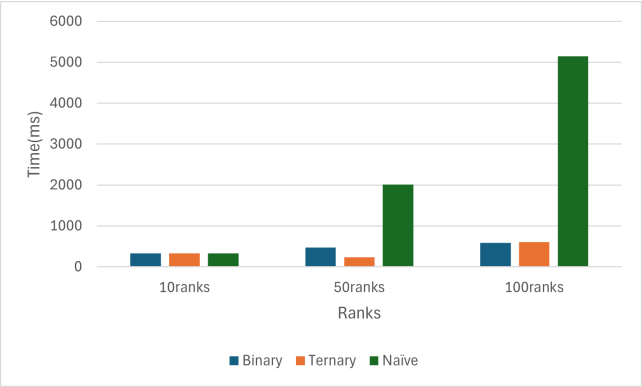

Figure 2: Runtime comparison between Binary, Ternary, and Naïve entailment across different ranks.

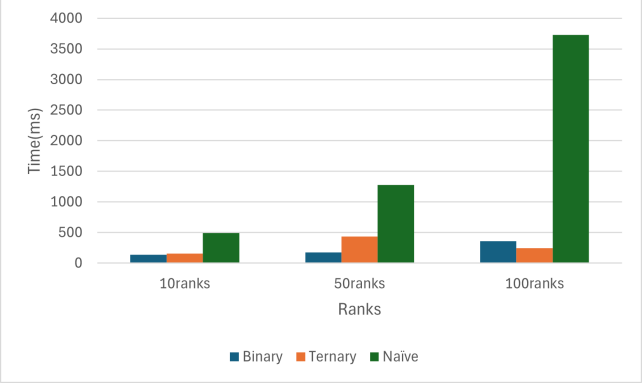

Figure 3: Runtime comparison between Binary, Ternary, and Naïve entailment across different ranks.

5.1.3 Comparison over varying statement distribution. Under a uniform distribution, where the statements are evenly distributed across ranks, all comparisons—binary vs naive, binary vs ternary, and ternary vs naive—showed significant differences, with p-values of 0.01563. This indicates that both binary and ternary search algorithms provide substantial performance improvements over the naive method in uniformly distributed scenarios. The consistent significance across all comparisons suggests that the ability to quickly narrow down the search space confers a decisive advantage to these optimized methods. In particular, the ternary method, which divides the search space into three sections, also outperforms the naive method significantly, indicating that even with its additional complexity, it remains efficient under uniform distribution conditions.

In contrast, the results for normal and exponential distributions present a more varied results. For the normal distribution, the binary vs naive comparison remains statistically significant, with a p-value of 0.03125, indicating that the binary method still outperforms the naive approach. However, the comparisons between binary and ternary ($p = 0.375$) and between ternary and naive ($p = 0.2969$) do not show significant differences. This suggests that while the binary method continues to offer performance gains over the naive method, the ternary method does not provide a meaningful improvement over either the binary or naive methods in this distribution. This could be due to the characteristics of the normal distribution, where ranks are more densely populated around the mean, potentially reducing the effectiveness of additional subdivisions in the search space.

Similarly, under an exponential distribution, the binary vs naive comparison is significant with a p-value of 0.01563, demonstrating that the binary method maintains its advantage in scenarios where the number of statements drop increase sharply after the initial ranks. However, the binary vs ternary ($p = 0.07813$) and ternary vs naive ($p = 0.1563$) comparisons do not reach significance, suggesting that the ternary method does not provide additional optimization benefits in this context. The lack of significant differences between ternary and the other methods may reflect the exponential distribution's nature, where later ranks have more statements, making the ternary method's additional complexity less effective.

The consistent performance advantage of the binary method across different distributions and rank sizes implies that the efficiency of dividing the search space into two sections is well-suited to a wide range of scenarios. The binary method's ability to halve the search space at each step offers a significant optimization over the linear approach of the naive method, particularly in cases where the rank distribution is not uniform. The ternary method, while potentially beneficial in certain scenarios, does not consistently outperform the binary method, especially in distributions where the additional division of the search space may not justify the added complexity.

These findings suggest that while both binary and ternary search methods are valid optimizations of the rational closure algorithm, the binary method generally provides a better balance of efficiency and computational overhead. This makes it the preferred choice for optimizing rational closure in most scenarios, especially when the distribution of statements across ranks is not uniform or when the number of ranks is large. Further research could explore specific conditions under which the ternary search method might outperform the binary approach, particularly in knowledge bases with highly complex or irregular rank distributions.

5.2 Caching Results

Our research findings highlight that caching, while not advantageous for unique query sets, significantly enhances performance for query sets involving repetition. This is in line with the expected behavior of caching, which reduces redundant calculations, particularly for repeated queries or parts of queries.

Like the naive algorithm, the binary algorithm shows no significant difference in performance with caching for the unique query set (p = 0.2188), suggesting that caching does not improve performance for unique queries. However, for repeated queries, the p-value of 0.07813 approaches significance, indicating a potential trend toward

caching being beneficial. In contrast, the binary algorithm shows significant performance improvements for the half-repeated (p = 0.01563), same query (p = 0.01563), and mixed (p = 0.02225) query sets when caching is employed. This underscores the role of caching in reducing computational overhead by avoiding the recalculation of results for similar or repeated queries.

The ternary algorithm exhibits similar behaviour to the naive and binary algorithms, with no significant advantage observed for caching when processing unique queries (p = 0.9375). However, for repeated (p = 0.01563), half-repeated (p = 0.01563), same query (p = 0.01563), and mixed (p = 0.02225) query sets, the p-values indicate that caching significantly improves performance. This is consistent with the nature of these queries benefiting from caching mechanisms, which help avoid redundant computations.

The data demonstrates that caching significantly enhances performance across all three algorithms (naive, binary, and ternary) for query sets involving repetition, such as repeated, half-repeated, same, and mixed queries. This is consistent with the expected behaviour, as caching allows the algorithms to store and reuse results from previously computed queries, thereby reducing the need for redundant computations.

To illustrate the runtime optimization provided by caching, Figure 3 presents the graph of the mixed query set, where half of the query set shares an antecedent, and the other half consists of repeated queries, fully leveraging the hybrid caching mechanism that we implemented.

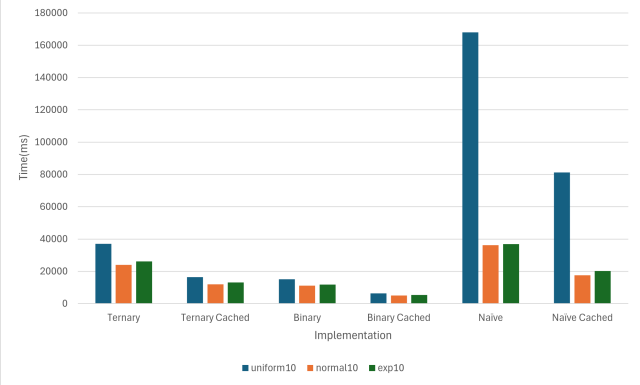

Figure 4: Runtime comparison between all implementations with and without caching on a mixed query set

Figure 3 further confirm that the binary search implementation is able continues to outperform both naive and ternary entailment. For this query set, the binary cached implementation has the lowest runtimes.

6 CONCLUSIONS

This research set out to optimize the execution time of the rational closure algorithm, with a particular focus on enhancing its scalability by implementing caching techniques. By conducting experiments on more extensive and more complex knowledge bases, we aimed to assess the extent to which binary and ternary search methods could be leveraged to improve the scalability of rational closure within the KLM framework.

The experimental results reveal a distinct and consistent difference in runtimes between the binary and ternary search methods compared to the naive implementation. These differences remained statistically significant even as the number of statements increased. This suggests that the optimization achieved by the binary and ternary search methods is robust and not adversely affected by the total number of statements in the knowledge base. The binary and ternary search methods consistently outperformed the naive method across various query sets and knowledge base combinations, primarily due to their ability to effectively narrow down the search space, as opposed to the linear processing of all ranks performed by the naive method.

However, the results also provide clear evidence of a need for significant difference between the binary and ternary methods as the number of statements increased. This suggests that while both methods are efficient, their performance may converge in specific scenarios, mainly when the statements are uniformly distributed across ranks. Additionally, our findings indicate a significant performance gap between the binary and ternary implementations at lower rank sizes, which diminishes as the rank size increases. The superior performance of the binary method at lower ranks could be attributed to the reduced overhead of managing only two sections compared to the ternary method, which divides the search space into three sections. This highlights a potential limitation of the ternary method in smaller knowledge bases, where the additional division may introduce unnecessary complexity without providing a comparable performance benefit.

The data demonstrates that caching, while not advantageous for unique query sets, significantly enhances performance for query sets involving repetition. This is consistent with the expected behaviour of caching, which reduces redundant calculations, particularly for repeated queries or antecedents of queries. In practice, the binary search method emerges as a robust optimization over the naive method and should be the default choice for optimizing rational closure, especially in scenarios where the distribution of statements across ranks is unknown or varied. The ternary method, while promising in specific contexts, may be more suitable for large knowledge bases with specific distribution characteristics that justify its additional complexity.

The significant reduction in runtime for these types of queries suggests that caching can make rational closure more scalable, allowing the system to handle larger knowledge bases and higher query loads more efficiently. This adaptability is crucial for real-world deployment in dynamic environments where query types can vary, such as in automated reasoning systems and ontology management systems.

Further exploration could focus on optimizing the system for more complex query patterns, such as nested queries or queries involving multiple dependencies, which are common in advanced reasoning tasks. In real-time systems where query response time is critical, the

benefits of caching must be carefully balanced against the overhead of managing the cache to ensure that the system remains responsive.

In conclusion, this research successfully answered the initial research question by demonstrating that caching, alongside binary and ternary search methods, significantly improves the scalability of rational closure within the KLM framework. The fundamental discoveries include the consistent performance gains provided by the binary method, the potential specific applications of the ternary method, and the critical role of caching in handling repeated queries. However, the research also highlights potential concerns, such as the scalability of the ternary method in smaller knowledge bases and the need for efficient cache management in larger systems.

Further studies should investigate the conditions under which the ternary method might outperform binary, particularly in knowledge bases with highly complex or irregular rank distributions. Additionally, exploring the impact of caching on more complex query sets or larger knowledge bases could provide deeper insights into the scalability of rational closure. The contribution of this research lies in its comprehensive evaluation of caching and search optimizations for rational closure, backed by rigorous statistical analysis using the Wilcoxon signed-rank test. By testing knowledge bases with up to 2000 statements, this work extends the understanding of the practical limits and advantages of these methods beyond previous implementations, providing a solid foundation for future advancements in the field.

REFERENCES

[1] Grigoris Antoniou and Mary-Anne Williams. 1997. *Nonmonotonic Reasoning*. MIT Press. https://doi.org/10.7551/mitpress/5040.001.0001

[2] P.A. Bonatti. 2018. Rational closure for all description logics. Article.

[3] H. K. Büning and T. Lettmann. 1999. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press.

[4] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2018. Defeasible entailment: From rational closure to lexicographic closure and beyond. In *Proceeding of the 17th International Workshop on Non-Monotonic Reasoning (NMR 2018)*. 109–118.

[5] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2019. Taking defeasible entailment beyond rational closure. In *European Conference on Logics in Artificial Intelligence*. 182–197.

[6] Victoria Chama. 2019. *Explanation For Defeasible Entailment*. Master's Thesis. Faculty of Science, University of Cape Town.

[7] Joel Hamilton, Daniel Park, Aidan Bailey, and Thomas Meyer. 2021. An Investigation into Optimisations of Defeasible Reasoning Algorithms. In *SACAIR'21 Proceedings Volume II*. University of Cape Town and Centre for Artificial Intelligence Research (CAIR), Cape Town, South Africa.

[8] Michael Harrison and Thomas Meyer. 2020. DDLV: A system for rational preferential reasoning for datalog. *South African Computer Journal* 32, 2 (2020). https://doi.org/10.18489/sacj.v32i2.850

[9] http://www.statskingdom.com. 2017. Wilcoxon Signed-Rank Test Calculator. https://www.socscistatistics.com/tests/signedranks/. Accessed: August 20, 2024.

[10] Internet Encyclopedia of Philosophy. [n. d.]. *Propositional Logic*. https://iep.utm.edu/propositional-logic-sentential-logic/ Accessed: Mar. 26, 2024.

[11] Adam Kaliski. 2020. *An Overview of KLM-Style Defeasible Entailment*. Master's Thesis. Faculty of Science, University of Cape Town.

[12] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence* 44, 1-2 (1990), 167–209. https://doi.org/10.1016/0004-3702(90)90101-5

[13] Daniel Lehmann. 1995. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence* 15, 1 (01 03 1995), 61–82. https://doi.org/10.1007/BF01535841

[14] Daniel Lehmann and Menachem Magidor. 1992. What does a conditional knowledge base entail? *Artificial Intelligence* 55, 1 (1992), 1–60. https://doi.org/10.1016/0004-3702(92)90041-U

[15] Maqhobosheane Mohlerepe. 2024. Extending Defeasible Reasoning Beyond Rational Closure. Literature review.

[16] Maqhobosheane Mohlerepe, Thabo Moloi, and Sadiki. 2024. Extending Defeasible Reasoning Beyond Rational Closure. Project Proposal.

[17] Thabo Moloi. 2024. *Extending Defeasible Reasoning Beyond Rational Closure*. Technical Report. University of Cape Town.

[18] Thabo Vincent Moloi. 2024. Extending Defeasible Reasoning beyond Rational Closure. Literature Review.

[19] Matthew Morris, Tala Rossa, and Thomas Meyer. 2020. Algorithmic definitions for KLM-style defeasible disjunctive Datalog. *SACJ* 32, 2 (December 2020). Research Article.

[20] Evashna Pillay. 2022. An Investigation into the Scalability of Rational Closure V2.

[21] Mamodike Sadiki. 2024. A Literature Review on Extending Defeasible Reasoning Beyond Rational Closure. Literature review.

[22] Mamodike Sadiki. 2024. *A Study of Parameters and Optimisation Strategies in Defeasible Knowledge Base Generation*. Technical Report. University of Cape Town.

[23] Yoav Shoham. 1987. A semantical approach to nonmonotonic logics. In *Logic in Computer Science*. https://api.semanticscholar.org/CorpusID:117644704

[24] Luke Slater and Thomas Meyer. 2023. Extending Defeasible Reasoning Beyond Rational Closure. In *Artificial Intelligence Research*, Anban Pillay, Edgar Jembere, and Aurona J. Gerber (Eds.). Springer Nature Switzerland, Cham, 151–171.

A P-VALUE RESULTS

A.1 Comparison Using Number of Statements

Knowledge Base	Naïve Vs Binary	Binary Vs Ternary	Ternary Vs Naïve	Naïve Vs Naïve Cache	Binary Vs Binary Cache	Ternary
0.5k50	0.01563	0.2188	0.01563	0.2188	0.2041	
1k50	0.01563	0.1563	0.01563	0.1094	0.07813	
2k50	0.01563	0.2188	0.01563	0.07813	0.07813	

Table 3: Comparison Using Number of Statements

A.2 Comparison Using Number of Ranks

Knowledge Base	Naïve Vs Binary	Binary Vs Ternary	Ternary Vs Naïve	Naïve Vs Naïve Cache	Binary Vs Binary Cache	Ternary
10ranks	0.01563	0.01563	0.01563	0.1094	0.1563	
50ranks	0.01563	0.1563	0.01563	0.1094	0.07813	
100ranks	0.01563	0.1563	0.01563	0.01563	0.2041	

Table 4: Comparison Using Number of Ranks

A.3 Comparison Using Statement Distribution

Knowledge Base	Naïve Vs Binary	Binary Vs Ternary	Ternary Vs Naïve	Naïve Vs Naïve Cache	Binary Vs Binary Cache	Ternary
uniform10	0.01563	0.01563	0.01563	0.1094	0.1563	
normal10	0.03125	0.375	0.2969	0.1563	0.01563	
exp10	0.01563	0.07813	0.1563	0.07813	0.4688	

Table 5: Comparison Using Statement Distribution

A.4 P-Value Results for Query Sets

Query Set	Naïve Vs Naïve Cache	Binary Vs Binary Cache	Ternary Vs Ternary Cache	Naïve Vs Binary	Binary Vs Ternary	Ter
Unique	0.9375	0.2188	0.9375	0.01563	0.01563	
Repeated	0.01563	0.07813	0.01563	0.01563	0.01563	
HalfRepeated	0.01563	0.01563	0.01563	0.03125	0.2188	
SameQuery	0.01563	0.01563	0.01563	0.01563	0.1563	
Mixed	0.02225	0.02225	0.02225	0.02225	0.02225	
FirstRank	0.8125	0.9375	0.09375	0.01563	0.5781	
LastRank	0.6875	0.9375	0.2188	0.01563	0.4688	

Table 6: P-Values for Various Query Sets