UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2022

Title: An Investigation into the Scalability of Rational Closure V2

Author: Evashna Pillay

Project Abbreviation: SCADR2

Supervisor(s): Professor Thomas Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	15
Experiment Design and Execution	*0*	*20*	8
System Development and Implementation	*0*	*20*	12
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	10
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**	

An Investigation into the Scalability of Rational Closure V2

Evashna Pillay
University of Cape Town
Cape Town, South Africa
PLLEVA005@myuct.ac.za

ABSTRACT

Knowledge representation and reasoning formally represents and enables reasoning of information about the world. This is an approach used in artificial intelligence (AI). Although many reasoning tasks can be efficiently completed by reasoning systems based on classical propositional logic, it is likely that a contradiction will occur when additional knowledge is obtained [16]. Non-monotonic reasoning, specifically defeasible reasoning allows previously held beliefs to be retracted when presented with additional information [2]. Defeasible reasoning addresses atypical reasoning scenarios in which humans think differently from one another due to beliefs, context or other factors. Thus, AI systems may emulate human thinking more accurately with defeasible reasoning rather than with classical reasoning. Lehmann and Magidor proposed a rational method of defeasible entailment, namely the Rational Closure. Defeasible entailment determines whether a defeasible implication is entailed (i.e. can be inferred) by a knowledge base. While there has been some work on developing efficient systems which support defeasible reasoning, little has been done to improve the scalability of these systems. The goal of this project is to build on previous work to design and implement algorithms for the Rational Closure approach and to evaluate the extent to which these algorithms are scalable.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**; • **Computing methodologies → Nonmonotonic, default reasoning and belief revision**.

KEYWORDS

artificial intelligence, knowledge representation and reasoning, defeasible reasoning

1 INTRODUCTION

Knowledge representation (KR) utilizes symbols to represent knowledge about the world, in which inferences are established and new elements of knowledge are formed. We will differentiate between propositional reasoning and defeasible reasoning in terms of representing information. Propositional reasoning is monotonic as previously drawn conclusions cannot be contradicted by new information. To illustrate this logic, we will use the following example, "Birds can fly." We know birds have wings. Penguins are birds as they have wings, therefore penguins can fly. However, we also know that penguins cannot fly. Thus, propositional reasoning does not have the ability to model human reasoning and limits any possibility of exceptions in the knowledge base. Our research focuses on non-monotonic reasoning, this allows for a more "common-sense" approach to reasoning in which elements of knowledge are typically true and not always definite. This paper presents a scalable defeasible console-application that that we refer to as reasoners. The reasoners can dismiss commonly held beliefs when presented with new contradictory information. Our focus will be on optimizing the execution time and scalability of our application in comparison to last year's honours project, with focus on ternary search and concurrency.

In conjunction with this research, T. Vallabh will be building on previous work to design and implement algorithms for the Lexicographic Closure and will be evaluating the extent to which these algorithms are scalable [19]. Lexicographic Closure follows presumptive reasoning, while Rational Closure follows prototypical reasoning [12]. Presumptive reasoning allows atypical formulae to inherit the properties of more typical formulae. Lexicographic Closure utilizes the Base Rank algorithm. In addition to this, formulae within the ranks are ranked where the most typical formulae are ranked higher. This approach allows us to remove one statement within a rank instead of the entire rank. However, if all statements are removed from a rank, only then will the entire rank be removed.

This paper will provide an overview of the principles and concepts that set the foundation for our reasoners. This includes a brief summary of propositional logic, defeasible reasoning, entailment, the KLM and Rational Closure approaches. We will then present a summary of related work. Following this, we will discuss the project aims, design and implementation of our reasoners, as well as experiment design. Lastly, a detailed summary of our results will be presented along with conclusions we have deduced from our findings.

2 THEORETICAL FRAMEWORK

2.1 Propositional Logic

Propositional logic is a framework that uses logical statements to represent information about the world, these statements are referred to as *formulas* [16]. Each formula consists of *propositional atoms* which can be assigned *truth values* (i.e., true or false). *Connectives* (i.e., $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) are used to join propositional atoms [16] [10].

2.1.1 Entailment. A knowledge base, $\mathcal{K}$ *entails* a formula, α (i.e., $\mathcal{K} \models \alpha$) when every model of $\mathcal{K}$ is satisfied by α. Propositional logic and therefore propositional entailment is monotonic. This means that any new information that creates a contradiction can never be defined as an exception [18]. However, the knowledge base must still reconcile with this new information and can form incorrect

Operator Symbol	Operator Name
$\neg$	Negation
$\wedge$	Conjunction
$\vee$	Disjunction
$\rightarrow$	Implication
$\leftrightarrow$	Equivalence

Table 1: Connectives used to join propositional atoms

conclusions. Thus, knowledge base generation becomes redundant which is why we require a non-monotonic approach [11] [13].

2.2 Defeasible Reasoning

Defeasible reasoning allows the reasoner to dismiss commonly held beliefs when presented with new contradictory information [15]. Defeasible reasoning addresses atypical reasoning scenarios in which humans think differently from one another due to beliefs, context, or other factors. Thus, AI systems may emulate human thinking more accurately with defeasible reasoning rather than with classical reasoning.

To illustrate defeasible reasoning, an example is provided below:

E.g., "Tweety is a bird" and the reasoner contains the following statements in the knowledge base:

(1) Most birds can typically fly.

(2) Birds have wings.

(3) Penguins are birds.

(4) Tweety is a penguin.

The reasoner can deduce that Tweety is a bird from (3). The reasoner can then deduce that Tweety can fly because (1) follows (3). However, we know that penguins cannot fly due to their biological makeup. As a result, penguins are distinct from other birds. With the additional information made available to the reasoner, the reasoner's understanding that birds normally fly is weakened.

The KLM framework specifically for defeasible reasoning provides a preferential approach to construct defeasible systems. A *preferential consequence relation* is indicated by $\mid\sim$. It is a set of conditional assertions, written as $\alpha \mid\sim \beta$ where $\alpha, \beta \in \mathcal{L}$. This is interpreted as follows, from α we are prepared to conclude β unless we receive contradictory information. As such, any defeasible assertion can be retracted after discovering contradictory information.

2.3 Entailment

The addition of new formulas to the knowledge base should not contradict current statements in $\mathcal{K}$. However, the knowledge base must still reconcile with this new information and this can form incorrect conclusions. Specifically, when contradictory statements are added to $\mathcal{K}$, no models of $\mathcal{K}$ are satisfied, which results in every statement being true [18]. This makes knowledge bases redundant which is why we require a non-monotonic approach [11] [13], such as the KLM style framework.

Satisfiability (SAT) solvers can be used to check if $\mathcal{K} \models \alpha$. The Boolean Satisfiability Problem is a NP-complete problem that determines if there exists some interpretation which satisfies a given Boolean formula [5]. The SAT solver checks the satisfiability of $\mathcal{K}$ by checking if the models of $\mathcal{K}$ satisfy $\mathcal{K} \cup \{\neg\alpha\}$.

2.4 KLM Approach to Defeasible Entailment

The KLM approach models statements written in the form of $\alpha \mid\sim \beta$, which is interpreted as "α typically implies β". This means that we are prepared to conclude β from α unless we receive contradictory information. Defeasible reasoning, unlike propositional logic, does not define a fixed method to determine defeasible entailment. Thus to determine whether a defeasible implication is entailed by a knowledge base, the method must only adhere to the rationality properties defined by Lehmann and Magnidor [10].

The rationality properties for knowledge base $\mathcal{K}$ and propositional formulas α, β, γ [10]:

$$\text{(Ref) } \mathcal{K} \mid\approx \alpha \mid\sim \alpha \qquad \text{And } \frac{\mathcal{K}\mid\approx\alpha\mid\sim\beta, \mathcal{K}\mid\approx\alpha\mid\sim\gamma}{\mathcal{K}\mid\approx\alpha\mid\sim\beta\wedge\gamma}$$

$$\text{(LLE) } \frac{\mathcal{K}\mid\approx\alpha\leftrightarrow\beta, \mathcal{K}\mid\approx\alpha\mid\sim\gamma}{\mathcal{K}\mid\approx\beta\mid\sim\gamma} \qquad \text{Or } \frac{\mathcal{K}\mid\approx\alpha\mid\sim\gamma, \mathcal{K}\mid\approx\beta\mid\sim\gamma}{\mathcal{K}\mid\approx\alpha\vee\beta\mid\sim\gamma}$$

$$\text{(RW) } \frac{\mathcal{K}\mid\approx\alpha\rightarrow\beta, \mathcal{K}\mid\approx\gamma\mid\sim\alpha}{\mathcal{K}\mid\approx\gamma\mid\sim\beta} \qquad \text{(CM) } \frac{\mathcal{K}\mid\approx\alpha\mid\sim\gamma, \mathcal{K}\mid\approx\alpha\mid\sim\beta}{\mathcal{K}\mid\approx\alpha\wedge\beta\mid\sim\gamma}$$

Two methods which fall within the KLM approach are Rational Closure and Lexicographic Closure. This paper will focus on the Rational Closure approach.

2.5 Rational Closure

Rational Closure is an LM-rational method of defeasible entailment [7]. We will define minimal ranked entailment, materialisation, Base Rank algorithm before providing an overview of the Rational Closure Algorithm.

Definition 2.5.1 *Minimal Ranked Entailment.* The lower the ranked model, the more likely there exists a minimum ranked model in the ordering. This principle involves a partial order for all ranked models of knowledge base $\mathcal{K}$ referred to as $\preceq_{\mathcal{K}}$. The Rational Closure of a knowledge base $\mathcal{K}$ is based on finding this minimum ranked model [7] as the minimum ranked model must satisfy some defeasible implication for the knowledge base $\mathcal{K}$ to entail it [3].

Definition 2.5.2 *Materialisation.* The material counterpart of a defeasible implication $\alpha \mid\sim \beta$ is the propositional formula $\alpha \rightarrow \beta$ [4]. The material counterpart of a defeasible knowledge base $\mathcal{K}$ is denoted as $\overrightarrow{\mathcal{K}}$ and represents the set of material counterparts, $\alpha \rightarrow \beta$, for every defeasible implication $\alpha \mid\sim \beta \in \mathcal{K}$.

E.g., $bird \mid\sim flies$ is replaced with $bird \rightarrow flies$.

Definition 2.5.3 *The Base Rank Algorithm.* The Base Rank algorithm is the initial step performed when implementing Rational Closure to $\mathcal{K}$ to determine the minimum ranked model [17]. Each formula in $\overrightarrow{\mathcal{K}}$ is mapped to a rank in $\mathcal{N} \cup \{\infty\}$, this denotes the exceptional subset of $\mathcal{K}$, $\mathcal{E}_n^{\mathcal{K}}$.

The BaseRank Algorithm separates $\mathcal{K}$ into classical (i.e., $\mathcal{K}_C$) and defeasible (i.e., $\mathcal{K}_D$) components. The materialisation of $\mathcal{K}_D$ takes place and forms $\overrightarrow{\mathcal{K}_D}$, this is then referred to as $\mathcal{E}_0^{\mathcal{K}}$. The algorithm will rank all statements in the current materialisation according to how general they are, with the most general being in rank 0 and

Algorithm 1: BaseRank

Data: A knowledge base $\mathcal{K}$
Result: An ordered tuple $(R_0, R_1, \ldots, R_{n-1}, R_\infty, n)$

$i := 0$;
$E_0 := \overrightarrow{\mathcal{K}}$;
while $E_{i-1} \neq E_i$ **do**
  $E_{i+1} := \{\alpha \rightarrow \beta \in E_i | E_i \models \neg\alpha\}$;
  $R_i := E_i \backslash E_{i+1}$;
  $i := i + 1$;
end
$R_\infty := E_{i-1}$;
if $E_{i-1} = \emptyset$ **then**
  $n := i - 1$;
else
  $n := i$;
end
return $(R_0, R_1, \ldots, R_{n-1}, R_\infty, n)$;

the least general at the bottom rank. For each $\alpha \rightarrow \beta \in \mathcal{E}_0^{\mathcal{K}}$, the algorithm determines if $\mathcal{E}_0^{\mathcal{K}} \cup K_C \models \neg\alpha$. If true, α is *exceptional* and all formulas within $\mathcal{E}_0^{\mathcal{K}}$ with α are moved to $\mathcal{E}_1^{\mathcal{K}}$. Rank $\mathcal{R}_0^{\mathcal{K}}$ is assigned to the formulas $\mathcal{E}_0^{K} \backslash \mathcal{E}_1^{\mathcal{K}}$. This process is repeated for each subset, i.e., $\mathcal{E}_i^{\mathcal{K}}$ until no more formulas need to be copied/assigned. The final rank contains $\mathcal{R}_\infty^{K}$ and K_C, which are always true.

Definition 2.5.4 *The Rational Closure Algorithm.* Rational Closure will make use of $\mathcal{R}^K$ from the Base Rank algorithm, given a knowledge base $\mathcal{K}$ and some defeasible implication, $\alpha \mid\sim \beta$.

Algorithm 2: RationalClosure

Data: A knowledge base $\mathcal{K}$ and a defeasible implication $\alpha \mid\sim \beta$
Result: **true** if $\mathcal{K} \mid\approx \alpha \mid\sim \beta$, and **false**, otherwise

$(R_0, R_1, \ldots, R_{n-1}, R_\infty, n)$:= BaseRank($\mathcal{K}$);
$i := 0$;
$R := \bigcup_{i=0}^{j<n} R_j$;
while $R_\infty \models \neg\alpha$ ***and*** $R \neq \emptyset$ **do**
  $R := R \setminus R_i$;
  $i := i + 1$;
end
return $R_\infty \bigcup R \models \alpha \rightarrow \beta$;

The Rational Closure Algorithm determines if (1) $\overrightarrow{\mathcal{K}}$ entails $\neg\alpha$. If this is not the case, α is compatible with $\mathcal{K}$. Then the algorithm checks if $\overrightarrow{\mathcal{K}}$ entails $\alpha \rightarrow \beta$ (i.e. the defeasible query). If this is the case, α is incompatible with $\mathcal{K}$. The most preferred rank is then withdrawn from $\mathcal{R}^{\mathcal{K}}$. The resulting knowledge base will be denoted as $\mathcal{K}'$. If $\mathcal{R}^{\mathcal{K}'}$ is an empty set, then $\mathcal{K} \not\mid\approx \alpha \mid\sim \beta$. If $\mathcal{R}^{\mathcal{K}'}$ contains at least one rank, we return to (1) with $\mathcal{K}'$.

The following example illustrates the Rational Closure algorithm: Consider the knowledge base $\mathcal{K} = \{b \mid\sim f, p \rightarrow b, p \mid\sim \neg f, r \rightarrow b, b \mid\sim w\}$. This is a continuation of the penguins and birds example mentioned in *Section 2.2. Robins are birds* (i.e., $r \rightarrow b$) and *birds typically have wings* (i.e., $b \mid\sim w$) are added to the knowledge base $\mathcal{K}$.

Using the Base Rank algorithm: $K_C = \{p \rightarrow b, R \rightarrow b\}$ and $K_D = \{b \rightarrow f, p \rightarrow \neg f, b \rightarrow w\}$. p is exceptional and found in $\mathcal{E}_0^{\mathcal{K}}$.

The algorithm produces the following ranking $\mathcal{R}^{\mathcal{K}}$:

0	$b \rightarrow f, b \rightarrow w$
1	$p \rightarrow \neg f$
∞	$p \rightarrow b, R \rightarrow b$

Using the Rational Closure Algorithm, we determine if $\mathcal{K}$ entails the query $p \rightarrow \neg f$, that is *penguins don't fly*. We deduce there is no model u of $\overrightarrow{\mathcal{K}}$ such that $u \Vdash p$. Therefore, we retract the most preferred rank from $\mathcal{R}^{\mathcal{K}}$, that is $\mathcal{R}_0^{\mathcal{K}}$.

1	$p \rightarrow \neg f$
∞	$p \rightarrow b, R \rightarrow b$

The algorithm will repeat the first step and determine if $\overrightarrow{\mathcal{K}} \models \neg p$, which is not the case. There exists a model u of $\overrightarrow{\mathcal{K}}$ such that $u \Vdash p$, e.g, $\{pbr\bar{f}\}$. $\overrightarrow{\mathcal{K}}$ does not entail $p \rightarrow \neg f$ as $p \rightarrow \neg f$ is a formula within $\overrightarrow{\mathcal{K}}$ and as such, any model of $\overrightarrow{\mathcal{K}}$ is also a model of $p \rightarrow \neg f$. Thus, we can conclude that $\mathcal{K} \mid\approx p \rightarrow \neg f$.

3 RELATED WORK

3.1 SAT Solvers

The Rational Closure algorithm reduces to a number of classical entailment checks. Therefore, Boolean satisfiability is applicable to our research. Boolean satisfiability (SAT) establishes whether an assignment exists that satisfies a specific Boolean formula [5] [21]. We have identified which SAT solvers are appropriate for our defeasible reasoning model by evaluating the SAT solvers for classical reasoning.

3.1.1 Semantic Tableaux. The tableau is constructed by decomposing a formula into sets of atomic literals, resulting in a tree-like tableau [16]. Formulae can contain a single clause or conjunction of clauses. Clauses are connected by the logical operator $\wedge$. A clause is a disjunction of literals where each literal is an atom or its negation. E.g., $(\alpha \vee b) \wedge (\neg\alpha \vee c)$ [9].

Each branch either contains a group of non-contradictory literals (i.e., referred to as an open branch) or ends with a complementary pair of formulae (i.e., referred to as a closed branch). Each open branch represents a model for the given formula. The construction of the tableau is complete when decomposition is no longer possible. If there is a conflict, the initial formula is unsatisfiable. When literals for an atom are located within the same subset there is a conflict, which leads to a contradiction. To determine a formula's satisfiability a completed tableau is required [8].

3.1.2 DPLL. DPLL is a backtracking algorithm [6] that accepts input in the form of a propositional formula in CNF format, returning true if the formula is satisfiable, or false if it is not. A branching procedure is executed where some atom in α is set to a random truth value. Until an assignment returns true and satisfies α branching continues; if α is false, the algorithm backtracks. Backtracking involves retrieving the recent branching assignment and re-branching with a different assignment. Backtracking occurs when there are no new assignments to branch to. If there exists no new branches to be taken and we are unable to backtrack, false is returned and α is unsatisfiable.

3.1.3 Conflict-Driven Clause Learning (CDCL). The DPLL SAT solver led to the development of the CDCL [14]. Backtracking in the CDCL SAT solver is non-chronological, unlike the DPLL SAT solver [6]. Conflicts caused by variable assignments are cached. This results in improved efficiency and performance.

3.2 SCADR 2021

There are currently only two implementations of defeasible entailment for propositional logic that aligns with the KLM approach of defeasible entailment. Each one focuses on the LM-rational defeasible entailment algorithms [7], namely Lexicographic and Rational Closure. This paper focuses on the latter. An investigation into the scalability of the Rational Closure was explored by Hamilton [8]. He proposed optimizing the *RationalClosure* algorithm using binary search, as well as with a binary indexing approach. He found that his approach outperformed the naive implementation when *a* became consistent with the knowledge base at ranks with larger numbers. These developments have created efficient systems but more can be done to improve the scalability of these systems going forward.

4 PROJECT AIMS

The key aims of this project were:

- To develop an optimised reasoner console-application that integrates an existing propositional reasoner and is based on some defeasible knowledge base that implements the Rational Closure algorithm.
- To refine the optimisation approaches used to increase the scalability of the Rational Closure [13] implementation compared to the previous year's SCADR project [9].
- To acquire empirical results that show the effective performance of our optimization techniques considering size of knowledge bases and number of queries.

5 SYSTEM DEVELOPMENT AND IMPLEMENTATION

Two defeasible reasoners were developed that given a defeasible implication statement in the form, $\alpha \mid\sim \beta$ and knowledge base K, returns whether or not the query is entailed by K. The reasoners were developed using a wrapper around the previous year's reasoners [1] [8] [18] in Java, along with the *TweetyProject* library (i.e., version 1.20) [20]. Sat-solvers were used to determine the satisfiability of the queries, specifically the built-in Sat4j tool in the *TweetyProject* library. Our reasoners adhere to the KLM Properties set out by Kraus, Lehmann and Magidor. More so, our reasoners passed the test cases provided. Therefore, we strongly believe in the validity and correctness of our reasoners. The first reasoner optimizes the previous year's binary search implementation using ternary search to reduce the size of intervals searched. The second reasoner optimizes the indexing implementation using ternary search, as well as concurrency. This allows for multiple queries in one instance. To ensure for completeness, the application outputs the ranking from the `BaseRank` algorithm. Additionally, the application will provide a synopsis of each material step taken by the reasoner, such as the removal of ranks. To download, view and run the code refer to the following link.

5.1 Optimisation 1: Ternary Search Entailment Approach

The first reasoner developed uses ternary search in its implementation, this will be referred to as `RCTerChecker`, which is an amended version of the Rational Closure. Ternary search is an inductive approach using the decrease and conquer technique to locate the element in the data structure. This approach is similar to binary search however, instead of dividing the search range into two parts we divide it into three parts resulting in two midpoint values. The main distinction between binary and ternary search is the time complexity, as ternary outperforms binary with a time complexity of *O(log n base 3)* in comparison to *O(log n base 2)* of binary search. `RCTerChecker` determines the rank from which all ranks need to be removed from rank 0, in contrast to iterating from the top towards the lower ranks in the naive implementation of the `RationalClosure`. The implementation is described below:

(1) The first computation of the algorithm sets a min value of 0 and a max value equal to the number of ranks n. This signifies the initial search range.

(2) The two midpoints are then determined given the min and max values, we will refer to these midpoints as $mid1$ and $mid2$.

(3) Check if removing rank $mid1$ and all those above it results in α being consistent with the knowledge base.
   - If **true**, go to (4).
   - If **false**, go to (5)

(4) Determine if adding rank $mid1$ back to K results in α being consistent with the knowledge base.
   - If **true**, $mid1$ is the rank from which we need to remove all ranks, including rank $mid1$. Go to (9).
   - If **false**, return to (1), set min to $left$ and set max to $mid1$. We have to search in the upper half of the ranks as α is consistent with K from ranks $mid1$ and down. Therefore, we need to remove more ranks further up (i.e., ranks with lower numbers).

(5) Check if mid2 < rankedKB.length.
   - If **true**, go to (6).

- If **false**, go to (7)

(6) Check if removing rank *mid*2 and all those above it results in α being consistent with the knowledge base.

- If **true**, go to (8).
- If **false**, return to (1), set min to $mid2 + 1$ and max to $right$. We have to search in the lower half of the ranks as α is still not consistent with K from ranks 0 down to and including rank *mid*2. Therefore, we need to remove more ranks further down (i.e., ranks with higher numbers).

(7) Check if mid2 == rankedKB.length.

- If **true**, return to (1), set min to $mid1 + 1$ and set max to $mid2 - 1$. We have to search in the upper half of the ranks as α is consistent with K from ranks *mid*2 and down. Therefore, we need to remove more ranks further upper (i.e., ranks with lower numbers).

(8) Determine if adding rank *mid*2 back to K results in α being consistent with the knowledge base.

- If **true**, *mid*2 is the rank from which we need to remove all ranks, including rank *mid*2. Go to (9).
- If **false**, return to (1), set min to $mid1 + 1$ and set max to $mid2 - 1$. We have to search in the lower half of the ranks as α is consistent with K from ranks *mid*2 and down. Therefore, we need to remove more ranks further up (i.e., ranks with lower numbers).

(9) Determine if $\alpha \mid\sim \beta$ is entailed by the ranked knowledge base with all ranks up to and including *mid*1 removed, return **true** or **false** otherwise.

We hypothesised that this optimisation technique would perform better than the naive implementation of the `RationalClosure` when checking for entailment at the lower ranks (i.e., ranks with higher numbers) and not necessarily when the number of ranks increase. This is due to the optimisation iterating over smaller ranges of the ranks and not requiring a linear iteration over all ranks to perform the entailment check. It is also hypothesized that this approach will perform better than the binary implementation from the previous year's project, specifically for defeasible queries $\alpha \mid\sim \beta$ where a becomes consistent at ranks considerably larger than $log(n)$ where n is the number of ranks.

While both binary and ternary have *O(log n)* time complexity for the average and best cases, ternary search will perform better (i.e., faster) when a becomes consistent with the knowledge base at lower ranks due to the interval being split into three parts and the algorithm's ability to "throw-away" two-thirds of the searching range. However, if the rank at which a becomes consistent is less than or equal to $log\ n$, then the naive and binary approach would outperform this optimisation. Similar to the previous year's implementation, we have to consider the Boolean satisfiability problem. Whilst we have optimised the number of entailment checks, checking for entailment still reduces to the Boolean satisfiability problem, which is NP-complete. Still, in the best and average cases our implementation can provide improvements in performance.

Algorithm 3: RCTerChecker

Data: A knowledge base $\mathcal{K}$ and a defeasible implication $\alpha \mid\sim \beta$
Result: **true** if $\mathcal{K} \not\approx \alpha \mid\sim \beta$, and **false**, otherwise

```
rankedKB := BaseRank(K);
f := α |~ β;
low := 0;
high := n;
if high > low then
    mid1 = low + (high − low)/3;
    mid2 = high − (high − low)/3;
    if ⋃_{i=mid1+1}^{j<n} R_j ⋃ R_∞ ⊨ ¬α then
        if mid2 < rankedKB.length then
            if ⋃_{i=mid2+1}^{j<n} R_j ⋃ R_∞ ⊨ ¬α then
                return RCTIC(rankedKB, f, mid2 + 1, high, neg);
            else
                if ⋃_{i=mid2}^{j<n} R_j ⋃ R_∞ ⊨ ¬α then
                    rankRemove = mid2;
                else
                    return RCTIC(rankedKB, f, mid1 + 1, mid2 − 1, neg);
                end
            end
        else if mid2 == rankedKB.length then
            return RCTIC(rankedKB, f, mid1 + 1, mid2 − 1, neg);
    end
    if ⋃_{i=mid1}^{j<n} R_j ⋃ R_∞ ⊨ ¬α then
        rankRemove = mid1;
    else
        return RCTIC(rankedKB, f, left, mid, neg);
    end
end
if rankRemove + 1 < rlength then
    if ⋃_{i=rankRemove+1}^{j<n} R_j ⋃ R_∞ ⊨ f then
        return true;
    else
        return false;
    end
else
    return true;
end
```

5.2 Optimisation 2: Ternary Indexing Entailment Approach

The second reasoner developed uses ternary search in its implementation, as well as concurrency this will be referred to as `RCTerIndexChecker`, which is an amended version of the

RCTerChecker. This concurrency implementation is a divide and conquer approach, using the fork/join framework it attempts to use all available processor cores to speed up parallel processing. More so, the framework uses ForkJoinPool, which manages a pool of worker threads (i.e., of type ForkJoinWorkerThread). The framework "forks" that is recursively splits the program into smaller independent subtasks that run asynchronously. After which, the "join" is called which recursively joins into a single result. This approach is used to increase the throughput and computational speed of the system by using multiple processors. In this implementation, the negations of the antecedents along with the rank at which the antecedents become consistent with the knowledge base are stored in a hashtable. Similar to last year's implementation, the console-application allows the user to enter multiple queries in one instance. Even though multiple defeasible queries with the same antecedent requires only one computation, we must note that this optimisation can be memory-intensive for large sets of queries.

We hypothesised that this optimisation technique would perform better than the naive implementation of the RationalClosure and the RCTerChecker, when checking for entailment of query sets that contain a large set of the same antecedent. Additionally, we believe that this optimization will perform better than the RationalClosure as the number of ranks increase as ternary search is notably faster than the linear search, as well as computing multiple entailment checks in parallel will increase the computational speedup of the system. We have formalised the implementation as follows:

Algorithm 4: RCTerIndexChecker

Data: A knowledge base $\mathcal{K}$ and a defeasible implication $\alpha \mid\sim \beta$
Result: **true** if $\mathcal{K} \not\models \alpha \mid\sim \beta$, and **false**, otherwise

$rankedKB := BaseRank(\mathcal{K})$;
$negRanks := hashTable$;
$f := \alpha \mid\sim \beta$;
$low := 0$;
$high :=$ n;
if $negRank.get(negation) != null$ **then**
  $rankRemove = negRank.get(neg)$;
else
  if $high > low$ **then**
    $mid1 = low + (high - low)/3$;
    $mid2 = high - (high - low)/3$;
    if $\bigcup_{i=mid1+1}^{j<n} R_j \cup R_\infty \models \neg\alpha$ **then**
      if $mid2 < rankedKB.length$ **then**
        if $\bigcup_{i=mid2+1}^{j<n} R_j \cup R_\infty \models \neg\alpha$ **then**
          return $RCTIC(rankedKB, f, mid2 + 1, high, neg)$;
        else
          if $\bigcup_{i=mid2}^{j<n} R_j \cup R_\infty \models \neg\alpha$ **then**
            $rankRemove = mid2$;
            $negRank.put(negation, mid2)$;
          else
            return $RCTIC(rankedKB, f, mid1 + 1, mid2 - 1, neg)$;
          end
        end
      else if $mid2 == rankedKB.length$ **then**
        return $RCTIC(rankedKB, f, mid1 + 1, mid2 - 1, neg)$;
      end
      if $\bigcup_{i=mid1}^{j<n} R_j \cup R_\infty \models \neg\alpha$ **then**
        $rankRemove = mid1$;
        $negRank.put(negation, mid1)$;
      else
        return $RCTIC(rankedKB, f, left, mid, neg)$;
      end
    end
  end
end
if $rankRemove + 1 < rlength$ **then**
  if $\bigcup_{i=rankRemove+1}^{j<n} R_j \cup R_\infty \models f$ **then**
    return true;
  else
    return false;
  end
else
  return true;
end

6 EXPERIMENT DESIGN AND EXECUTION

6.1 Test Cases

We evaluated our reasoners using existing knowledge bases and query sets from the previous year's project to ensure our testing for comparison is as accurate as possible. Query sets containing only defeasible implication statements were executed by our reasoners.

6.1.1 Knowledge Bases. Knowledge bases generated by the knowledge base generator developed by Aidan Bailey were used [1]. These knowledge bases contain only defeasible statements (i.e., in the format of $\alpha \mid\sim \beta$). Two elements of the knowledge bases that vary are the number of ranks and statement distribution. The number of ranks and distribution of statements in each rank vary as a result of the ranking of the `BaseRank` algorithm.

- The number of ranks used to compare our optimised approaches were ranks of 10, 50 and 100. To ensure our testing is consistent with the previous year's implementation, we used the same number of varying ranks.
- Uniform, normal and exponential distributions of statements with the same number of ranks were generated during the previous year's testing process, we will be adopting these test cases.

6.1.2 Query Sets [1]. Manually generated query sets were used, each which tested a specific characteristic of our optimized implementations. The pool of query sets differ in five distinct ways, we list these characteristics below:

- Query sets containing defeasible implications consistent with the first rank (i.e., rank 0 is removed). These sets will be referred to as `QSFRank`.
- Query sets containing defeasible implications consistent with the last rank (i.e., ranks 0 up to and including the final rank are removed). These sets will be referred to as `QSLRank`.
- Query sets containing defeasible implications with the same antecedents. These sets will be referred to as `QSSameAnt`.
- Query sets containing defeasible implications with unique antecedents (i.e., all antecedents differ). These sets will be referred to as `QSDifAnt`.
- Query sets containing defeasible implications with half unique and half repeated antecedents. These sets will be referred to as `QSHHAnt`.

`QSFRank` tests the execution time between the `RCTerChecker` and the naive implementation of the `RationalClosure`. We hypothesize that the naive approach may be more effective in performance as it performs a linear iteration of the ranks from 0 whereas `RCTerChecker` will check the mid values first and iterate through those intervals until it reaches rank 0. `QSLRank` provides insight into the execution time of the `RCTerChecker` and the `RCTerIndexChecker`. This query set requires the ternary approach to identify the last rank as being consistent with the antecedent as well as remove all ranks and check for entailment. `QSSameAnt` provides insight into the execution time of the `RCTerIndexChecker`. The naive and previous year's binary implementations would compute each query in the set whereas our approach will perform the computation once. Both `QSDifAnt` and `QSHHAnt` provides insight into the execution times of our optimized reasoners in comparison to the naive and binary implementations, here focus is strictly on the computational speedup of our approaches.

6.2 Generating Results

These experiments were run on a MacBook Air (13-inch, 2017 model), with a 1.8GHz dual-core, Intel Core i5 processor and 8GB of RAM. We used an existing program called *TimedReasonerComparison.java* that was modified to measure the execution time of our two optimized reasoners, using `RCTerChecker` and `RCTerIndexChecker` to check for entailment. The execution time was limited to the entailment checks. The ranking of the knowledge base according to the `BaseRank` is not within scope as no alterations were made to the algorithm.

This program requires a knowledge base and query sets in .txt format. Multiple query sets can be input into the program if testing requires more than one. The program uses built-in Java system time calls to measure the execution time of each query in milliseconds, by iterating through the query sets and checking whether or not $\mathcal{K} \mid\approx \alpha \mid\sim \beta$. The program will output the number of ranks, the total number of queries checked and each query along with the execution time to a CSV file.

7 RESULTS AND FINDINGS

The performance of each implementation was measured using the average execution times for each knowledge base and query set combination. This was compared to the performance of the naive implementation, as well as binary and binary-indexing implementations of the `RationalClosure` from last year's project. Appendix A contains the complete set of execution times that were obtained from all tests.

7.1 Ternary Search Entailment Results

7.1.1 Comparative analysis over varying numbers of ranks. The complete set of execution times obtained from all tests are presented in section 2 of Appendix A.

The performance of the `RCTerChecker` in comparison to the `RCBinCheck` of the `RationalClosure` is faster for a 10-rank knowledge base with query set where all antecedents become consistent at rank 1. Whilst the `RCTerChecker` outperforms the `RCBinCheck`, the difference in performance is only 52.95 milliseconds with the average performance of the `RCBinCheck` at 321.31 milliseconds and the `RCTerChecker` at 269.14 milliseconds. It was observed that the performance of the `RCTerChecker` in comparison to the naive implementation of the `RationalClosure` was significantly worse. The naive implementation outperformed our optimization at an average of 50.245 milliseconds. However, the `RCTerChecker` outperforms both the naive implementation and `RCBinCheck` for the 10-rank knowledge base with query sets where all antecedents become consistent at rank 10.

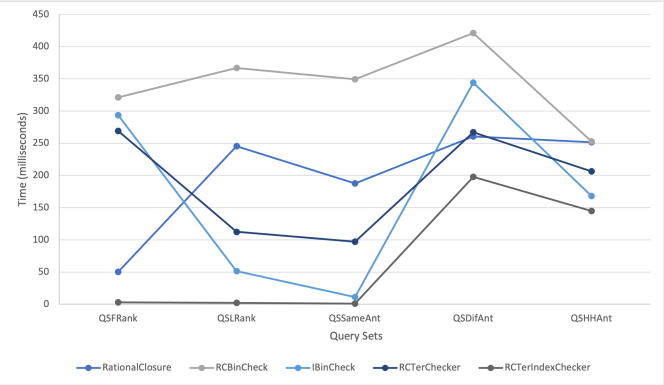

Figure 1: Graphs showing the comparison of all implementations' performance for 10-rank knowledge bases

There is a significant difference in the the performance of the RCTerChecker in comparison to the RCBinCheck implementation of the RationalClosure for a 10-rank knowledge base with query sets where all antecedents are the same. The RCTerChecker outperforms the RCBinCheck by reducing performance by more than a half. The average runtime for the RCBinCheck is 349.50 milliseconds whereas the RCTerChecker is 97,02 milliseconds. For a 10-rank knowledge base with query sets where all antecedents are unique there is a considerable difference in performance with the runtime of the RCTerChecker at 267,05 milliseconds and the RCBinCheck at 420.95 milliseconds. The same can be said for query sets where the antecedents are half repeated and half unique.

The performance of the RCTerChecker in comparison to the RCBinCheck implementation of the RationalClosure is faster for a 50-rank knowledge base for the query set where all antecedents become consistent at rank 1. The RCTerChecker outperforms the RCBinCheck, the difference in performance is 248.42 milliseconds with the average performance of the RCBinCheck at 588.24 milliseconds and the RCTerChecker at 339.83 milliseconds. The performance of the RCTerChecker in comparison to the naive implementation of the RationalClosure was significantly worse similar to the observation of the rank-10 knowledge base. The naive implementation outperformed our optimization at an average of 106.88 milliseconds. However, the RCTerChecker outperforms both the naive and RCBinCheck implementation for the 50-rank knowledge base with query sets where all antecedents become consistent at rank 50.

The RCBinCheck outperforms the RCTerChecker for a 50-rank knowledge base with query sets where all antecedents are the same by 30 milliseconds. For a 50-rank knowledge base with query sets where all antecedents are unique there is a 118.12 millisecond difference in performance with the runtime of the RCTerChecker at 330.06 milliseconds and the RCBinCheck at 448.18 milliseconds. This is compelling evidence that suggests the optimization is most effective when the rankings at which the antecedents of the queries become consistent with the knowledge base are higher in rank number. In addition to this, our findings strongly suggests that the RCTerChecker always outperforms the RCBinCheck when the number of antecedents that are repeated are significantly higher and the number of ranks is small however, when this is not the case the RCTerChecker may be consistent with or outperform the RCBinCheck, but the difference in performance is negligible.

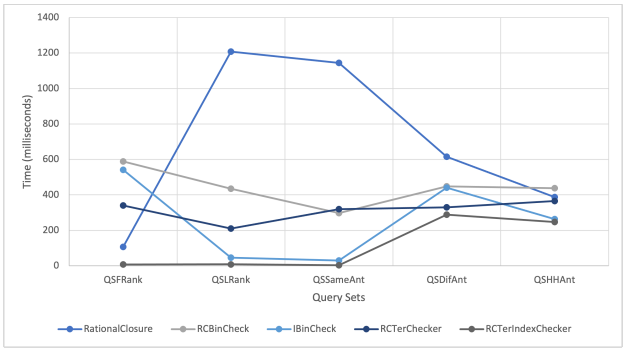

Figure 2: Graphs showing the comparison of all implementations' performance for 50-rank knowledge bases

Similar to the comparisons made for the 50-rank knowledge base, the RCTerChecker outperforms the RCBinCheck for a 100-rank knowledge base with all query sets. In the case of antecedents that are consistent with the rank-1 the naive implementation outperforms the RCTerChecker by 335.51 milliseconds. Taking into account that the execution time measures the entailment checks, one could deduce that the difference in time is accounted for by the increase in number of checks in the ternary approach compared to the linear approach of the naive implementation.

The extent to which the RCTerChecker outperforms the RCBinCheck for query sets with antecedents which become consistent at the final rank tends to increase as the number of ranks in the knowledge base decreases. Overall, the performance of the RCTerChecker was fairly uniform, which disproves the theory that the number of ranks in a knowledge base is a major factor in whether or not the optimization will perform better. Instead, it suggests that the variable of interest is the ranks at which the antecedents of our queries become consistent.

7.1.2 Comparative analysis over varying distributions of statements. The complete set of execution times obtained from all tests are presented in section 3 of Appendix A.

The RCTerChecker performed considerably better than the RCBinCheck for uniform distributions of all query sets except sets that contained the same antecedents, as well as antecedents that become consistent with the knowledge base at the final rank. The RCBinCheck performed slightly better than the RCTerChecker for normal distributions of the query set where the antecedents become consistent with the knowledge base at the final rank. The RCTerChecker performed significantly faster than the RCBinCheck for exponential distributions of all query sets except sets that contained the same antecedents. In cases where the RCBinCheck performed better than the RCTerChecker, there was only a difference of 17 - 100 milliseconds.

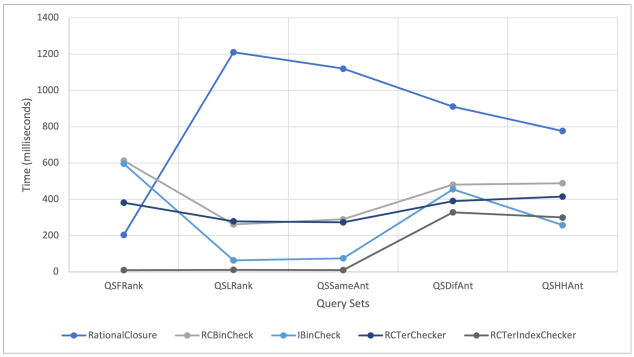

Figure 3: Graphs showing the comparison of all implementations' performance for normal distributions

There is a recurring observation of the RCBinCheck outperforming the RCTerChecker with query sets of the same antecedent, whilst the RCTerChecker outperforms the RCBinCheck with query sets of half repeated and half unique antecedents. This may be due to the distribution of antecedents being consistent with lower ranks as such this is a consequence of the ternary search having two mid values, and thus searching in the higher ranks first before limiting the search range to the lower ranks. We determine that the rank at which the antecedents become consistent with the knowledge base is more important than the distribution of statements over the ranks.

7.2 Ternary Indexing Entailment Results

7.2.1 Comparative analysis over varying numbers of ranks. The complete set of average execution times obtained from all tests are presented in section 2 of Appendix A.

There is a significant difference in the performance of the RCTerIndexChecker in comparison to the IBinCheck implementation of the RationalClosure for a 10-rank knowledge base for all query sets. The RCTerIndexChecker outperforms the IBinCheck for query sets with antecedents that becomes consistent at rank 1 by 290.49 milliseconds, with the average performance of the IBinCheck at 293.39 milliseconds and the RCTerIndexChecker at 2.9 milliseconds. It was observed that the performance of the RCTerIndexChecker in comparison to the naive implementation of the Rational Closure was significantly faster as well. The RCTerIndexChecker outperformed the naive implementation for query sets with antecedents that become consistent at rank 10 by 243.04 milliseconds.

There is a significant difference in the the performance of the RCTerIndexChecker in comparison to the IBinCheck implementation of the RationalClosure for a 10-rank knowledge base with query sets where all antecedents are the same. The RCTerIndexChecker outperforms the IBinCheck by reducing performance by more than a 90%. The average runtime for the IBinCheck is 11.26 milliseconds whereas the RCTerIndexChecker is 0,94 milliseconds. For a 10-rank knowledge base with query sets where all antecedents are unique there is a 146.25 millisecond difference in the performance of the RCTerIndexChecker at 197.7 milliseconds and the IBinCheck at 343.95 milliseconds. The same can be said for query sets where the antecedents are half repeated and half unique.

There is a significant difference in the performance of our RCTerIndexChecker in comparison to the IBinCheck implementation of the RationalClosure for a 50-rank knowledge base with query sets where all antecedents become consistent at rank 1. The difference in performance is 533.64 milliseconds with the average performance of the IBinCheck at 540.8 milliseconds and the RCTerIndexChecker at 7.16 milliseconds. The RCTerIndexChecker outperforms the IBinCheck for a 50-rank knowledge base with query sets where all antecedents are the same, as well as where all antecedents are unique, and half repeated and half unique. In the case of antecedents that are the same RCTerIndexChecker outperforms the IBinCheck by more than 90%. This suggests that the use of threads to check for entailment has a considerable computational speedup.

The performance of the RCTerIndexChecker in comparison to the naive implementation of the RationalClosure was significantly faster similar to the observation of the rank-10 knowledge base. The RCTerIndexChecker outperforms both the naive and IBinCheck implementation for the 50-rank knowledge base for all query sets.

Similar to the comparisons made for the 50-rank knowledge base, the RCTerIndexChecker outperforms the IBinCheck for a 100-rank knowledge base with query sets where all antecedents become consistent at rank 1 and rank 100 as well as antecedents that are all the same and unique sets. In the case of antecedents which are half repeated and half unique the IBinCheck outperforms both the RCTerIndexChecker and naive implementation. There is strong evidence to support that the RCTerIndexChecker outperforms the IBinCheck when the number of antecedents that are repeated is significantly high and the number of ranks is small however, when this is not the case the RCTerIndexChecker is still very much consistent in performance when compared to the IBinCheck as the difference in performance is insignificant.

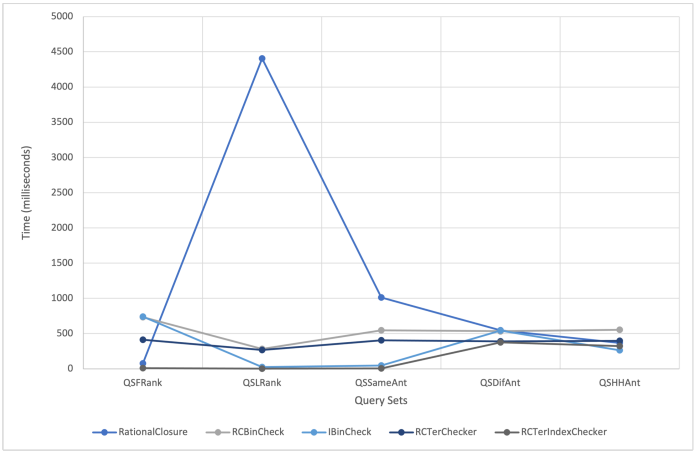

Figure 4: Graphs showing the comparison of all implementations' performance for 100-rank knowledge bases

The extent to which the RCTerIndexChecker outperforms the IBinCheck for all query sets tends to increase as the number of

ranks in the knowledge base decreases. However, in some cases the IBinCheck outperforms the RCTerIndexChecker for instances of half repeated and half unique antecedents. Overall, the performance of the RCTerIndexChecker was fairly uniform for each query set, which disproves the theory that the number of ranks in a knowledge base is a major factor in whether or not the optimization will perform better. Instead, it suggests that the variable of interest is the ranks at which the antecedents of our queries become consistent with the knowledge base.

7.2.2 Comparative analysis over varying distributions of statements. The complete set of average execution times obtained from all tests are presented in section 3 of Appendix A.

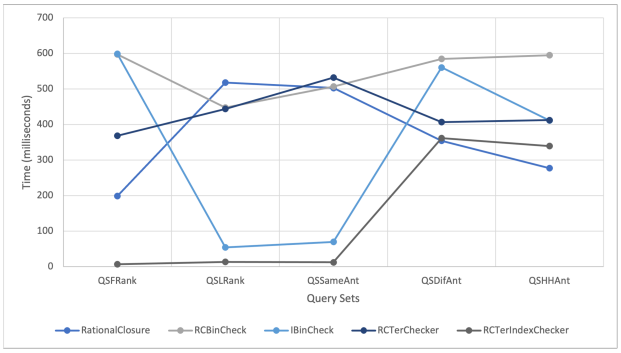

Figure 5: Graphs showing the comparison of all implementations' performance for uniform distributions

The RCTerIndexChecker performed faster than the IBinCheck for all distributions of statements. The naive implementation performed significantly better than the RCTerIndexChecker for the exponential and uniform distributions for query sets that contained unique antecedents, as well as antecedents that were half repeated and half unique. The RCTerIndexChecker performed significantly faster than the IBinCheck for all distributions for query sets where the antecedents become consistent with the knowledge base at rank-1 and the final rank, as well as for antecedents that were the same. In cases where the RCTerIndexChecker performed better than the IBinCheck, there was a difference of 120 - 180 milliseconds.

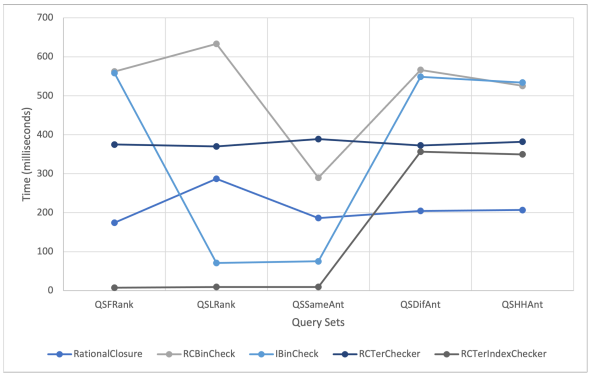

Figure 6: Graphs showing the comparison of all implementations' performance for exponential distributions

8 CONCLUSIONS

Our reasoners are fully functional console-applications, each with their own specifications to test scalability. Our first optimization allows the user to enter one query at a time whereas our second reasoner allows the user to enter multiple queries at a time. This allowed us to investigate the impact of scalability on computational speedup. The results of the tests indicate that the RCTerChecker performs significantly faster than the RCBinCheck when the rank at which the antecedent becomes consistent with the knowledge base is the final rank. We can deduce that because the ternary approach has two midpoint values instead of the single midpoint value of the binary search, it is able to find the rank more quickly due to the search range being split into three parts instead of two. That being said the ternary approach favours antecedents that become consistent at ranks with a higher number as opposed to the binary search.

It is evident that the RCTerIndexChecker is a more scalable approach as we observed that the implementation is able to query multiple sets at a time, and an increase in rank size leads to a consistent increase in performance above all other implementations. While there is a case or two where the IBinCheck outperforms the RCTerIndexChecker, the difference is negligible in comparison. This is due to the use of threads computing asynchronously and the ability of the implementation to store antecedents, as well as the rank at which it becomes consistent with knowledge base in a hash-table. This approach speeds up computation time by recovering the rank, instead of performing the calculation several times for the same antecedent.

9 FUTURE WORK

Future researchers can extend the following research by testing a range of sequential thresholds on computers with more than two cores for the RCTerIndexChecker to determine the optimum threshold. Our reasoners along with the previous years SCADR project are written in Java, thus future researchers should extend the system to other languages such as C or C++, which are platform independent and much faster than Java. In addition to this, there are currently very few efficient propositional tools. TweetyProject libraries were used in this investigation however, future researchers should consider developing or utilizing other tools to increase performance. Lastly, researchers should consider developing Bailey's knowledge base generator [1] to produce more complex statements as no modifications were made to the generator during testing.

REFERENCES

[1] Aiden Bailey. 2021. Scalable Defeasible Reasoning. (2021), 1–16. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/BLYAID001_SCADR.pdf

[2] Giovanni Casini, Thomas Meyer, Kodylan Moodley, and Ivan Varzinczak. 2013. Towards practical defeasible reasoning for description logics. (2013), 1–13. http://ceur-ws.org/Vol-1014/paper_17.pdf

[3] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2018. Defeasible Entailment: from Rational Closure to Lexicographic Closure and Beyond. *In 17th International Workshop on Non-Monotonic Reasoning (NMR)* (2018), 109–118. http://pubs.cs.uct.ac.za/id/eprint/1304

[4] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2019. Taking defeasible entailment beyond rational closure. *European Conference on Logics in Artificial Intelligence*, 182–197. https://doi.org/10.1007/978-3-030-19570-0

[5] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. *In Proceedings of the third annual ACM symposium on Theory of computing* (1971), 151–158. https://doi.org/10.1145/800157.805047

[6] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5 (1962), 394–397. Issue 7. https://doi.org/10.1145/368273.368557

[7] L. Giordano, V. Gliozzi, N. Olivetti, and G.L. Pozzato. 2015. Semantic characterization of rational closure: From propositional logic to description logics, Artificial Intelligence. 226 (2015), 1–33. https://doi.org/10.1016/j.artint.2015.05.001

[8] Joel Hamilton. 2021. An Investigation into the Scalability of Rational Closure. (2021), 6–7. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/SCADR_HMLJOE001.pdf

[9] Joel Hamilton, Daniel Park, and Aiden Bailey. 2021. Scalable Defeasible Reasoning. (2021), 1–10. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/SCADR_Project_Proposal.pdf

[10] Adam Kaliski. 2020. An Overview of KLM-Style Defeasible Entailment. (2020), 52–83. http://hdl.handle.net/11427/32743

[11] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial intelligence* 44 (1990), 167–207. https://doi.org/10.1016/0004-3702(90)90101-5

[12] Daniel Lehmann. 1995. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence* 15, 1 (1995), 61–82. https://doi.org/10.48550/arXiv.cs/0203002

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

[14] Joaão P. Marques-silva and Karem A. Sakallah. 1999. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48 (1999), 506–521. https://doi.org/10.1109/12.769433

[15] Drew McDermott and Jon Doyle. 1980. Non-monotonic logic I. *Artificial Intelligence* 13, 1 (1980), 41–72. https://doi.org/10.1016/0004-3702(80)90012-0 Special Issue on Non-Monotonic Logic.

[16] Ben-Ari Mordechai. 2012. *Propositional Logic: Formulas, Models, Tableaux* (3 ed.). Springer London, 1, 7–47. https://doi.org/10.1007/978-1-4471-4129-7

[17] Matthew Morris, Tala Ross, and Thomas Meyer. 2020. Algorithmic definitions for KLM-style defeasible disjunctive Datalog. *South African Computer Journal* 32, 2 (2020), 141–160. https://doi.org/10.18489/sacj.v32i2.846

[18] Joon Soo Park. 2021. Scalable Defeasible Reasoning. (2021), 1–12. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/PRKJOO001_SCADR_Final_Paper.pdf

[19] Dhiresh Thakor Vallabh and Evashna Pillay. 2022. Scalable Defeasible Reasoning V2 with focus on Rational and Lexicographic Closure. (2022).

[20] Matthias Thimm. 2014. Tweety - A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)* (Vienna, Austria).

[21] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. 2015. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proc. IEEE* 103 (2015), 2021–2035. https://doi.org/10.1109/JPROC.2015.2455034d

Appendix A

1 Comparison of Optimizations of Rational Closure Runtime

The following tables contain values pertaining to the average execution time (i.e., in milliseconds) of four optimization approaches, as well as the naive implementation of the `Rational Closure`. Each approach determined whether or not each query in the set was entailed by the knowledge base given. The optimization approaches used for comparison along with our new approaches are referred to as follows:

- `RC` is the naive implementation of the `Rational Closure`.
- `RCBinCheck` is the previous year's implementation with focus on the binary search.
- `IBinCheck` referred to as `StoredRankBinCheck` in last year's paper focuses on an indexing approach.
- `RCTerChecker` is our first implementation with focus on the ternary search.
- `RCTerIndexChecker` is our second implementation with focus on the ternary search and concurrency.

2 Comparison over varying ranks

In order to ensure that the results obtained for these tests were due to varying numbers of ranks and not due to varying distributions of statements over the ranks, the distribution of statements over the ranks in the knowledge bases used was uniform, with each rank containing the same number of statements.

2.1 Knowledge Base with 10 Ranks

This knowledge base contains 38 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 10. The following query sets were tested:

- `QSFRank` has 51 queries, all of which are the same (i.e., $1 \mid\sim 0$).
- `QSLRank` has 50 queries, all of which are the same (i.e., $10 \mid\sim 8$).
- `QSSameAnt` has 50 queries, all of which have the same antecedent.
- `QSDifAnt` has 20 queries, all of which are unique.
- `QSHHAnt` has 40 queries, half unique and half repeated.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	50.45	321.31	293.39	269,14	2,90
QSLRank	245.3	366.70	51.40	112,4	2,26
QSSameAnt	187.58	349.50	11.26	97,02	0,94
QSDifAnt	260.70	420.95	343.95	267,05	197,7
QSHHAnt	251.65	252.58	167.95	206,2	144,98

Table 1: Average execution times of Knowledge base with ranking of 10

2.2 Knowledge Base with 50 Ranks

This knowledge base contains 198 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 10. The following query sets were tested:

- `QSFRank` has 50 queries, all of which are the same (i.e., $1 \mid\sim 0$).
- `QSLRank` has 50 queries, all of which are the same (i.e., $50 \mid\sim 0$).
- `QSSameAnt` has 50 queries, all of which have the same antecedent.
- `QSDifAnt` has 50 queries, all of which are unique.
- `QSHHAnt` has 50 queries, half unique and half repeated.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	106.88	588.24	540.80	339,82	7,16
QSLRank	1208.66	435.24	45.84	210,42	8,06
QSSameAnt	1144.78	296.64	30.34	319,64	3,04
QSDifAnt	615.34	448.18	440.12	330,06	288,76
QSHHAnt	387.84	437	263.42	365,22	247,14

Table 2: Average execution times of Knowledge base with ranking of 50

2.3 Knowledge Base with 100 Ranks

This knowledge base contains 398 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 10. The following query sets were tested:

- `QSFRank` has 49 queries, all of which are the same (i.e., $1 \mid\sim 0$).
- `QSLRank` has 49 queries, all of which are the same (i.e., $200 \mid\sim 198$).
- `QSSameAnt` has 84 queries, all of which have the same antecedent.
- `QSDifAnt` has 50 queries, all of which are unique.
- `QSHHAnt` has 52 queries, half unique and half repeated antecedents.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	78.35	732.80	740.55	413,86	10,24
QSLRank	4406.43	283.80	23.57	266,94	4,04
QSSameAnt	1013.64	547.31	48.12	403,61	7,36
QSDifAnt	545.48	533.30	544.24	391,04	376,84
QSHHAnt	368.90	555.46	263.21	398,56	322,75

Table 3: Average execution times of Knowledge base with ranking of 100

3 Comparison varying distributions of statements

To ensure that our findings were as a result of varying distributions of statements over ranks and not the varying numbers of ranks, the number of ranks in the knowledge bases utilized for these tests was fixed at 50. The following query sets were tested:

- `QSFRank` has 50 queries, all of which are the same (i.e., $1 \mid\sim 0$).
- `QSLRank` has 50 queries, all of which are the same (i.e., $50 \mid\sim 0$).
- `QSSameAnt` has 50 queries, all of which have the same antecedent.
- `QSDifAnt` has 50 queries, all of which are unique.
- `QSHHAnt` has 52 queries, half unique and half repeated antecedents.

3.1 Knowledge Base with Uniform Distribution

This knowledge base contains 990 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 10.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	198.58	597.02	598.34	368,12	6,84
QSLRank	518.04	447.46	54.76	443,6	13,82
QSSameAnt	502.70	506.60	69.78	532,36	12,48
QSDifAnt	354.30	584.64	560.52	406,88	361,5
QSHHAnt	277.38	595.17	411.13	412,42	339,66

Table 4: Average execution times of Knowledge base with Uniform Distribution

3.2 Knowledge Base with Normal Distribution

This knowledge base contains 990 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 25.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	203.38	614	595.36	381,52	9,68
QSLRank	1210	261.52	63.96	278,78	11,94
QSSameAnt	1119.96	289.72	75.16	274,04	9,62
QSDifAnt	910.46	480.74	455.80	390,18	328,2
QSHHAnt	776.42	488.94	257.90	415,5	300,02

Table 5: Average execution times of Knowledge base with Normal Distribution

3.3 Knowledge Base with Exponential Distribution

This knowledge base contains 990 defeasible implications. The `RCTerIndexChecker` uses a sequential threshold set at 25.

QuerySets	RC	RCBinCheck	IBinCheck	RCTerChecker	RCTerIndexChecker
QSFRank	174.58	562.28	558.48	375,08	7,48
QSLRank	286.72	633.16	70.88	370,36	9,2
QSSameAnt	186,08	289.70	75.16	389,16	9,2
QSDifAnt	204.36	566.18	548.80	372,74	356,74
QSHHAnt	206,9	525.42	534.12	381,96	349,66

Table 6: Average execution times of Knowledge base with Exponential Distribution