UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2021

Title: An Investigation into the Scalability of Rational Closure

Author: Joel Hamilton

Project Abbreviation: SCADR

Supervisor(s): Professor Thomas Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	20
Experiment Design and Execution	*0*	*20*	10
System Development and Implementation	*0*	*20*	0
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**	

An Investigation into the Scalability of Rational Closure

Joel Hamilton
University of Cape Town
Cape Town, South Africa
HMLJOE001@myuct.ac.za

ABSTRACT

Knowledge representation and reasoning (KRR) is an approach to artificial intelligence (AI) in which a system has some information about the world represented formally (a knowledge base), and is able to reason about this information. Defeasible reasoning is a non-classical form of reasoning that enables systems to reason about knowledge bases which contain seemingly contradictory information, thus allowing for exceptions to assertions. Currently, systems which support defeasible entailment for propositional logic are ad hoc, are few and far between, and little to no work has been done on improving the scalability of defeasible reasoning algorithms. We investigate the scalability of Rational Closure, and propose optimisations thereof, as well as present a tool to perform defeasible entailment checks using these optimised algorithms.

CCS CONCEPTS

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

KEYWORDS

artificial intelligence, knowledge representation and reasoning, defeasible reasoning, satisfiability solving

1 INTRODUCTION

Artificial Intelligence (AI) has been around for many years, with its two core aspects being machine learning (ML) and knowledge representation and reasoning (KRR) [2]. Our research will focus on the latter. Knowledge representation refers to the notion of using some formal set of symbols or notation to represent information about the world. Reasoning refers to the idea of drawing inferences from this information, with the key idea for this research being the use of algorithms to reason about information (automated reasoning). We will utilise logics (a mechanism for formalising ways to reason [1]) to represent information.

Propositional logic [1] is monotonic, which means the addition of new information cannot contradict any previous conclusions you could draw. This is not a "common-sense" approach to reasoning [4].

Reasoning in such a way limits the ability to model human reasoning, as the property of monotonicity does not hold in the way that humans think.

For example, if a human knows that it rains every Saturday, and that today is a Saturday, then it makes sense to conclude that it is raining today, however if we are then told that it is not raining today, a human would interpret the addition of this new fact to mean that we have simply come across an exception to the notion of it raining every Saturday.

Reasoning according to propositional logic simply notes that a contradiction has occurred, meaning our knowledge can never all be true, thus any and all information can be inferred from what we know, rendering our knowledge useless. What may have allowed the additional information to not cause a contradiction is if the initial knowledge we had rather stated that “typically, it rains every Saturday”. We will use a framework for nonmonotonic reasoning in which statements of the form "typically, something is the case" are allowed. Sections 1 and 2 were done jointly with Park and Bailey.

2 BACKGROUND

2.1 Propositional Logic

Propositional Logic [1] is a framework for modelling information about the world, in which statements (known as *formulas*) are built up using *propositional atoms*, which are sentences which can be assigned a *truth value* (true or false), and *Boolean operators* ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$). Formulas can be defined recursively as either being simply an atom (e.g. p), or if α and β are formulas in $\mathcal{L}$ (the set of all formulas), then so are $\neg\alpha, \alpha \wedge \beta, \alpha \vee \beta, \alpha \rightarrow \beta,$ and $\alpha \leftrightarrow \beta$.

An *interpretation* is a function $I : \mathcal{P} \rightarrow \{T, F\}$ which attributes a single truth value to each propositional atom, and $\mathcal{W}$ is the set of all interpretations. The truth value of a formula α under a given interpretation I is written as $I(\alpha)$, and if $I(\alpha)$ is true for some formula α, then we say I satisfies α, written $I \Vdash \alpha$. A *knowledge base* is a finite set of formulas, and it is said that an interpretation I *satisfies* a knowledge base $\mathcal{K}$ if for every $\alpha \in \mathcal{K}, I \Vdash \alpha$. A knowledge base that is satisfied by at least one interpretation I is said to be *satisfiable*, and I is then a *model* of $\mathcal{K}$. The set of all models of a knowledge base $\mathcal{K}$ or of all models of a formula α are denoted $Mod(\mathcal{K})$ and $Mod(\alpha)$ respectively.

2.2 Entailment

If a formula α is true in every model of $\mathcal{K}$ (i.e., $Mod(\mathcal{K}) \subseteq Mod(\alpha)$), then we say $\mathcal{K}$ *entails* α (denoted $\mathcal{K} \models \alpha$). This form of propositional entailment uses classical reasoning, which is a framework for inferring information from given knowledge. Classical reasoning is monotonic, which means that the addition of new formulas to a knowledge base $\mathcal{K}$ should not contradict any previous inferences made from $\mathcal{K}$. This causes a problem when contradictory statements are added to a knowledge base, as it then means that there will be no models of $\mathcal{K}$, and thus any statement is true in every model of $\mathcal{K}$, so such a knowledge base is meaningless, as it entails everything. We therefore require a framework for nonmonotonic reasoning. Our preferred approach to nonmonotonic reasoning is the KLM approach [9, 12].

Checking whether or not $\mathcal{K} \models \alpha$ can be reduced to checking the satisfiability of the knowledge base with $\neg\alpha$ added to it ($\mathcal{K} \cup \{\neg\alpha\}$), where the satisfiability check can be done using a propositional logic satisfiability (SAT) solver, which is a program which, given a propositional logic formula, will return whether or not there exists an interpretation which satisfies the formula.

2.3 KLM Approach to Defeasible Reasoning

Defeasible reasoning is a form of reasoning which allows one to reason about knowledge bases which contain seemingly contradictory information, and allows for exceptions to general assertions. In the KLM Approach [9] [12], we have a way to model statements of the form $\alpha \mathrel{|}\!\sim \beta$, which is read as "$\alpha$ typically implies β", which simply means that if α is true, then this is typically enough information to believe that β is also true. A detailed overview of this approach is provided by Kaliski in [8].

The notion of defeasible entailment (denoted $\mathrel{|}\!\approx$) is not unique, i.e., there are many acceptable ways to infer information from a defeasible knowledge base (a knowledge base containing statements of the form $\alpha \mathrel{|}\!\sim \beta$). The KLM approach is our approach of choice, due to its properties which determine whether or not a defeasible entailment approach is accepted within the framework. These properties are known as the KLM properties. [9, 12]. Rational Closure [12] and Lexicographic Closure [11] are two approaches to defeasible reasoning which both fall within the KLM approach, and relevant closure [5] and ranked entailment [12] are two which do not [6]. This research focuses on Rational Closure [12] (RC).

2.4 Rational Closure

Rational closure is the most conservative form of defeasible entailment (i.e., it infers very little from a defeasible knowledge base), and can be defined both semantically and algorithmically, as laid out in [6].

The semantic definition is based on structures known as *ranked interpretations*, which are simply rankings of all interpretations in $\mathcal{W}$ (the set of all interpretations), in order of typicality. [6]

This paper focuses on the algorithmic definition of Rational Closure, which is split into two sub-algorithms, namely `BaseRank` and `RationalClosure`, as proposed in [6].

The `BaseRank` algorithm begins with the materialisation $\overrightarrow{\mathcal{K}}$ of the knowledge base $\mathcal{K}$ which is merely $\mathcal{K}$ which has had all its defeasible implications converted into classical implications. The algorithm ranks all statements in $\overrightarrow{\mathcal{K}}$ by how general they are. The algorithm produces a ranking such that statements which are always true (classical statements), are all in the bottom rank (the infinite rank), and the rest of the ranks contain materialisations of defeasible statements which become increasingly general as one moves up the ranking.

Algorithm 1: `BaseRank`

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

`RationalClosure` takes in a ranked knowledge base, and a defeasible implication statement (query) (e.g. $\alpha \mathrel{|}\!\sim \beta$), and returns whether or not the statement is defeasibly entailed by the ranked knowledge base. The algorithm removes ranks one by one until it finds that the antecedent of the implication statement (in this case, α), is consistent with the knowledge base, and then returns true if these remaining statements entail the materialisation of the original query (in this case, $\alpha \to \beta$), and false otherwise. Note that a formula is consistent with a knowledge base $\mathcal{K}$ when its negation is not entailed by $\mathcal{K}$.

Algorithm 2: `RationalClosure`

```
Input: A knowledge base $\mathcal{K}$ and a Defeasible Implication
       $\alpha \mathrel{|}\!\sim \beta$
Output: true, if $\mathcal{K} \mathrel{|}\!\approx \alpha \mathrel{|}\!\sim \beta$, 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
    $R := R \setminus R_i$;
    $i := i + 1$;
return $R_\infty \cup R \models \alpha \to \beta$;
```

The following is an example which illustrates how the Rational Closure algorithm works.

Consider a knowledge base containing the information that "birds typically fly", "penguins are birds", "birds typically have wings", and "penguins typically don't fly", which has been formalised as follows:

$$\mathcal{K} = \{b \mathrel{|}\!\sim f, p \to b, b \mathrel{|}\!\sim w, p \mathrel{|}\!\sim \neg f\}$$

and then ranked according to the Base Ranks algorithm, producing the following ranking:

Rank 0	$b \to f, b \to w$
Rank 1	$p \to \neg f$
Rank ∞	$p \to b$

Figure 1: Base Ranking of Knowledge base $\mathcal{K}$

Given that "penguins are birds", it follows that birds are more general concepts than penguins, and thus statements with birds as its antecedent are higher up in the ranking than those with penguins as its antecedent.

Consider that we want to investigate whether or not the statement $p \mid\!\sim w$ is entailed by the knowledge base, then:

(1) We check if $\neg p$ is entailed by $\overrightarrow{\mathcal{K}}$. It is, so we throw away the top rank.

Rank 0	~~$b \to f, b \to w$~~
Rank 1	$p \to \neg f$
Rank ∞	$p \to b$

Figure 2: Removal of rank 0

(2) We then check if $\neg p$ is entailed by the remaining statements in ranks 1 and ∞. It is not, so we check whether the statement $p \to w$ is entailed by the remaining statements. It is not, so we conclude $\mathcal{K} \not\approx p \mid\!\sim w$.

2.5 Defeasible Reasoning Implementations

There exist some implementations of defeasible reasoning for both propositional and first-order logic, as discussed in [3], however none of these implementations are aligned with the KLM approach [9, 12], and thus no implementatons of KLM-style defeasible reasoning exist in the propositional case.

Some work has been done which looks at implementing the ability to reason defeasibly in the context of description logics, such as in [13], but the lack of a similar implementation in the propositional case appears as a large gap in the literature regarding nonmonotonic reasoning, as does the lack of research into the scalability of established LM-rational defeasible entailment algorithms such as Rational Closure [12] in the propositional case.

3 PROJECT AIMS

The key aims of this project were:

- To build a prototype defeasible reasoner which integrates with an existing classical reasoner to reason about defeasible knowledge bases according to rational closure.
- To improve the scalability of such a rational closure implementation through the use of various optimisation approaches.
- To obtain empirical results which help inform in which contexts (i.e. for which knowledge bases, and for which queries) our optimisations are effective in obtaining better performance than the existing Rational Closure algorithm.

4 IMPLEMENTATION & OPTIMISATION OF RATIONAL CLOSURE

4.1 Naive Implementation

We developed a defeasible reasoner which returns whether or not a defeasible implication statement (i.e. a statement of the form "$\alpha \mid\!\sim \beta$") is entailed by a specific defeasible knowledge base. This program is "naive", due to its implementation of the `RationalClosure` algorithm being implemented exactly as laid out in Algorithm 2 in this paper, as per [6]. We also created a user-friendly version for the purposes of future research which shows the ranking of statements before any queries are made, as well as shows the steps made by each algorithm when computing a query. The naive implementation, along with those described in the following three subsections, were all accepted as being correct, as they all produced correct results when tested with cases which correspond to the KLM Properties. [12] This is strong evidence that our implementations are all *LM-rational*, and given their results were all exactly as expected according to both the semantic and algorithmic definitions of Rational Closure [6], we are confident in the correctness of these defeasible reasoners.

All of these reasoners were developed using Java, and all made extensive use of the *TweetyProject* library, version 1.20 [14, 15].

We utilise a built-in classical reasoner tool built into the *TweetyProject* library, which uses a solver called Sat4j. [10]

4.2 Optimisation 1: Binary Search Entailment Check

We developed a second reasoner, which uses an amended version of `RationalClosure`, which we propose here, as `RCBinCheck`. This algorithm works by using a binary search algorithm to find the rank from which all ranks need to be removed, as opposed to iterating linearly from the top, downwards, as in `RationalClosure`.

When checking whether or not a statement $\alpha \mid\!\sim \beta$ is defeasibly entailed by the already-ranked knowledge base, the binary search would would replace the linear search for the rank where α becomes consistent with the knowledge base. This works as follows:

(1) Start with a min and a max value, representing the smallest and largest possible rank which could be the rank we are looking for. (If this is the first time performing step 1, then set min as zero, and max as the number of ranks n).
(2) Choose the midpoint of *min* and *max*, let's call this rank k.
(3) Check if removing rank k and all those above it results in α being consistent with the knowledge base.
    - If yes, it would proceed to the next step.
    - If no, it would go back to the first step, but instead pass in k as *min*, and the keep *max* the same, as we'd know that the rank we are searching for (i.e. the rank where α becomes consistent with the knowledge base) is in the 'lower' half of our ranks, (i.e. a higher rank number), as with all ranks up to and including rank k removed, α is still not consistent with the knowledge base, and thus we need to remove more ranks.
(4) Check if adding rank k back in results in α still being consistent with the knowledge base.

- If yes, then go back to step 1 but pass in *max* as k, and keep *min* the same, as we'd know that the rank we re searching for is in the 'upper' half of our ranks (i.e. a lower rank number), as with all ranks up to and including rank k removed, α is consistent with the knowledge base, and thus we need to remove fewer ranks.
- If no, then rank k is the rank from which we need to remove all ranks, including rank k.

(5) Return true if $\alpha \rightarrow \beta$ is entailed by the ranked knowledge base with all ranks up to and including k removed, and false otherwise.

Given that the efficacy of this optimisation is reliant on it not requiring linear iteration through ranks to find which ranks to remove when performing the entailment check, we hypothesised that there would be a certain number of ranks (which will be determined by further research) for which this optimisation would perform better (i.e. faster) than the naive implementation, and that the extent to which this optimisation performed better would increase as the number of ranks increase.

However, we also believed that this improved performance provided by the `RCBinCheck` implementation would only be present for defeasible queries $\alpha \mid\!\sim \beta$ where α becomes consistent at a rank larger than approximately $log(n)$ where n is the number of ranks in the knowledge base. This is because a binary search for the rank would perform $O(log\ n)$ consistency checks in both the average and best cases, and the linear search performs $O(n)$ consistency checks in the worst case, but $O(1)$ consistency checks in the best case, and thus if the rank at which α becomes consistent is less than $log(n)$, then the `RationalClosure` implementation is likely to find this rank quicker than the `RCBinCheck` implementation.

Note that although the efficiency, in terms of number of consistency checks, can be improved, each consistency check still reduces to the Boolean satisfiability problem, which is NP-complete (proof available in [7]), however this algorithm still ought to yield improvements in the best case, and possibly the average case. We have formalised this algorithm as follows.

Algorithm 3: `RCBinCheck`

```
Input: A knowledge base $\mathcal{K}$, and a Defeasible Implication
       $\alpha \mid\!\sim \beta$
Output: true, if $\mathcal{K} \approx \alpha \mid\!\sim \beta$, and false, otherwise
( $R_0, ..., R_{n-1}, R_\infty, n$ ) := BaseRank($\mathcal{K}$);
$low := 0$;
$high := n$;
while $high > low$ do
    $mid = (low + (high - low)/2)$;
    $R := \bigcup_{i=mid+1}^{j<n} R_j$;
    if $R_\infty \cup R \models \neg\alpha$ then
        $low := mid + 1$;
    else
        $R := \bigcup_{i=mid}^{j<n} R_j$;
        if $R_\infty \cup R \models \neg\alpha$ then
            $R := \bigcup_{i=mid+1}^{j<n} R_j$;
            return $R_\infty \cup R \models \alpha \rightarrow \beta$;
        else
            $high := mid$
return true
```

4.3 Optimisation 2: Indexing Approach

We developed a third reasoner, which uses another amended version of the entailment check from the original algorithm, which we propose here, as `StoredRankCheck`. This algorithm works by storing a hashtable of antecedents along with the rank number at which they become consistent with the knowledge base. This means that if one is querying a knowledge base with multiple defeasible queries, then those which have the same antecedent will only require this computation (i.e. finding the rank at which the antecedent becomes consistent with the knowledge base) to be run once. This optimisation is more memory-intensive (as for knowledge bases with large numbers of ranks, our hashtable can grow quite large).

Consider a ranked knowledge base where a formula α becomes consistent with the knowledge base at rank 20. If the queries we wish to query this knowledge base are as follows:

- $\alpha \mid\!\sim \beta$
- $\alpha \mid\!\sim \gamma$
- $\alpha \mid\!\sim \phi$
- $\alpha \mid\!\sim \delta$

Then the calculation of rank 20 as being the rank from which we need to remove statements would only need to be done once, as the pair $(\alpha, 20)$ would be stored in our hash table, and when computing the remaining queries, we could simply retrieve this value from our hash table as opposed to recalculating the value of 20 for each query.

We hypothesised that this implementation would outperform the naive implementation for query sets in which a large proportion of queries have the same antecedents (the exact proportion of repeated antecedents at which this improved performance starts to occur will need to be finetuned by further research). This is because the computation of which ranks to remove would not need to be repeated for statements which have the same antecedents. We predicted that for query sets whose queries all use different antecedents, there would be no notable difference in terms of execution between the StoredRankCheck and the RationalClosure implementations, as the same number of rank decision computations will be performed by both implementations. We have formalised this algorithm as follows.

```
Algorithm 4: StoredRankCheck
  Input: A knowledge base $\mathcal{K}$ and a set of Defeasible
         Implications $S$
  Output: An array containing responses for each query,
         true, if $\mathcal{K} \mathrel{|\!\approx} \alpha \mathrel{|\!\sim} \beta$, and false, otherwise
antecedentRanks := empty hashtable;
arrayOfResults := $\emptyset$;
( $R_0, ..., R_{n-1}, R_\infty, n$) := BaseRank($\mathcal{K}$);
for each $\alpha \mathrel{|\!\sim} \beta \in S$ do
    if antecedentRanks contains $\alpha$ then
        i := antecedentRanks.get($\alpha$);
        $R := \bigcup_{i}^{j<n} R_j$;
    else
        $i := 0$;
        $R := \bigcup_{i=0}^{j<n} R_j$;
        while $R_\infty \cup R \models \neg\alpha$ and $R \neq \emptyset$ do
            $R := R \setminus R_i$;
            $i := i + 1$;
        antecedentRanks.put($\alpha$, i);
    add ($R_\infty \cup R \models \alpha \rightarrow \beta$) to arrayOfResults;
return arrayOfResults
```

4.4 Optimisation 3: Binary Indexing Approach

The fourth reasoner we developed uses an amended version of StoredRankCheck, which instead uses the key aspect of RCBinCheck (namely, the binary search algorithm) to compute the ranks before storing them, as opposed to the linear search used in the original algorithm (RationalClosure), and in StoredRankCheck. We propose this algorithm here, as StoredRankBinCheck.

We hypothesised that this implementation would outperform the naive implementation for query sets which contain a large proportion of statements which use the same antecedents, and that the extent to which this implementation outperformed RationalClosure would increase as the number of unique antecedents in the queries in a given query set decreases. We also predicted that the implementation of StoredRankBinCheck would outperform the naive implementation to an increasingly larger degree as the number of ranks in the knowledge base grows, due to the binary search aspect of this algorithm resulting in finding which ranks to remove significantly faster. We have formalised this algorithm as follows.

```
Algorithm 5: StoredRankBinCheck
  Input: A knowledge base $\mathcal{K}$ and a set of Defeasible
         Implications $S$
  Output: An array containing responses for each query,
         true, if $\mathcal{K} \mathrel{|\!\approx} \alpha \mathrel{|\!\sim} \beta$, and false, otherwise
arrayOfResults := $\emptyset$;
( $R_0, ..., R_{n-1}, R_\infty, n$) := BaseRank($\mathcal{K}$);
for each $\alpha \mathrel{|\!\sim} \beta \in S$ do
    if antecedentRanks contains $\alpha$ then
        i := antecedentRanks.get($\alpha$);
        $R := \bigcup_{i}^{j<n} R_j$;
        add ($R_\infty \cup R \models \alpha \rightarrow \beta$) to arrayOfResults;
        continue;
    else
        $low := 0$;
        $high := n$;
        while $high > low$ do
            $mid = (low + (high - low)/2)$;
            $R := \bigcup_{i=mid+1}^{j<n} R_j$;
            if $R_\infty \cup R \models \neg\alpha$ then
                $low := mid + 1$;
            else
                $R := \bigcup_{i=mid}^{j<n} R_j$;
                if $R_\infty \cup R \models \neg\alpha$ then
                    $R := \bigcup_{i=mid+1}^{j<n} R_j$;
                    add ($R_\infty \cup R \models \alpha \rightarrow \beta$) to
                      arrayOfResults;
                    break;
                else
                    $high := mid$
        add true to arrayOfResults;
return arrayOfResults
```

5 EXPERIMENT DESIGN

In order to determine how successful these optimisations were, we compared their execution time against one another using predetermined knowledge bases and query sets (containing defeasible implications) which we then passed into our different reasoners. We then timed the execution time taken by each reasoner, using built-in Java system time calls.

5.1 Creating Test Cases

5.1.1 Knowledge Bases. We used a knowledge-base generator (created by Bailey) to generate test knowledge bases which would be queried using our query sets. All our knowledge bases consisted of only defeasible statements, all of the form $a \mathrel{|\!\sim} b$, where a and b are propositional atoms. Our knowledge bases differed only in terms of the following variables:

(1) *Number of Ranks*: The number of ranks in the knowledge base when ranked according to BaseRank.

(2) *Statement Distribution:* The distribution of statements over the ranks in the knowledge base when ranked according to BaseRank.

These are the variables over which we are comparing our optimised implementations.

Thus, for *Number of Ranks*, we generated knowledge bases which had varying numbers of ranks (10, 50, and 100), but which each had the same number of statements in each rank (2), and thus each had the same distribution of statements over the ranks (uniform).

For *Statement Distribution*, we generated knowledge bases which had different distributions of statements over the ranks (uniform, normal, exponential), but the same number of ranks (50).

5.1.2 Query Sets. Our query sets were created manually, as each query set was required to have some characteristic which would allow us to gain insight into in which contexts our optimised algorithms are most effective. For each knowledge base, we created 5 query sets:

(1) A set of queries whose antecedents all differ from one another.
(2) A set of queries which all have the same antecedent.
(3) A set of queries whose antecedents are half unique, and are half repeated.
(4) A set of queries whose antecedents all become consistent with the knowledge base after all but the final rank have been removed.
(5) A set of queries whose antecedents all become consistent with the knowledge base after the first rank (rank 0) has been removed.

The query sets whose number of unique antecedents are controlled are to provide insight into how effective the `StoredRankCheck` implementation is in particular at improving the execution time from that of the `RationalClosure` implementation, as the idea behind this optimisation relies on there being a large number of repeated antecedents, and thus a large amount of unnecessary computation no longer being done.

The query sets where the level at which their antecedents become consistent with the knowledge base are controlled are to provide insight into the effectiveness of the `RCBinCheck` implementation in particular, as the idea behind this optimisation relies on the index of the ranks from which all above ranks must be removed being as large as possible.

All query sets should provide insight into the extent to which the `StoredRankBinCheck` algorithm is effective in optimising the execution of a Rational Closure entailment check, as the basis of this optimisation is that it will be more effective for higher ranks at which antecedents become consistent, as well as for query sets which contain a large number of repeated antecedents.

5.2 Running Experiments

We created a program which, for a given knowledge base $\mathcal{K}$, instantiates four reasoners, which all use the same ranking algorithm (BaseRank), but which each use a different entailment checking algorithm (RationalClosure, RCBinCheck, StoredRankCheck, and StoredRankBinCheck). This program then iterates through a given query set, and measures the time taken (in milliseconds) for each reasoner to determine whether or not $\mathcal{K} \approx \alpha$. The program then writes these measurements to a CSV file, where each line represents a particular query, and each individual value is the number of milliseconds taken for each reasoner to respond to the given query. Note that the implementation of this testing program was written in such a way that only the entailment check was timed, and not the ranking of the statements, as there was no change made to this aspect of the algorithms, and thus comparing the time taken to rank the statements are not of any interest to us, and any variation would be random.

These tests were run on a laptop with a Intel Core i5 processor, 2 cores, 4 logical processors, and 8GB ram.

Source code for rerunning experiments & reproducing results are in the project repository at www.github.com/joelvhamilton/rational-closure.

6 RESULTS & DISCUSSION

For each knowledge base and query set combination, we performed a Wilcoxon signed rank test to compare the runtimes of each different implementation with that of the naive implementation, and obtained a p-value for the null hypothesis that there is no statistically significant difference between the means of the two groups, and the alternative hypothesis that the two group means do in fact differ significantly. The full set of p-values obtained from all these tests can be seen in section 1 of Appendix A.

6.1 Binary Search Optimisation Results

The query sets whose results are most relevant to evaluating the performance of the `RCBinCheck` optimisation over varying numbers of ranks are those where the ranks at which the queries' antecedents become consistent with the knowledge base have been controlled.

6.1.1 Comparison over Varying Numbers of Ranks. The comparison between the runtimes of the `RCBinCheck` and `RationalClosure` implementations yielded statistically significant p-values ($\alpha = 0.05$) for a 100-rank knowledge base for the query set where all antecedents are rank-1 consistent, as well as for where all antecedents are rank-100 consistent. An exploration of the data showed that the mean of the runtimes for the rank-1 consistent entailment check using the `RCBinCheck` implementation were significantly worse than that of the `RationalClosure` implementation, and that for the rank-100 consistent entailment check, the `RCBinCheck` implementation performed significantly better than its naive counterpart. This strongly supports the idea that this optimisation is more effective when the ranks at which the antecedents of the queries become consistent with the knowledge base are larger (i.e. a higher rank number).

This same pattern is observed for the knowledge bases with 50- and 10 ranks, as the `RCBinCheck` optimisation performs significantly better for query sets where the antecedents of the queries become consistent at ranks 50 and 10, respectively, and significantly worse for queries with rank-1 consistent antecedents. The extent to which the `RCBinCheck` implementation outperforms the `RationalClosure` implementation for queries with antecedents which are final-rank-consistent tends to decrease as the number of ranks in the knowledge base decreases, however some exploration of the data revealed that the performance of the RCBinCheck implementation was relatively consistent, which is in fact evidence against the number of ranks in a knowledge base being a key determinant of whether or not the RCBinCheck optimisation will perform better, and implies that the variable of interest is in fact the ranks at which the antecedents of our queries become consistent, as, of course, the rank number of the final rank changes for knowledge bases with less ranks. This variable being key to evaluating our optimisation's efficacy is perfectly logical, as it determines how many ranks the program needs to iterate over, and thus, with varying numbers of ranks in the knowledge base, if the rank at which the antecedent became consistent with the knowledge base were to remain the same, the performance of the RCBinCheck would not change.

It may be the case that on average, knowledge bases with more ranks will be queried with queries whose antecedents become consistent at higher ranks, however this will vary from case to case, and is not certain enough to be able to argue that this optimisation will always be effective for knowledge bases containing more ranks.

6.1.2 Comparison over Varying Distributions of Statements over Ranks. Over all distributions (Normal, Uniform, and Exponential), we obtained extremely small p-values (all ≈ 0) when querying with the rank-1 consistent query sets, and with the rank-50 query sets. Once again, the RationalClosure implementation outperformed the RCBinCheck implementation significantly for the rank-1 consistent query sets, and the performance was the other way around for the rank-50 consistent query sets. This was unsurprising, as while the structure of the knowledge base was changing, the ranks at which our queries' antecedents became consistent remained unchanged, and thus the change in number of ranks in our knowledge base did not yield any difference in the performance of this optimisation. Once again, the RationalClosure implementation drastically outperformed the RCBinCheck implementation for the rank-1 consistent query set, and vice-versa for the rank-50 query set. This is very strong evidence that RCBinCheck is a significant improvement on the original RationalClosure algorithm for queries whose antecedents become consistent at a very high rank number. We foresee that future research will be able to refine at which rank this optimisation becomes effective, and that this information will be able to inform choice of implementation when a researcher (or user) has a large number of queries to query a knowledge base with.

Similarly, it could be the case that on average, knowledge bases with specific distributions of statements over ranks will be queried with queries whose antecedents become consistent at higher ranks, however this will vary from case to case, and is not certain enough to be able to argue that this optimisation will always be effective for knowledge bases containing more ranks.

6.2 Indexing Optimisation Results

The query sets whose results are most relevant for evaluating the performance of the StoredRankCheck implementation are those where the number of distinct antecedents is controlled.

6.2.1 Comparison over Varying Numbers of Ranks. The comparison between StoredRankCheck and RationalClosure (i.e. the comparison between the execution times of their implementations) yielded statistically significant p-values ($\alpha = 0.05$) for the query sets where half of the antecedents were distinct, and where all queries had the same antecedent (both < 0.05), across all the knowledge bases (i.e. with 10, 50, and 100 ranks, respectively). In both cases, the StoredRankCheck implementation significantly outperformed the RationalClosure implementation. This was slightly less significant for the 10-rank knowledge base, which suggests that the extent to which this optimisation performs better than the RationalClosure implementation would increase for cases with larger knowledge bases. This is however slightly misleading, as once again, if the ranks at which the antecedents become consistent were to be kept constant, then on average, the number of ranks in the knowledge base would not significantly impact the performance of either of our implementations, as the number of ranks over which our program would need to iterate (either repeatedly, or before storing this rank number, depending on the implementation) would be the same, and thus we believe the improvement in performance for knowledge bases with more ranks is merely a product of having queries which only become consistent in lower ranks.

A narrowly significant p-value was only obtained for one of the three cases where the query set contained all distinct antecedents, which we attribute to random variation, as when a query set has all distinct antecedents, the execution of RationalClosure and the StoredRankCheck implementation are essentially the same (no execution time is saved). These results strongly support the hypothesis that the StoredRankCheck implementation outperforms the naive implementation for query sets where a large proportion of the antecedents in the query set are repeated. The impressive performance of this optimisation for the query set with half-distinct antecedents strongly suggests that the proportion of repeated antecedents at which this optimisation begins to outperform the naive implementation is somewhere below 50%. We foresee that future research will refine the proportion of repeated statements at which this optimisation begins to outperform the naive implementation. It is important to note that this proportion would vary depending on the ranks at which the antecedents in the query sets become consistent with the knowledge base, as more execution time is saved when the work being replaced by the hash table call is more lengthy (i.e. iterations over more ranks).

6.2.2 Comparison over Varying Distributions of Statements over Ranks. For all distributions of statements over the knowledge bases (Uniform, Normal, and Exponential), the comparison between the StoredRankCheck and RationalClosure implementations yielded a similar pattern, whereby the p-value obtained for the query sets with all the same antecedent was lower than the p-value obtained for the query sets with half-distinct antecedents, which itself was consistently much lower than the p-value obtained for the query sets with all-distinct antecedents. This is very strong evidence in favour of the notion that the distribution of statements does not directly impact the performance of this implementation, which makes perfect sense, as the more specific variable (which could be impacted by the distribution, however we cannot say this with certainty) which has a more direct impact on the performance of the StoredRankCheck and RationalClosure implementations is the rank number of the rank at which the antecedents of our queries become consistent with the knowledge base.

6.3 Binary Indexing Optimisation Results

6.3.1 Comparison over Varying Numbers of Ranks. The comparison between the StoredRankBinCheck and RationalClosure implementations yielded statistically significant results ($p < 0.05$) for the 100 rank knowledge base when queried with all query sets, except for the one containing statements with all distinct antecedents. For the 50 and 10 rank knowledge bases, the query sets with half-distinct (i.e. all repeated) antecedents also yielded relatively high p-values. This can be explained by noting that the rank index storing aspect of the StoredRankBinCheck algorithm is likely to only yield a significant improvement over RationalClosure for cases where the query sets contain a large proportion of repeated antecedents. It appears that there are enough distinct antecedents in the half-distinct query set that the rank storing does not result in improved performance for the 10- and 50-rank knowledge bases, and that the antecedents in these query sets become consistent with the knowledge base at a low enough rank number that the binary search aspect of the algorithm did not yield a significant improvement in execution time when using the StoredRankBinCheck implementation.

The remaining 12 trials when comparing over varying numbers of ranks all yielded very low p-values (≈ 0), and the same pattern from the binary search optimisation was observed, where the optimised version (in this case, the StoredRankBinCheck optimisation) performed significantly worse for the rank-1 consistent antecedents, and performed significantly better for the final-rank consistent antecedents. This is unsurprising, as all rank-1 consistent antecedents result in best-case execution for the RationalClosure implementation, and final-rank consistent antecedents result in worst-case execution for RationalClosure, and thus it makes sense that our optimisation would result in better performance for final-rank consistent antecedents. We foresee that future research can determine at what rank (relative to the number of ranks n) this optimisation begins to perform better. Similar to the indexing optimisation comparison, for all numbers of ranks, the performance for the query sets containing queries with the same antecedent was very impressive (all p-values ≈ 0), which is strong evidence in favour of the idea that as the proportion of repeated antecedents grows, the performance of the optimisation improves.

6.3.2 Comparison over Varying Distributions of Statements over Ranks. Changing the distributions had no discernible impact on the results. The results over all distributions are very similar, and differ from one another similarly to how those results differ where the distributions are kept constant. This suggests that it is once again the rank number at which the antecedents of our queries become consistent with the knowledge base, along with the proportion of repeated antecedents in our query set, that determine the extent to which the StoredRankBinCheck implementation outperforms the naive implementation.

7 CONCLUSIONS AND IMPLICATIONS FOR DEFEASIBLE REASONING

Our findings strongly suggest that query set characteristics are a stronger determinant than knowledge base characteristics when figuring out in which contexts our different optimisations result in improved performance. The first key aspect of query sets to mention is that if there is a large number of repeated antecedents in our set of queries, then StoredRankCheck and StoredRankBinCheck will both tend to outperform the naive RationalClosure implementation, as every time these optimised algorithms encounter an antecedent it has come across before, it saves time by not performing the same calculation of which ranks to remove repeatedly.

The second key characteristic of query sets which determine the extent to which our optimisations provide improved efficiency over the original algorithm (as laid out by Casini et al in [6]) is the average rank number at which the antecedents of our queries become consistent with the knowledge base. If this number is 1, then RationalClosure will tend to obtain a result quicker than the RCBinCheck, and if this number is equal to the final rank in the knowledge base, then if the knowledge base contains a fair number of ranks (at least 10, based on this research), then RCBinCheck will outperform the original algorithm tremendously. The overall impression is that the larger the rank number at which our queries' antecedents become consistent with the knowledge base, the greater the extent to which RCBinCheck will outperform RationalClosure.

Based on how BaseRank works, statements whose antecedents are consistent with the knowledge base in low number ranks are often about more general concepts, e.g. a bird would likely be represented as a more general (i.e. less specific) concept than a penguin, and thus the generality of the defeasible implications being checked, more specifically the generality of the antecedents, could play large a role in determining how effective the RCBinCheck and StoredRankBinCheck optimisations are.

8 FUTURE WORK

As mentioned throughout the paper, there are key aspects of this research which we believe future researchers will be able to refine. We foresee that these refinements should be done in a similar way, but where the rank number at which queries become consistent with the knowledge base is treated as the key explanatory variable, and where knowledge base structure is kept the same. We believe such research would determine to a greater degree of certainty what proportion of repeated antecedents in a query set results in better performance by the StoredRankCheck and StoredRankBinCheck algorithms. We also foresee that future researchers will be able to determine at which rank (relative to the number of ranks in a knowledge base) individual queries are handled quicker by the StoredRankBinCheck and RCBinCheck implementations.

The key limitation of this research was the inability to isolate the experiments from the variation of other factors such as CPU usage, RAM usage, and other volatile aspects of the systems/architectures on which the experiments were run. We'd therefore suggest that future researchers use a high-performance computer in a more isolated environment in order to obtain less noisy data, and thus more reliable results.

The knowledge base and query set characteristics investigated in this paper are merely the tip of the iceberg as it relates to factors which determine the efficiency of optimisations, and therefore a large amount of similar research can be done controlling other variables, such as formula structure, given that all our knowledge bases contained statements of the same form, and that our implementations can handle much more complex propositional defeasible formulas.

REFERENCES

[1] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012.

[2] Zied Bouraoui, Antoine Cornuéjols, Thierry Denœux, Sebastien Destercke, Didier Dubois, Romain Guillaume, João Marques-Silva, Jérôme Mengin, Henri Prade, Steven Schockaert, Mathieu Serrurier, and Christel Vrain. From shallow to deep interactions between knowledge representation, reasoning and machine learning (kay r. amel group), 12 2019.

[3] Daniel Bryant and Paul Krause. A review of current defeasible reasoning implementations. *The Knowledge Engineering Review*, 23(3):227–260, 2008.

[4] G. Casini, T. Meyer, K. Moodley, and I. Varzinczak. Towards practical defeasible reasoning for description logics. Jul 2013.

[5] Giovanni Casini, Thomas Meyer, Kodylan Moodley, and Riku Nortjé. Relevant closure: A new form of defeasible reasoning for description logics. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, pages 92–106, Cham, 2014. Springer International Publishing.

[6] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. Taking defeasible entailment beyond rational closure. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, volume 11468 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2019.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[8] Adam Kaliski. An overview of klm-style defeasible entailment, 2020_.

[9] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial intelligence*, 44(1-2):167–207, 1990.

[10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7:59–6, 01 2010.

[11] Daniel Lehmann. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence*, 15(1):61–82, Mar 1995.

[12] Daniel Lehmann and Menachem Magidor. What does a conditional knowledge base entail? *Artificial intelligence*, 55(1):1–60, 1992.

[13] Kody Moodley, Thomas Meyer, and Ivan Varzinczak. A protégé plug-in for defeasible reasoning. volume 846, 06 2012.

[14] Matthias Thimm. 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)*, July 2014.

[15] Matthias Thimm. The tweety library collection for logical aspects of artificial intelligence and knowledge representation. *Künstliche Intelligenz*, 31(1):93–97, March 2017.

Appendix A

1 Rational Closure Implementation Runtime Comparison

The values in the tables are the p-values obtained when performing a Wilcoxon Signed Rank Test (with continuity correction) on the time taken by each knowledge base to determine whether or not each query in a given set of queries are entailed by the knowledge base. The sets of queries used in each case all contain defeasible implications whose antecedents can be described by the description in the left column.

- 'All Unique' simply refers to a set of queries whose antecedents all differ from one another.
- 'All the same' refers to a set of queries which all have the same antecedent.
- 'All repeated once' refers to a set of queries which have k/2 distinct antecedents, each repeated once, where k is the number of queries in the set.
- 'Rank n-consistent' refers to a set of queries whose antecedents all become consistent with the knowledge base with all ranks up to and including rank n-1 having been removed when performing the Rational Closure entailment check.

The phrases in the top row of the results tables indicate the Rational Closure algorithm implementation being compared, i.e.:

- 'Naive' refers to the `RationalClosure` implementation, with no optimisation approaches implemented.
- 'Binary' refers to the `RCBinCheck` implementation.
- 'Indexing' refers to the `StoredRankCheck` implementation.
- 'Binary Indexing' refers to the `StoredRankBinCheck` implementation.

1.1 Comparison over varying numbers of ranks

For these tests, the distribution of statements over the ranks in the knowledge bases used were consistently "uniform", with each rank containing only 2 statements, in order to ensure that the results obtained were a result of varying numbers of ranks and not due to varying numbers of statements within ranks or due to varying distributions of statements over the ranks.

1.1.1 Knowledge Base with 100 ranks

Query Set	Indexing vs Naive	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.141	0.8507	0.9769
All the same	1.805e-15	4.893e-14	1.742e-15
All repeated once	0.0004468	0.0008448	0.0354
Rank 1-consistent	3.8e-0.7	1.144e-09	1.145e-09
Rank 100-consistent	7.105e-15	1.145e-09	3.553e-15

1.1.2 Knowledge Base with 50 ranks

Query Set	Indexing vs Naive	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.002597	0.5495	0.4931
All the same	7.789e-10	7.789e-10	7.789e-10
All repeated once	0.0005995	0.004143	0.1449
Rank 1-consistent	7.161e-08	7.780e-10	7.783e-10
Rank 50-consistent	2.039e-09	1.038e-08	7.79e-10

1.1.3 Knowledge Base with 10 ranks

Query Set	Indexing vs Naïve	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.1231	0.165	0.6477
All the same	4.371e-09	6.248e-06	2.096e-09
All repeated once	0.003174	0.8472	0.09556
Rank 1-consistent	2.162e-0.6	1.188e-07	1.455e-11
Rank 10-consistent	1.055e-09	0.0008376	1.748e-08

1.2 Comparison over varying distributions of statements over ranks

For these tests, the number of ranks in the knowledge bases used were kept constant at 50, in order to ensure that results obtained were a result of the varying distributions and not varying numbers of ranks.

1.2.1 Knowledge Base with Uniform Distribution of statements over ranks

Query Set	Indexing vs Naive	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.002597	0.5495	0.4931
All the same	7.789e-10	7.789e-10	7.789e-10
All repeated once	0.0005995	0.004143	0.1449
Rank 1-consistent	7.161e-08	7.780e-10	7.783e-10
Rank 50-consistent	2.039e-09	1.038e-08	7.79e-10

1.2.2 Knowledge Base with Normal Distribution of statements over ranks

Query Set	Indexing vs Naïve	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.06955	0.3876	0.05412
All the same	2.435e-09	8.793e-10	7.785e-10
All repeated once	3.765e-06	0.6157	0.0001513
Rank 1-consistent	1.17e-06	8.279e-10	3.896e-09
Rank 50-consistent	7.78e-10	1.457e-06	7.782e-10

1.2.3 Knowledge Base with Exponential Distribution of statements over ranks

Query Set	Indexing vs Naïve	Binary vs Naive	Binary Indexing vs Naive
All Unique	0.4457	7.789e-10	8.275e-10
All the same	2.228e-07	7.79e-10	8.797e-10
All repeated once	0.00204	1.631e-08	1.766e-07
Rank 1-consistent	3.085e-08	7.789e-10	8.277e-10
Rank 50-consistent	1.298e-09	9.806e-09	1.161e-08