UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2021

Title: Scalable Defeasible Reasoning

Author: Joon Soo Park

Project Abbreviation: SCADR

Supervisor(s): Thomas Meyer

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

Scalable Defeasible Reasoning

Daniel Park
University of Cape Town
Cape Town, South Africa
PRKJOO001@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, and little to no work has been done on improving the scalability of defeasible reasoning algorithms. We investigate the scalability of defeasible entailment algorithms, and propose optimised versions thereof, as well as present a tool to perform defeasible entailment checks using these algorithms. We also present a knowledge base generation tool which can be used for testing implementations of these 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) [3]. Our research will focus on the latter. Knowledge representation refers to the notion of using some formal set of symbols or notation in order 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 (i.e., automated reasoning). We will utilise logics (a mechanism for formalising ways to reason [2]) to represent information.

Propositional logic [2] 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, which means that 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.

The work in Sections 1,2 and 5 was done jointly with Bailey and Hamilton.

2 BACKGROUND

2.1 Propositional Logic

Propositional Logic [2] 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 $\mathcal{K}$ ($Mod(\mathcal{K}) \subseteq Mod(\alpha)$), then we say $\mathcal{K}$ *entails* α (denoted $\mathcal{K} \models \alpha$). Propositional entailment 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 [12] [14].

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 satisfiability (SAT) solver, as discussed further in the related work section.

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 [12] [14], we have a way to model statements of the form $\alpha \mid\sim \beta$, which is read as "α typically implies β", which simply means that if α is true, then this is typically enough information to believe that β is also true. For $\alpha \mid\sim \beta$, α is called **antecedent** and β is called **consequent**. A detailed overview of this approach is provided by Kaliski in [11].

The notion of defeasible entailment (denoted $\mid\approx$) 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 $\alpha \mid\sim \beta$). Rational closure [14] and lexicographic closure [13] are two approaches to defeasible reasoning which both fall within the KLM approach, and relevant closure [5] and ranked entailment [14] are two which do not [6]. Thus, this research will focus on the first two.

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].

2.4.1 Semantic Definition. A *ranked interpretation* is simply a ranking of all interpretations in $\mathcal{W}$ (the set of all interpretations), in order of typicality, starting at rank 0 (indicating the most typical interpretations) and ending at rank n, which will include those interpretations which are the least typical, but are still plausible), followed by a single infinite rank, indicating those interpretations which are impossible to occur. In ranked interpretations, no rank can be empty. Formally, a ranked interpretation is a function $R : \mathcal{W} \to \mathbb{N} \cup \{\infty\}$, such that $R(I) = 0$ for some $I \in \mathcal{W}$, and for every $r \in \mathbb{N}$, if $R(I) = r$, then for every j such that $0 \leq j \leq r$, there is an $I \in \mathcal{W}$ for which $R(I) = j$.

R *satisfies* a formula α if α is true in all non-infinite ranks of a ranked interpretation R, denoted $r \Vdash \alpha$. For defeasible statements (statements which use the $\mid\sim$ operator), we say R *satisfies* $\alpha \mid\sim \beta$ if in the lowest rank (the most typical rank) where α holds, β also holds.

We can construct an ordering on the set of all possible ranked interpretations for a knowledge base $\mathcal{K}$, where $R_1 \leq_{\mathcal{K}} R_2$ if for every interpretation $I \in \mathcal{W}$, $R_1(I) \leq R_2(I)$. There is a unique minimal element R_m of the ordering $\leq_{\mathcal{K}}$ such that for every other ranked interpretation R_i in the set of all ranked interpretations for a given knowledge base, $R_m \leq_{\mathcal{K}} R_i$, as shown in [10]. If $R_m \Vdash \alpha \mid\sim \beta$, then we say $\mathcal{K}$ defeasibly entails $\alpha \mid\sim \beta$ (denoted $\mathcal{K} \mid\approx \alpha \mid\sim \beta$).

2.4.2 Algorithmic Definition: Base Ranks Algorithm. Before showing the algorithm, we define the *materialisation* of a knowledge base $\mathcal{K}$ as

$$\overrightarrow{\mathcal{K}} = \{\alpha \to \beta : \alpha \mid\sim \beta \in \mathcal{K}\}$$

(1) First, we create a sequence of materialisations $E_0, E_1, ..., E_{n-1}, E_\infty$ where $E_0 = \overrightarrow{\mathcal{K}}$, and each $E_i = \{\alpha \to \beta \in E_{i-1} : E_{i-1} \models \neg\alpha\}$ for $i > 0$. (This essentially means that each E_i contains only those statements $\alpha \to \beta$ in E_{i-1} such that α can be proven false according to those statements not in E_{i-1}). This process terminates when $E_i = E_{i-1}$ (or if $E_{i-1} = \emptyset$) and we set n=i-1 and $E_\infty = E_n$.
(2) We then create a ranking such that $E_n \setminus E_{n-1}$ is in the bottom rank, (call this R_∞) $E_{n-2} \setminus E_{n-1}$ is in the next highest rank, and so on. The ranking becomes such that the classical statements in $\mathcal{K}$ are in the bottom rank (the infinite rank), and statements are more general the higher the rank they are in. If a statement $\alpha \to \beta$ is in a rank i, then it is said that $\alpha \to \beta$ has base rank i. What this means is that in every ranked interpretation r of E_i, $\alpha \to \beta$ will be true in at least one of the interpretations in the most typical rank of r.

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 := K⃗;
while E_{i-1} ≠ E_i do
    E_{i+1} := {α → β ∈ E_i | E_i ⊨ ¬α};
    R_i := E_i \ E_{i+1};
    i := i + 1;
R_∞ := E_{i-1};
if E_{i-1} = ∅ then
    n := i − 1;
else
    n := i;
return (R_0, ..., R_{n-1}, R_∞, n);
```

2.4.3 Checking Defeasible Entailment. Input: a knowledge base $\mathcal{K}$ and a defeasible implication statement $\alpha \mid\sim \beta$.

(1) First, we check if $\neg\alpha$ is entailed by those statements in the infinite rank. If the negation of the antecedent is not entailed, we say that the formula, $\alpha \mid\sim \beta$, is **compatible** with the knowledge base.
(2) We then remove sets of classical implications rank by rank, starting at the highest rank and working our way down until we find a rank such that all the statements in that rank and those remaining do not entail $\neg\alpha$ (thus, α is satisfiable w.r.t. the set of statements which are in the current rank and the remaining ranks).
(3) We then check whether or not these remaining statements entail $\alpha \to \beta$.
(4) If we get to the stage where we only have the bottom rank remaining, and the statements in this rank entail $\neg\alpha$ then we can simply conclude that it is the case that $\mathcal{K} \mid\approx \alpha \mid\sim \beta$.

This algorithm only returns that a statement is defeasibly entailed by a knowledge base if this is true in terms of the minimal ranked interpretation for the knowledge base [9].

2.5 Lexicographic Closure

Lexicographic closure [13] is a refinement of rational closure, in that the entailment check is done the same way, and only the ranking of statements is done differently. A detailed description of the intricacies of this approach is provided by Casini et al in [6]. Lexicographic closure is less conservative than rational closure in terms of how much it infers from a knowledge base.

Lexicographic closure uses a more refined ranking, where statements within ranks (according to the `BaseRank` algorithm) are ranked amongst themselves, thus preserving the original ranking. We consider all possible refined rankings of these statements, and in the case of determining whether or not $\mathcal{K} \approx \alpha \mid\sim \beta$, when checking whether or not $\overrightarrow{\mathcal{K}} \models \neg\alpha$, we remove statements one at a time such that as few statements as possible are removed where the remaining statements do not entail $\neg\alpha$. If this is not able to occur, then we remove the entire top rank and continue similarly with a refinement of those statements in the following rank, etc.

Once the remaining statements do not entail $\neg\alpha$, we check whether these statements classically entail $\alpha \rightarrow \beta$. If we reach the stage where only one rank is remaining and the remaining statements still entail $\neg\alpha$, then we can return that $\mathcal{K} \approx \alpha \mid\sim \beta$.

The following is an example which may better explain how the entailment checks for rational and lexicographic closure differ:

If we have a knowledge base

$$\mathcal{K} = \{b \mid\sim f, p \rightarrow b, b \mid\sim w, p \mid\sim \neg f\}$$

which has been ranked according to the `BaseRank` algorithm as follows:

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

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

And we wish to investigate whether or not the statement $p \mid\sim w$ is entailed by the knowledge base, then:

2.5.1 *Entailment Check using Rational Closure.*

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

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

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 $p \rightarrow w$ is entailed by the remaining statements. It is not, so we conclude $\mathcal{K} \not\approx p \mid\sim w$.

2.5.2 *Entailment Check using Lexicographic Closure.*

(1) Consider all possible refined rankings of $\overrightarrow{\mathcal{K}}$ such that the statements in rank 0 have been ranked amongst themselves.

Rank 0	$b \rightarrow f$ $b \rightarrow w$
Rank 1	$p \rightarrow \neg f$
Rank ∞	$p \rightarrow b$

Figure 3: Refined Ranking A

Rank 0	$b \rightarrow w$ $b \rightarrow f$
Rank 1	$p \rightarrow \neg f$
Rank ∞	$p \rightarrow b$

Figure 4: Refined Ranking B

(2) We check if $\overrightarrow{\mathcal{K}} \models \neg p$. It does, so we remove the top statement from each of our possible refined rankings.

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

Figure 5: Refined ranking A with top statement removed

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

Figure 6: Refined ranking B with top statement removed

(3) We then check if the remaining statements in each of our refined rankings entail $\neg p$. We see that the remaining statements in refined ranking A do not entail $\neg p$, and thus we check if these statements entail $p \rightarrow w$. This is the case, so we conclude that $\mathcal{K} \approx p \mid\sim w$.

Thus, it is clear that rational closure (RC) is more conservative than lexicographic closure (LC), as we were able to conclude the statement $p \mid\sim w$ when using LC but not when using RC.

3 IMPLEMENTATION OF THE ALGORITHMS

The language used for this project is Java. Java was chosen for the usage of *TweetyProject* at section 3.1, and Java Microbenchmark Harness at section 4.3.

3.1 TweetyProject

TweetyProject is a collection of libraries that can be used to implement different areas of AI. This project uses the libraries dealing with Belief Revision, Satisfiability Reasoner and Propositional Logic Parser from the *TweetyProject*. The most essential library for this project is the classical satifiability solver. This project uses the *Sat4jSolver* of *TweetyProject* for checking if a query is satisfiable by a knowledge base. *Sat4jSolver* is one of the variants of satisfiablity solver provided by *TweetyProject*.

3.2 The Implementations

The naïve implementation was developed first and then optimised implementation using the power set was developed. After these two implementations, further optimisations were considered. Binary search, one of the most famous search heuristics, was the first approach. An implementation using binary search was developed on top of the implementation that uses power set. For the performance comparison, ternary search was also developed on top of the implementation that uses power set.

The implemented algorithms use the knowledge representation form of classical logics. The implementations first take in a defeasible knowledge base and a defeasible query as the inputs. The `BaseRank` algorithm is then executed to construct the ranking of the statements in the defeasible knowledge base. The infinite rank of this knowledge base will only consist of the classical statements in the knowledge base, if there are any. Otherwise, the infinite rank will be empty. The defeasible statements in the knowledge base are then converted to classical statements by replacing $\mid\sim$ to $\rightarrow$, and they are ranked according to the `BaseRank` algorithm. It is then checked whether the given query is entailed by the ranked knowledge base by one of the four different implementations.

3.3 Analysis

A time complexity of an algorithm is the amount of time it takes to execute the algorithm. This is given by time as a function of length of input. We use O to represent this function, also known as the Big O notation. The Big O notation considers the worst-case scenario of the algorithm. Algorithms with different implementations can have same Big O since we disregard the constants and lower order terms.

For the measure of the time complexity, we will only consider the number of entailment checks executed, and not how long it took for the entailment checker to execute. This paper focuses on the implementation around the satisfiability checker and not the satisfiability checker itself. Thus, the aim of this paper to optimise the algorithm wrapping around the satisfiability checker.

3.3.1 The naïve implementation. This implementation is based off the description from section 2.5.2. The only difference between the implementation and the description is the layout of the refinement. In the description, we consider the ranking under all possible ways of removing a statement from the current rank, which is the worst rank after removing the lower ranks if there were any. We store every refined rankings and then check whether the query is compatible with each refinements. The space usage of this method is ample since we do not have to store every refined rankings. This implementation stores the worst rank separate to the ranking of the knowledge base. It then removes the worst rank in the ranking of the knowledge base, replacing with a subset in the worst rank with a statement in the worst rank removed. This refinement is from the worst rank we stored earlier. It then checks whether the query is compatible with this ranking of the knowledge base, with substituted worst rank. If it does entail, then we replace the refined ranking from the ranking of the knowledge base with a subset of the worst rank with a different statement removed. We do this until each and every statement was removed in the worst rank, or the query is compatible with the remaining statements. When the infinite rank has been reach after removing the ranks, we do not go through the subsets of the infinite rank. We conclude whether the query is entailed by the infinite rank. The rest of the procedures work as described in the description. In summary, the only difference between the implementation and the description is whether storing every subsets of the worst rank together with rest of the statements, or storing the subsets one at a time with rest of the statements.

The below time and space complexity analysis are for the worst-case scenario.

3.3.2 Time complexity of the naïve implementation.

- The algorithm first checks the entailment of the whole ranking of the knowledge base, which equals to one entailment check.
- If the lowest rank consists of n statements, then the algorithm executes the classical entailment checker $n \times (n-1)$times.
- If each rank consists of k statements, then the algorithm executes k multiplied by the number of executions from the above points.

In total, we have $k \times (n \times (n-1)+1) = kn^2 - kn + k$. We are considering a large knowledge base where the number of ranks is strictly larger than the number of statements in the worst rank. Letting $k << n$, this is $O(n^2)$ for the time complexity.

3.3.3 Space complexity of the naïve implementation.

- The algorithm first clones the whole knowledge base of size n.
- If the worst rank consists of k statements, then the algorithm stores $k-1$ statements at a time.

In total, we have $k+n-1$. This is $O(n)$ for the space complexity.

3.3.4 The optimised implementation of the algorithm using power set. The key factor of Lexicographic Closure is to check whether we can prove the negation of the antecedent of the query we are checking for using one of the subsets of the statements in the worst rank. Using the new definition of Lexicographic Closure adapted for Datalog$^{\vee}$[16], it is possible to achieve this with fewer entailment checks. Rather than checking each subset of the worst rank, we can execute a single entailment check by combining the subsets using disjunctions.

Definition 3.1 (Power set). A power set of $\mathcal{A}$ is the set of all subsets of the set $\mathcal{A}$ including the set itself and the null or empty set, denoted as $\mathcal{P}(\mathcal{A})$. For an example, for a set $\mathcal{A}$ consisting of $\{a, b, c\}$, $\mathcal{P}(\mathcal{A}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

The implemented `PowerSet`($\mathcal{X}$)function uses this property of power set to generate a new tuple of statements. By using the previous example, $\mathcal{A} = \{a, b, c\}$, the `PowerSet`($\mathcal{X}$) function first stores the $\mathcal{P}(\mathcal{A})$ into a tuple. Then, the statements in each set of the tuple are combined by conjunction. Afterwards, the sets containing same number of statements are combined by disjunction. Thus, the `PowerSet`($\mathcal{A}$) returns the following:

$\{\{a \wedge b \wedge c\}, \{a \wedge b\} \vee \{a \wedge c\} \vee \{b \wedge c\}, \{a\} \vee \{b\} \vee \{c\}\}$.

Algorithm 2: Lexicographic Closure using PowerSet (X)

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

```
(R_0, ..., R_k, R_∞, k + 1) := BaseRank(K);
i := 0;
R := ⋃_{i=0}^{j≤k} R_j;
while R_∞ ∪ R ⊨ ¬α and R ≠ ∅ do
    for SR ∈ PowerSet(R_i) do
        R := R \ R_i;
        R := R ∪ SR;
        if R_∞ ∪ R ⊭ ¬α then
            return R_∞ ∪ R ⊨ α → β;
    R := R \ R_i;
    i := i + 1;
return R_∞ ∪ R ⊨ α → β;
```

For an example of the above algorithm, let a rank $\mathcal{R}$ consist of statements $\{a \rightarrow b, a \rightarrow c, a \rightarrow d\}$. Calling PowerSet($\mathcal{R}$) will generate the following tuple: $\{\{a \rightarrow b, a \rightarrow c, a \rightarrow d\}, \{a \rightarrow b, a \rightarrow c\}, \{a \rightarrow c, a \rightarrow d\}, \{a \rightarrow b, a \rightarrow d\}, \{a \rightarrow b\}, \{a \rightarrow c\}, \{a \rightarrow d\}\}$. The statements in each set of the tuple are combined using conjunction:$\{a \rightarrow b \wedge a \rightarrow c \wedge a \rightarrow d\}, \{a \rightarrow b \wedge a \rightarrow c\}, \{a \rightarrow c \wedge a \rightarrow d\}, \{a \rightarrow b \wedge a \rightarrow d\}, \{a \rightarrow b\}, \{a \rightarrow c\}, \{a \rightarrow d\}\}$. The sets containing same number of statements are then combined by disjunction: $\{a \rightarrow b \wedge a \rightarrow c \wedge a \rightarrow d\}, \{a \rightarrow b \wedge a \rightarrow c\} \vee \{a \rightarrow c \wedge a \rightarrow d\} \vee \{a \rightarrow b \wedge a \rightarrow d\}, \{a \rightarrow b\} \vee \{a \rightarrow c\} \vee \{a \rightarrow d\}\}$. This tuple now only contains three statements separated by comma. This is then returned by the function PowerSet($\mathcal{R}$).

The below time and space complexity analysis are for the worst-case scenario.

3.3.5 Time complexity of the implementation using PowerSet.

- If the lowest rank consists of n statements, then the algorithm executes the entailment checker n times.
- If each rank consists of k statements, then the algorithm executes k multiplied by the number of executions from the above points.

In total, we have $k \times n = kn$. With $k << n$, we have O(n) for the time complexity.

3.3.6 Space complexity of the implementation using PowerSet. For the space complexity, PowerSet($\mathcal{A}$) = $2^{|\mathcal{A}|}$ [20]. $\emptyset$ is not used since it is redundant.

- The algorithm first clones the whole knowledge base of size n.
- If the worst rank consists of k statements, then the algorithm stores 2^k statements at a time.

In total, we have $2^k + n$. So we have O(2^n) for the space complexity.

The following implementation is optimised on top of the implementation using the PowerSet.

3.3.7 The optimised implementation using Binary Search. Binary Search is used to search an element in a sorted array by dividing the array in half repeatedly. We start by setting a key to the item in the middle of the array. We then check whether the value of the element we are looking for is less than the key or greater. If it is less, then we disregard the first half of the array. Otherwise, we disregard the second half. Repeat this procedure until the value is found, or the array is empty.

Since our statements do not have "values" that we can use to compare with other statements, we do not follow the exact description of the binary search. In fact, we use the "dividing in half" of the binary search. Binary search can be used to find the rank, such that if that rank and the above ranks are removed, then the query is not compatible with the remaining statements, but removing a rank that is right below, one rank higher, will make the query compatible with the remaining statements

For an example, let $\mathcal{K}$ be a knowledge base that forms eight ranks, including the infinite rank, after applying the BaseRank algorithm. Let the rank $\mathcal{R}_i$ be a rank containing statements such that when some of the statements in that rank are removed, then the query will be compatible with the remaining statements. Let $\mathcal{R}_i$ be rank 5. We let our key to be 4, since 8 divided by 2 is 4. We check whether removing the ranks from 0 to 3 still makes the query compatible with the remaining statements. It does not, so we remove the top 4 ranks. Since the remaining are from rank 4 to rank ∞, our mid now equals to $4 + (8 - 4)/2 = 6$. Removing rank 4 and 5 will make the query compatible with the remaining statements, so we check whether removing only rank 4 also makes the query compatible with the remaining statements. Since it does not, we apply the Lexicographic Closure algorithm with the knowledge base containing statements from rank 5 to rank ∞.
The below time and space complexity analysis are for the worst-case scenario.

3.3.8 Time complexity of the implementation using binary search.

- If the lowest rank consists of k statements, then the algorithm executes the entailment checker k times.
- To find a rank where the subsets has to be checked for the compatibility of the query, first, two entailment checks are required and then it takes $\log_2(n)[1]$ times to find the rank. n is the number of ranks.

In total, we have $k + 2 \times \log_2(n) = 2\log_2(n) + k = O(\log(n))$.

3.3.9 Space complexity of the implementation using binary search.

- The algorithm first clones the whole knowledge base of size n.
- If the worst rank consists of k statements, then the algorithm stores 2^k statements at a time.

In total, we have $2^k + n = 2^k + n$. So we have O(2^n) for the space complexity.

Algorithm 3: Lexicographic Closure using Binary Search

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

$(R_0, ..., R_k, R_\infty, k+1) :=$ `BaseRank`$(\mathcal{K})$;
left := 0;
right := $k+1$;
$R := \bigcup_{i=0}^{j \leq k} R_j$;
mid := `Binary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{right})$;
if mid == ∞ **then**
  return False;
else
  $R := \bigcup_{i=\text{mid}}^{j \leq k} R_j$;
  for SR $\in$ `PowerSet`(R_i) **do**
    $R := R \setminus R_i$;
    $R := R \cup \text{SR}$;
    if $R_\infty \cup R \not\models \neg\alpha$ **then**
      return $R_\infty \cup R \models \alpha \rightarrow \beta$;
  $R := R \setminus R_i$;
  return $R_\infty \cup R \models \alpha \rightarrow \beta$;
Function `Binary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{right})$**:**
  if right > left **then**
    mid = left + (right - left)/2;
    $R := \bigcup_{i=\text{mid}+1}^{j \leq k} R_j$;
    if $R_\infty \cup R \models \neg\alpha$ **then**
      return `Binary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{mid}, \text{right})$
    else
      $R := \bigcup_{i=\text{mid}}^{j \leq k} R_j$;
      if $R_\infty \cup R \models \neg\alpha$ **then**
        return mid
      else
        return `Binary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{mid})$
  else
    return ∞

The following implementation is optimised on top of the implementation using the `PowerSet`.

3.3.10 The optimised implementation using Ternary Search. Ternary Search uses the properties of Binary Search, except Ternary Search uses two keys, mid1 and mid2, to set the intervals of the array. The first key is set to diving the size of the array by three, and the second key to diving the size of the array by three and multiplying it by two. We then check whether removing the ranks from the first key and above makes the query compatible with the remaining statements. If it does not, we check whether removing the ranks above our second key makes the query compatible with the remaining statements. If removing the ranks from the first key and above makes the query compatible with the remaining statements then we have

Algorithm 4: Lexicographic Closure using Ternary Search

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

$(R_0, ..., R_k, R_\infty, k+1) :=$ `BaseRank`$(\mathcal{K})$;
left := 0;
right := $k+1$;
$R := \bigcup_{i=0}^{j \leq k} R_j$;
mid := `Ternary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{right})$;
if mid == ∞ **then**
  return False;
else
  $R := \bigcup_{i=\text{mid}}^{j \leq k} R_j$;
  for SR $\in$ `PowerSet`(R_i) **do**
    $R := R \setminus R_i$;
    $R := R \cup \text{SR}$;
    if $R_\infty \cup R \not\models \neg\alpha$ **then**
      return $R_\infty \cup R \models \alpha \rightarrow \beta$;
  $R := R \setminus R_i$;
  return $R_\infty \cup R \models \alpha \rightarrow \beta$;
Function `Ternary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{right})$**:**
  if right > left **then**
    mid1 = left + (right - left)/3;
    mid2 = right - (right - left)/3;
    $R := \bigcup_{i=\text{mid1}+1}^{j \leq k} R_j$;
    if $R_\infty \cup R \models \neg\alpha$ **then**
      $R := \bigcup_{i=mid2+1}^{j \leq k} R_j$;
      if $R_\infty \cup R \models \neg\alpha$ **then**
        return `Ternary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{mid2} + 1, \text{right})$;
      else
        $R := \bigcup_{i=\text{mid2}}^{j \leq k} R_j$;
        if $R_\infty \cup R \models \neg\alpha$ **then**
          return mid2;
        else
          return `Ternary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{mid1} + 1, \text{mid2} - 1)$;
    else
      $R := \bigcup_{i=\text{mid1}}^{j \leq k} R_j$;
      if $R_\infty \cup R \models \neg\alpha$ **then**
        return mid
      else
        return `Ternary`$((R_0, ..., R_k, R_\infty, k+1), \alpha \mid\sim \beta, \text{left}, \text{mid1})$
  else
    return ∞

found the rank to check the subsets of. The same procedure is done for the second key. If the corresponding rank is not found, then the two keys are adjusted according to the size of the remaining ranks. This procedure is repeated until the corresponding rank has been found. When it is found, the Lexicographic Closure algorithm is applied to the remaining statements and the query.

3.3.11 Time complexity of the implementation using ternary search.

- If the lowest rank consists of k statements, then the algorithm executes the entailment checker k times.
- To find a rank where the subsets has to be checked for the compatibility of the query, first, three entailment checks are required and then it takes $\log_3(n)[1]$ times to find the rank. n is the number of ranks.

In total, we have $k + 3 \times \log_3(n) = k + 3\log_3(n) = O(\log(n))$.

3.3.12 Space complexity of the implementation using ternary search.

- The algorithm first clones the whole knowledge base of size n.
- If the worst rank consists of k statements, then the algorithm stores 2^k statements at a time.

In total, we have $2^k + n = 2^k + n$. So we have $O(2^n)$ for the space complexity.

3.4 Summary of the analysis

Time and space complexity of the implementations:

	Naïve	Power set	Binary Search	Ternary Search
Time	$O(n^3)$	$O(n^2)$	$O(n\log(n))$	$O(n\log(n))$
Space	$O(n)$	$O(2^n)$	$O(2^n)$	$O(2^n)$

4 EXPERIMENT EXECUTION

The aim of this experiment is to see the performance of the four implementations on different variants of knowledge bases. The tests are made on knowledge bases with different sizes and distributions. The knowledge base generator by Bailey is able to generate knowledge bases with different parameters. The parameters that can be adjusted are the number of ranks, total number of statements and the distribution of the statements. The number of ranks is the total number of ranks that will be generated after applying the `BaseRank` algorithm to the generated knowledge base. The total number of statements is the total number of statements in the generated knowledge base. The statements are distributed throughout the ranks according to the distribution parameter. Currently, the available distributions are uniform, normal, inverse normal, exponential and inverse exponential. The knowledge base generator is able to generate defeasible statements and classical statements, and those statements consists of numeric atoms. The statements are in form $\{\alpha \mid\sim \beta | \alpha, \beta \in \mathbb{Z}\}$.

For the comparison between the implementations using binary search and ternary search, it is believed that the key factor is the time taken to find the rank that is required for the `BaseRank` function to be applied. For this purpose, the consequent is set to 1 for all queries testing the knowledge base.
For an example, let a query α consist of 5 as its antecedent and 1 as its consequent such that $5 \mid\sim 1$. Let a knowledge base $\mathcal{K}$ consist of 10 defeasible statements and no classical statements. Let the generated ranks from $\mathcal{K}$ by the `BaseRank` algorithm be the following:

Rank 0	$1 \rightarrow 0, 2 \rightarrow 1$
Rank 1	$3 \rightarrow \neg 0, 3 \rightarrow 2$
Rank 2	$4 \rightarrow 0, 4 \rightarrow 3$
Rank 3	$5 \rightarrow \neg 0, 5 \rightarrow 4$
Rank 4	$6 \rightarrow 0, 6 \rightarrow 5$
Rank ∞	$\emptyset$

Both implementations removes rank 0 and rank 1, calling the `PowerSet` function on rank 2. It will then decide whether $5 \mid\sim 1$ is compatible with the remaining statements. In this example, $5 \mid\sim 1$ is compatible with the remaining statements but they do not entail $5 \mid\sim 1$, thus returning **false**.

4.1 Specifications

The machine used for testing the implementations have the following specifications:

- CPU: AMD Ryzen 5 2600, 6 core 12 thread, 3.4GHz clock speed
- RAM: 2×8gb DDR4
- Power supplier: 530W

The machine used to test the implementations fails to execute the subset rank function on a rank with 13 or more statements. Since the space complexity of the subset rank is $O(2^n)$, this machine returns a "out of space" error when the function is called with 13 or more statements. Due to the `PowerSet` function being called by every implementation, except the naïve implementation, the exponential and inverse exponential distribution was not tested.

4.2 Hypothesis

The main focus of this experiment is comparing the execution time of two implementations that uses binary search and ternary search. By the analysis in section 3.3, binary search takes $2\log_2(n)$ times for the worst case, where ternary search takes $3\log_3(n)$. Meaning that the ternary search performs slightly better than the binary search for the worst case. It is expected to obtain results where the implementation using ternary search performs better than the implementation using binary search.
Due to the structure of the implementations using binary search and ternary search, it is expected to observe a result where the execution time decreases as the index of the rank containing the antecedent of the query increases. This is due to the implementations executing the entailment checker with fewer statements as the index increases.

4.3 Java Microbenchmark Harness

Java Microbenchmark Harness (JMH) is a tool used to benchmark Java. JMH was developed by the team who developed Java Virtual Machine (JVM). JMH is more accurate than calling the System.currentTimeMills() function before and after the execution of the algorithms, and calculating the difference. JMH runs benchmarks in a way where the results are not erroneous due to the optimization of the virtual machine. JMH eliminates dead codes, meaning that it will get rid of codes that is never used or does nothing. Since we want to stress test our implementation around the satifiability solver, we use JMH to test our implemented algorithms and measure the execution time.

The benchmarking starts after the execution of the `BaseRank` algorithm, exactly before the start of the execution of the reasoner algorithms. This was done to measure the exact difference of the execution time between the algorithms, excluding as many external factors as possible.

The aim of this project is to investigate the scalability of defeasible entailment algorithms, thus it is focused on optimising the implementations on top of the classical reasoner, and not the classical reasoner itself.

4.4 Testing

Each tests were done with the following parameters:

- Fork (value = 2)
- Measurement (iterations = 10, time = 1)
- Warmup (iterations = 5, time = 1)

Fork, also called trail, is the number of trials the JMH will perform. A trail contains a set of warmups and iterations. We will be running two trials per each benchmark tests. Warmup is the number of warmups it performs before running the actual benchmark. This is to allow the JVM to perform and class loading, compilation to native code, and caching steps before running the benchmark for gathering results. The results obtained from the warmups are disregarded. We will run 5 warmup runs per trail. The measurements is the number of iterations the JMH performs after warming up. It gathers the worst, average and the best values from the iterations. Our measurement is set to 10, meaning that we will run our benchmark tests 10 times per trail. In total, the JMH will gather results from 20 runs.

4.4.1 Test 1. The first test consists of a knowledge base with 50 ranks and 200 statements, containing classical statements. The statements in this knowledge base is distributed with normal distribution. Eight queries were selected to test the implementations starting from query containing the antecedent that is not in the knowledge base. When the algorithms are executed with these kinds of queries and a knowledge base, the query is compatible with the knowledge base without removing any statements from the knowledge base. Thus, this can be used to check the execution time when the index is 0. The following queries contains the antecedent of the statements in every eighth rank. The index ∞ is the index of the rank containing classical statements. The execution time is measured in milliseconds.

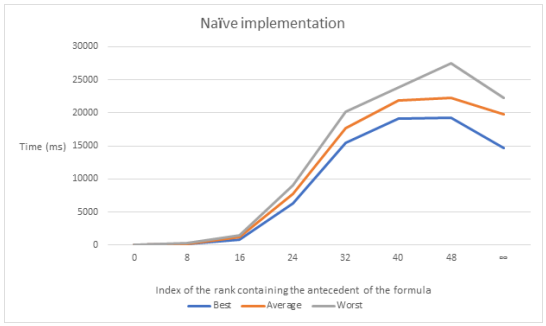

Figure 7: Execution time for the naïve implementation

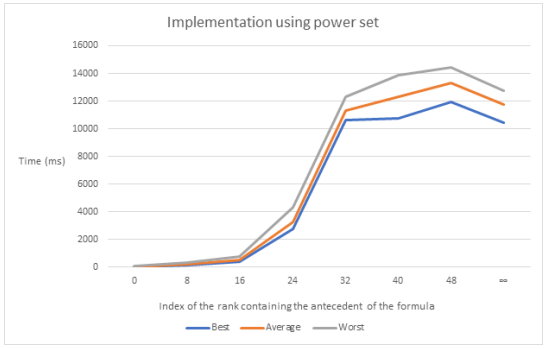

Figure 8: Execution time for the implementation using `PowerSet`

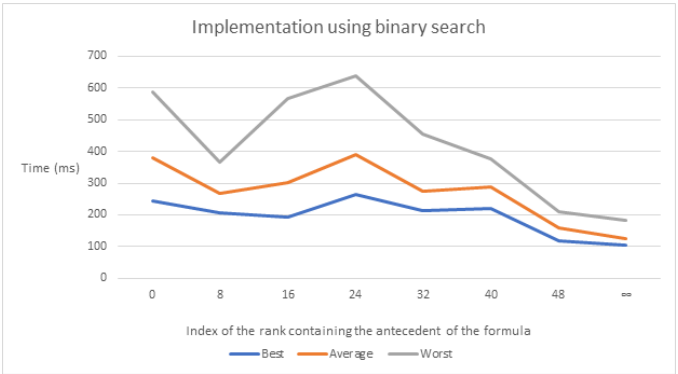

Figure 9: Execution time for the implementation using binary search

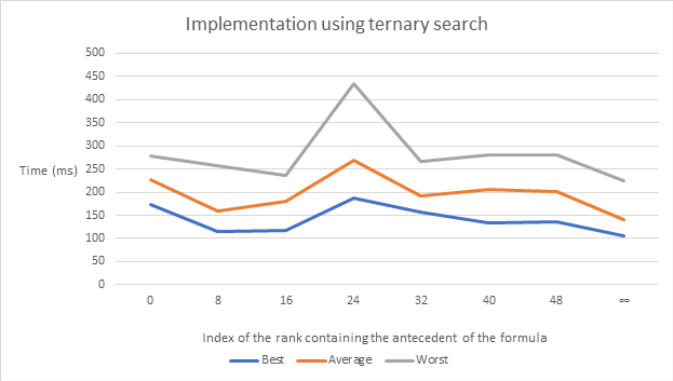

Figure 10: Execution time for the implementation using ternary search

Figure 7 and figure 8 shows an exponential increase in the execution time as the index increases. This is due to the implementations having to go through every rank before reaching the rank it is looking for. By inspection, it is clear that the implementation using `PowerSet` performs significantly better than the naïve implementation. Both figures have a slight drop from index 48 to ∞. This is due to the implementations not having to go through the subsets of the infinite rank. Since the results for the naïve implementation and the implementation using `PowerSet` shows the correlation between the execution time and the index, further tests are done for these two implementations.
Figure 9 and figure 10 shows unusual graphs. Not much could be analysed by these results, thus more tests were done focusing on these two implementations.

4.4.2 Test 2. The second test was conducted with a knowledge base of 50 ranks and 200 statements, not containing any classical statements. The statements in this knowledge base is distributed with uniform distribution. For more results and accuracy, every single antecedent in the knowledge base was tested.

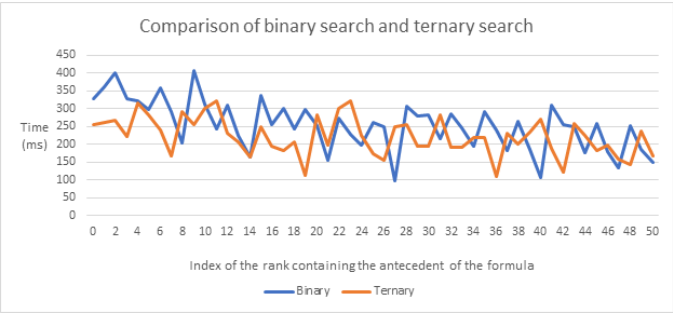

Figure 11: The comparison of the execution time for the implementations using binary search and ternary search

The summary of the figure 11:

	Binary Search	Ternary Search
Best	96.60	109.00
Average	253.97	222.03
Worst	407.11	323.46

This test shows that the implementation using ternary search performed slightly better than the implementation using binary search on average. This is due to ternary search having better time complexity than the binary search. The binary search outperformed the ternary search for the best case. This is due to the required number of execution of the entailment checker for the binary search is at least two, and three for the ternary search.

4.4.3 Test 3. This test aims to see the affect of the entailment checker to the execution time of the implementations using search heuristics. Cases where fewer comparisons are made, but resulting in longer execution time due to large number of statements in the rank, should be considered.
This test was conducted with a knowledge base of 250 ranks and 1000 statements, not containing any classical statements. The statements in this knowledge base is distributed with uniform distribution

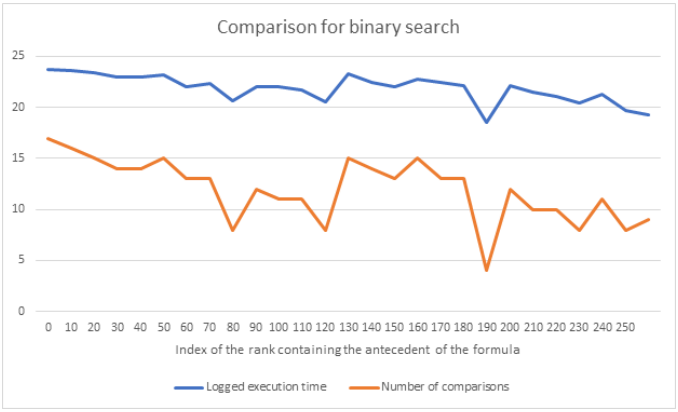

Figure 12: The comparison of logged execution time against the number of entailment checks are called for the implementations using binary search

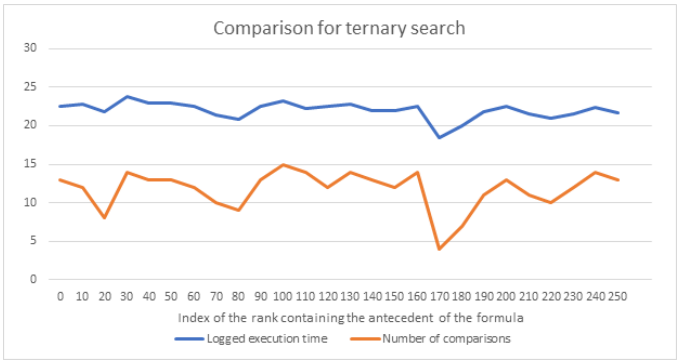

Figure 13: The comparison of logged execution time against the number of entailment checks are called for the implementations using ternary search

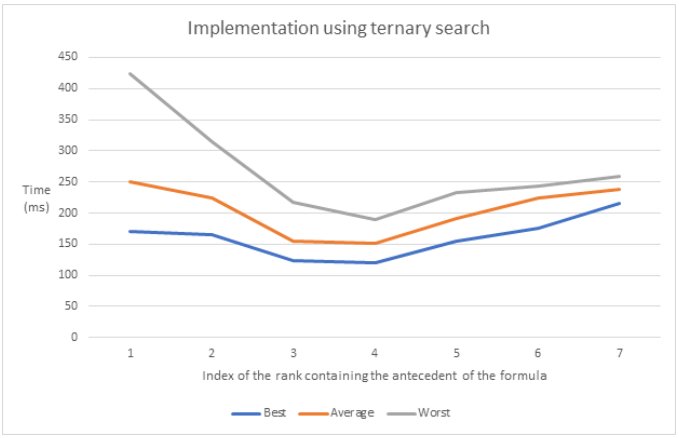

Figure 14: Execution time for the implementation using ternary search

To measure the the number of entailment checks called, a counter was made to increment every time a comparison was made by the implementations. The entailment checks are called when the comparisons are made.
Figure 12 and figure 13 shows a clear correlation between the execution time and the number of entailment checks called. The trends are not exactly identical, but a reasonable reference to each other. This allows us to focus more on the implementations by using the knowledge base with uniform distribution.

4.4.4 Test 4. This test was done to see if the time complexity of binary search is $2\log_2(n)$ and $3\log_3(n)$ for the ternary search. The knowledge base from test 3, 250 ranks, 1000 statements, not containing any classical statements and uniformly distributed, was used for this test. The query was set to every antecedent in the knowledge base and the number of entailment checks made were stored for both implementations using binary search and ternary search.
The summary of the test:

	Binary Search	Ternary Search
Best	2	3
Average	12.10	11.86
Worst	16	15

The best case for both implementations are as expected. $2\log_2(250) = 15.93 \approx 16$ and $3\log_3(250) = 15.07 \approx 15$. These values are equal to what we observed.

4.4.5 Test 5. This test was done to see the behaviour of the implementation using ternary search on knowledge bases with different distributions.

Figure 14 was tested with a knowledge base of 50 ranks, 200 statements, without any classical statements and distribution of inverse normal.
By referring to figure 10, figure 11 and figure 14, the graphs tend to have a shape of the distribution of the corresponding knowledge bases. This is due to statements are distributed with the distribution of the knowledge base, causing longer execution time of the entailment checker with the ranks containing more statements relatively than the other ranks.

4.5 Conclusions

From test 1, we have observed the behaviour of the naïve implementation and the implementation using `PowerSet`. The execution time of these implementations increase as the index increases.
From test 2, we have observed that the implementation using binary search outperforms the implementation using ternary search for the best case, but the opposite for the average and the worst case. The summary of the results showed that the implementations worked as expected, but more tests were required.
From test 3, we have observed that, for knowledge base where the statements are distributed with uniform distribution, the execution time and the number of comparisons made were highly correlated. From this, we can rely on the execution time of the implementations, not having to measure the number of comparisons made separately.
From test 4, we have shown that the implementations using binary search and ternary search do have the time complexities analysed in section 3.3.8 and section 3.3.11.
From test 5, we have shown that the shape of the execution time of the implementation using ternary search follows the shape of the distribution of the knowledge base.

From the analysis and the tests, it has been shown that the implementation using ternary search performs better than the implementation using binary search. This was not a drastic improvement, but a step towards a better scalable defeasible reasoner.

5 RELATED WORK

Rational closure (RC) and Lexicographic Closure (LC) both reduce to a series of classical entailment checks. This means it is at its core the problem of boolean satisfiability (SAT) and thus, research pertaining to this problem is highly relevant to our proposed project.

5.1 Classical Satisfiability Solving

A *satisfiability (SAT) solver* is a system which computes the existence of a satisfying valuation (i.e., a model) for a specified boolean formula. There are many such SAT solvers designed for classical propositional satisfiability checking, thus understanding the approaches in this field will aid in the search for a suitable SAT solver to be used in the prototype defeasible reasoners.

5.1.1 The DPLL Algorithm. *DPLL* [8] is a complete and sound SAT checking algorithm. DPLL uses backtracking (retracting the most recent assignment and performing a different assignment) to speculatively test various assignment combinations, along with Boolean Constant Propagation (BCP) which constantly ensures that all variables which need to be true to satisfy a problem are kept true. DPLL forms the basis of many modern SAT solvers.

5.1.2 The CDCL Process. GRASP [15, 18], introduced the *Conflict Driven Clause Learning (CDCL)* process, in which variable assignments which potentially cause conflicts are cached. This results in better execution, and unlike DPLL [8], this does not rely on chronological backtracking, and as such is more efficient.

5.1.3 Other Advancements. Several more advanced SAT solvers utilise low-level optimisations such as different storage structures and parallelism in order to obtain better performance [19], e.g., GridSAT [7] and Chaff [17]. Due to prolific research being done on developing efficient SAT solving algorithms, many efficient SAT solver implementations are now freely available.

6 FUTURE WORK

Currently the knowledge base generator by Bailey was not able to generate complicated statements. If further research is done to create complicated statements, more analysis could be made with these implementations. The biggest knowledge base that was used for this paper consists 250 ranks and 1000 statements. A more powerful machine can be used to test larger knowledge bases. Since the `PowerSet` has a space complexity of $O(2^n)$, better implementations can be made to decrease this space complexity.

REFERENCES

[1] Nitin Arora, Mamta Martolia Arora, and Esha Arora. A novel ternary search algorithm. *International Journal of Computer Applications*, 144(11), 2016.

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

[3] 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.

[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] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, page 37, New York, NY, USA, 2003. Association for Computing Machinery.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[9] Michael Freund. Preferential reasoning in the perspective of poole default logic. *Artificial Intelligence*, 98(1):209–235, 1998.

[10] Laura Giordano, Valentina Gliozzi, Nicola Olivetti, and Gian Luca Pozzato. Semantic characterization of rational closure: From propositional logic to description logics. *Artif. Intell.*, 226:1–33, 2015.

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

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

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

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

[15] Joaão P. Marques-silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[16] Matthew Morris, Tala Ross, and Thomas Meyer. Algorithmic definitions for klm-style defeasible disjunctive datalog. *South African Computer Journal*, 32(2):141–160, 2020.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery.

[18] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, page 220–227, USA, 1997. IEEE Computer Society.

[19] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

[20] Cong-Cong Xing. Seven different proofs for| p (a)|= 2n. *Journal of Computing Sciences in Colleges*, 31(5):53–61, 2016.