UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS Honours Project Final Paper 2024

Title: [*A Study of Parameters and Optimization Strategies in Defeasible Knowledge Base Generation*]

Author: Mamodike Sadiki

Project Abbreviation: EXTRC

Supervisor(s): Tommie Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	10
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	15
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**	

A Study of Parameters and Optimization Strategies in Defeasible Knowledge Base Generation

Mamodike Sadiki
University of Cape Town
Cape Town, South Africa
sdkmam001@myuct.ac.za

ABSTRACT

Knowledge Representation and Reasoning (KRR) is a fundamental concept in Artificial Intelligence (AI). It provides frameworks and methods for consistently and coherently representing real-world information in AI systems. This structured representation of knowledge is essential for communication, decision-making, and problem-solving applications. One formal framework used within KRR is propositional logic, which represents information as atomic propositions that are either true or false. Considering that not everything in the real world has a definitive state of being 'true' or 'false', this poses a limitation that can lead to exceptions and inaccuracies in reasoning. Defeasible reasoning is employed to address such issues. This approach accommodates the notion of 'typicality' in knowledge representation. Defeasible reasoning uses frameworks like the KLM framework and concepts such as ranked interpretations and rational closure to make more flexible and context-sensitive inferences. The development of defeasible reasoners relies heavily on knowledge bases of different types for testing, and there are not many in existence. hence this paper aims to address this limitation in the existence of propositional knowledge bases with exceptions and their generators by implementing a defeasible knowledge base generator and studying the parameters involved using the Monte Carlo simulation.

KEYWORDS

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

1 INTRODUCTION

Artificial Intelligence (AI) plays a vital role in the advancement of computing technology, it provides a wide range of methods and frameworks that allow computers to perform tasks requiring some intellect, in the context of Knowledge Representation and Reasoning (KRR), an example would be automated reasoning systems used for decision-making. These systems exist to serve some purpose in the real world and the real world is complex, thus making it difficult to both represent and reason about knowledge. In propositional logic, the properties of the world are defined as logical propositions that are either true or false. When reasoning with such a representation of knowledge inaccurate inferences can be made about information that is neither 'true' nor 'false' in the real world. This indicates that propositional logic has both a knowledge representation and reasoning limitation. To address this, the KLM framework extends the propositional logic by introducing the notion of 'typicality' and uses defeasible reasoning to make inferences[12, 13]. Defeasible reason is a form of reasoning where inferences are made on incomplete information or information that may change, which in turn may change the inferences. Inferences are made relative to a collection of information, commonly referred to as a knowledge base. A knowledge base in which the addition or removal of information can change previously made inferences is referred to as a non-monotonic knowledge base, and one that does not change inferences is monotonic.

A defeasible reasoning technique of interest in this paper is the Rational Closure. This paper focuses on the development of defeasible knowledge bases for testing the Rational Closure algorithm and its extension, as well as studying the parametric constraints used in the development of the knowledge bases. It further forms a part of a larger project that aims to extend defeasible reasoning beyond rational closure. The other two parts of this project focus on optimizing existing Rational Closure algorithms and extensions and analyzing the process in which inferences are made. One such part, by Maqhobosheane Mohlerepe, involves a comprehensive study of previous Rational Closure implementations and a Scalability evaluation of the reasoners, along with exploring the use of caching as a Rational Closure optimization technique[18]. Another part, led by Vincent Moloi involves the implementation of a debugger tool for analyzing the defeasible reasoning process using the Rational Closure and Lexicographic closure reasoning techniques to allow users to better understand the process of drawing a conclusion given a query on a specific knowledge base[19].

While each part addresses different concerns in the overall project this paper is concerned with That there are not that many knowledge base generators that generate defeasible knowledge bases. Considering that their main function is to represent real-world information in the form of statements with exceptions, which are used in the development and testing of defeasible reasoners. Essentially, the accuracy of reasoning algorithms depends on the testing data and its level of expressiveness relative to the context in which they will be used. Hence, the main aim is to implement a variant of a pseudo-random defeasible knowledge base generator that is more expressive and optimal. Another is to Study the parameters that are involved in knowledge base generation. To provide a knowledge base generator that is more expressive, the statements generated will have a wide range of complexity to choose from and multiple distributions of data. Parallel programming will be used on multiple tasks to optimize the knowledge base generator, and through databases optimize by using existing knowledge bases(space-time trade-off). To analyze the parameters, a Monte Carlo simulation will be used and statistical analysis performed to understand the correlation between parameters and the generation time.

2 BACKGROUND

2.1 Propositional Logic

2.1.1 Syntax. Propositional logic is a branch of logics that uses atomic propositions and Boolean connectives as building blocks for representing knowledge [11]. Formally, a set of atomic propositions $\mathcal{P}$ and Boolean connectives $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ define the propositional language, denoted by $\mathcal{L}$ such that for $\alpha, \beta \in \mathcal{P}$, the language $\mathcal{L}$ is recursively defines as $\mathcal{L} :: \alpha, \neg\alpha, \alpha \wedge \beta, \alpha \vee \beta, \alpha \rightarrow \beta, \alpha \leftrightarrow \beta, ...$[12]. A knowledge base, denoted $\mathcal{K}$, is a finite subset of the propositional language $\mathcal{L}$, $\mathcal{K} \subset \mathcal{L}$.

2.1.2 Semantics. Atomic propositions and Boolean connectives can be used to construct complex propositions, where the interpretation can be derived by evaluating the atomic propositions and Boolean connectives. This is possible because atomic propositions have assigned to them truth values (True or False), $I : \mathcal{P} \rightarrow \{T, F\}$[12]. Interpretations help us describe all the possible states of the world; the set of all interpretations is denoted by $\mathcal{U}$. An interpretation I satisfies a formula $\alpha \in \mathcal{L}$, denoted $I \Vdash \alpha$, if the truth value of α under the interpretation I is True [17, 20, 22].A formula can be satisfied by multiple interpretations, any interpretation that satisfies α is a model of α, denoted Mod(α). The same applies to a set of formulas. So, given a knowledge base $\mathcal{K}$, The set of interpretations that satisfy all the formulas in a knowledge base $\mathcal{K}$ are the models of $\mathcal{K}$, denoted Mod($\mathcal{K}$) [17, 20, 22].

2.1.3 Classical Entailment. Entailment is a fundamental concept in classical reasoning, as it allows us to use a set of formulas to derive conclusions. It shows us how some conclusions are a logical consequence of premises [24]. Formally, for $\alpha, \beta \in \mathcal{L}$, α entails β, or β is said to be a logical consequence of α, denoted $\alpha \models \beta$, if and only if the models of α are a subset of the models of β, denoted as $Mod(\alpha) \subseteq Mod(\beta)$[12, 17, 20, 22].

2.1.4 Limitations of Classical Entailment. Although classical entailment allows us to conclude from premises or statements in knowledge bases, there are cases where such conclusions may change due to the addition of new premises. These kinds of knowledge bases are referred to as Non-monotonic. in contrast to those that retain their conclusions, which are monotonic[12]. Non-monotonic knowledge bases require a Non-monotonic approach to reasoning, such as Defeasible reasoning.

2.2 Defeasible reasoning

This type of reasoning is used on non-monotonic knowledge base due to its ability to derive conclusions from uncertain or incomplete information, which makes it suited reasoning or deriving conclusions in the real world [1, 12, 17, 20, 22]. Taking the real world in consideration, lets suppose we create a knowledge base that represents the statement *"Every UCT bus that leaves upper campus goes to Tugwell before heading to its destination, but that's not the case with the Hiddingh bus"*: { *UCT bus⟶ goes to tugwell, UCT bus → hiddingh bus, UCT bus → tugwell bus, hiddingh bus → ¬ goes to tugwell*}. Now if we are to query whether a Hiddingh bus is a UCT bus classical reasoning would suggest that there is no Hiddingh bus since it does not go to tugwell first, which in reality is false, just because something deviates from what is generally true it does not make it false or non-existent [17, 20, 22]. Kraus, Lehmann, and Magidor address this through the use of the KLM framework which introduces the concept of 'typicality'. denoted $\vdash$ [12, 24]

2.2.1 KLM: Defeasible Entailment. The framework does so by replacing the material implications in formulas that cause exceptions within the knowledge base that give rise to exceptions, so rather have {"*UCT bus*" $\vdash$ "*goes to Tugwell*" than {"*UCT bus*" $\rightarrow$ "*goes to Tugwell*". Knowledge bases with such statements are called defeasible knowledge bases conclusions are drawn from them using Defeasible entailment, denoted by $\mid\!\sim$. There are no specific methods to determine if a defeasible entailment is satisfied, but there are KLM properties that must be satisfied for the defeasible entailment to hold, referred to as the LM – Rational [3, 9, 12, 13, 16, 17, 20, 22] [12]:

$$Reflexivity : \mathcal{K} \mid\!\sim \alpha \vdash \alpha$$

$$Leftlogicalequivalence : \frac{\mathcal{K} \mid\!\sim \alpha \leftrightarrow \beta, \mathcal{K} \mid\!\sim \alpha \vdash \gamma}{\mathcal{K} \mid\!\sim \beta \vdash \gamma}$$

$$Rightweaking : \frac{\mathcal{K} \mid\!\sim \alpha \rightarrow \beta, \mathcal{K} \mid\!\sim \gamma \vdash \alpha}{\mathcal{K} \mid\!\sim \gamma \vdash \beta}$$

$$And : \frac{\mathcal{K} \mid\!\sim \alpha \vdash \beta, \mathcal{K} \mid\!\sim \alpha \vdash \gamma}{\mathcal{K} \mid\!\sim \alpha \vdash \beta \wedge \gamma}$$

$$Or : \frac{\mathcal{K} \mid\!\sim \alpha \vdash \gamma, \mathcal{K} \mid\!\sim \beta \vdash \gamma}{\mathcal{K} \mid\!\sim \alpha \vee \beta \vdash \gamma}$$

$$CautiousMonotonicity : \frac{\mathcal{K} \mid\!\sim \alpha \vdash \gamma, \mathcal{K} \mid\!\sim \alpha \vdash \beta}{\mathcal{K} \mid\!\sim \alpha \wedge \beta \vdash \gamma}$$

$$Rationalmonotonicity : \frac{\mathcal{K} \mid\!\sim \alpha \vdash \gamma, \mathcal{K} \not\mid\!\sim \alpha \not\vdash \beta}{\mathcal{K} \mid\!\sim \alpha \wedge \beta \vdash \gamma}$$

2.2.2 Preferential Interpretations. The semantics of "typicality" are based on preferential semantics, which imposes an ordering of interpretations so that the most preferred interpretation is considered before the least preferred. This is useful in defeasible reasoning since it looks at more probable interpretations of the world.[12, 13, 13, 16, 17, 20, 22].

The ordering is achieved by assigning ranks from most typical (0) to least typical (n), and for the improbable a rank of ∞. Formally, $R : \mathcal{U} \rightarrow \mathbb{N} \cup \infty$, where $\mathcal{U}$ is the set of interpretations, for interpretation $u \in \mathcal{U}$, if $R(u) = 0$, it means that the interpretation u the most typical of interpretations. All ranked interpretations must satisfy the convexity property, which states that for every $i \in \mathbb{N}$, if $u \in \mathcal{U}$ such that $R(u) = i$, then there must be a v such that $R(v) = j$ with $0 \leq j < i$ [4, 17, 20, 22]. This property ensures that there are no empty ranks within the ordering[15, 17, 20, 22]. $R^{\mathcal{K}}$ denotes a ranking interpretation R with respect to knowledge base $\mathcal{K}$[4]. A ranked interpretation R satisfies a defeasible implication $\alpha \vdash \beta$ if in the lowest rank where α is satisfied, β is satisfied in every interpretation within that specific rank, this is formally denoted as $R \Vdash \alpha \vdash \beta$[4]. R is considered a model of $\alpha \vdash \beta$ [23] and more importantly, R entails $\alpha \vdash \beta$ if and only if the implication $\alpha \rightarrow \beta$ holds true in all the most typical interpretations of α, thus making it a monotonic entailment [12, 17, 20, 22].

2.2.3 Minimal Ranked Entailment. To define the semantics of non-monotonic entailment we explore the minimal ranked entailment

and its partially ordering $\preceq_{\mathcal{K}}$ of all ranked models of a knowledge base $\mathcal{K}$. The ordering is such that for any two ranked interpretations $\mathcal{R}_1$ and $\mathcal{R}_2$, $\mathcal{R}_1 \preceq_{\mathcal{K}} \mathcal{R}_2$ if for every interpretation $v \in \mathcal{U}$, $\mathcal{R}_1(v) \leq \mathcal{R}_2(v)$ [4]. The most typical models are ranked lower and are referred to as the minimal ranked interpretations, denoted $\mathcal{R}^{\mathcal{K}}_{RC}$ [7]. For the defeasible knowledge base $\mathcal{K}$, the minimal ranked interpretation satisfying $\mathcal{K}$, defines an entailment relation referred to as the minimal ranked entailment, denoted $\approx$. For any defeasible implication $\alpha \vdash \beta$, $\mathcal{K} \approx \alpha \vdash \beta$ holds if and only if $\mathcal{R}^{\mathcal{K}}_{RC} \Vdash \alpha \vdash \beta$ [12, 17, 20, 22].

2.3 Rational Closure

An alternative to the minimal ranked entailment is the rational closure, it defines the the non-monotonic entailment syntactically[12]. Informally, it is a process used to derive conclusions or inference on a defeasible knowledge bases [5], while ensuring the conclusions are logically sound and consistent. It draws conclusions from a ranked defeasible Knowledge base, hence extending preferential entailment [17, 20, 22, 24]. It consists of the ranking component and inference component.

2.3.1 Materialization. prior to ranking the defeasible implications are materialized to ensure that they exceptional [5], the counterpart of each defeasible implication $\alpha \vdash \beta$ is turned into a material implication $\alpha \rightarrow \beta$. For a knowledge base $\mathcal{K}$, its materialized set or counterpart is denoted as $\overrightarrow{\mathcal{K}}$. Formally, $\overrightarrow{\mathcal{K}} = \{\alpha \rightarrow \beta : \alpha \vdash \beta \in \mathcal{K}\}$. A propositional statement (atomic proposition or complex proposition) α is exceptional in a knowledge base $\mathcal{K}$ if and only if $\mathcal{K} \models \neg\alpha$, meaning that α is false in all the most typical interpretations in every ranked model in $\mathcal{K}$ [17, 20, 22].

2.3.2 BaseRank Algorithm. Ranking is achieved using the base rank algorithm. Given a knowledge base $\mathcal{K}$, the base rank algorithm assigns propositional formulas to ranks based on whether they are exceptional during specific iterations. Each iteration evaluates exceptionalism on a subset of the original knowledge base $\mathcal{K}$. The overall results are ranks associated with with sets of exceptional formulas, $\mathcal{R}_i : \mathcal{E}_i$ [4]. Materialization of the knowledge base $\mathcal{K}$ is the starting point for the base rank algorithm, the resulting knowledge base is assigned to initial exception set $\mathcal{E}_0 = \overrightarrow{\mathcal{K}}$. Then subsequent exception sets $\mathcal{E}_i$ are generated by assessing whether formulas in previous exception sets $\mathcal{E}_{i-1}$ are exceptional, if $\alpha \rightarrow \beta \in \mathcal{E}_{i-1}$ and $\mathcal{E}_{i-1} \models \neg\alpha$ then $\alpha \rightarrow \beta$ must be moved into $\mathcal{E}_i$. This process is continuous until there in no formula $\alpha \rightarrow \beta \in \mathcal{E}_i$ with exceptions, $\mathcal{E}_i \not\models \neg\alpha$. In which case the contents of $\mathcal{E}_i$ will make up $\mathcal{R}_\infty$, $i \in [0...n, \infty]$(see algorithm 1 below).

Algorithm 1: BaseRank Algorithm[12]

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

2.3.3 Rational Closure Algorithm. The rational closure determines whether defeasible implication are entailed by a knowledge base using a method that ensures coherent and consistent conclusions by ensuring LM-rational properties are satisfied. Magidor[16]. Rational closure can be defined either semantically or algorithmically, [4, 17, 20, 22].Algorithmcally, the rational closure algorithm works as follows: given a knowledge base $\mathcal{K}$, and a defeasible implication $\alpha \vdash \beta$, it uses the base rank algorithm to rank the statements, then checks if the negation of the antecedent is entailed by the rank, $R_\infty \cup R_i \models \neg\alpha$. If so, then the rank is eliminated, this process is continuous until there is no rank to consider or $R_\infty \cup R_i \models \alpha \rightarrow \beta$, the output in the described outcomes is false and true, respectively [6, 17, 20, 22](See algorithm 2 below).

Algorithm 2: RationalClosure Algorithm [12]

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

2.4 Related Work: Existing Defeasible Knowledge base generators

There are currently three knowledge base generator that have been introduced, the first contribution was by Aiden Bailey[2]. If focused on the construction of defeasible knowledge bases as defeasible clash graphs, which can be converted to defeasible knowledge base consisting of defeasible and classical statements. The defeasible clash graphs, denoted $\mathcal{G}$, where $\mathcal{G} = (A, C, D)$, A being antecedents (atomic propositions), C being a set of direct clash relations, and D being a set of direct defeasible relations[2]. The produced defeasible

knowledge base generators focused on the following parameters, *number of ranks*, *number of statements*, *distribution* (uniform, exponential, normal, inverted normal and inverted exponential), and whether the statements should be *defeasible only*.Furthermore, the conservative method for defeasible knowledge base generation was introduced, it uses a single atomic proposition to bring about clashes or exceptions that build up the defeasible knowledge base structure [2], this used a chained defeasible clash graph(see the algorithm 3 in appendix B). This knowledge base generator does not address the complexity of the generated propositional statements, the complexity allows us to express more detailed information in knowledge bases. The last two introduced knowledge bases introduced by Alec Lang[14] address the complexity issue by allowing users to set complexity of the antecedent and consequent to either 1 or 2. A setting of 2 meaning that the propositional statements will have two connectives and three atomic propositions within the statements. This is not applied on propositional statements that maintain the structure of the defeasible knowledge base, thus limiting them to being atomic propositions instead. Furthermore, there is limited control over which statements are to be complex statements[14]. The newly introduced parameters in this case where *antecedent* and *consequet complexity*, as well as the *connectivity types*, more distributions were used (random, linear incline and linear decline). The difference between the two knowledge base is that one is an optimization of the other, using multiple threads to improve generation time, each thread responsible for generating a rank (see algorithm 4 in appendix B).

2.5 Project Goals

The main objectives of this experimental project are:

- To create a pseudo-random defeasible knowledge base generator for testing the rational closure algorithm and its extension.
- To optimize knowledge base generation by improving the parallel programming component in previously introduced defeasible knowledge base generator, by reusing previously generated knowledge bases (space-time trade-off), and improving the overall architecture.
- To improve the level of expressiveness in generated knowledge bases.
- To investigate and analyse key parameters involved in knowledge base generation and the effect they have in knowledge base generation.

2.6 Materials

2.6.1 Hardware. Laptop with specifications: Device name-LAPTOP-GE9EQ6Q6, Processor Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz, Installed RAM 8,00 GB (7,85 GB usable), and System type 64-bit operating system, x64-based processor.

2.6.2 Software. Package manager and build automation tool - Apache Maven 3.9.8, Database - MongoDB v7.0.5, Visual Studio IDE v1.92.2 and Java v22.0.1.

3 DEFEASIBLE KNOWLEDGE BASE GENERATOR DESIGN

3.1 Defeasible knowledge base generation

3.1.1 Atomic proposition Generation. The atomic propositions are represented by alphabets that can be selected during runtime. The alphabets are either upper Latin, lower Latin, or Greek letters, each associated with a specific Unicode range. The driving force is the number of atoms required to build a defeasible knowledge base. What the previous Knowledge base generator does in the process is choose a Unicode from the selected range and then check if its corresponding character is within the list of already generated atomic propositions, if it's not then added to the list, otherwise, it's discarded. As the atomic propositions list increases the more we discard, this then slows down the atomic proposition generation. In the current generator, the selection process is different because the Unicode is converted to a character or atomic proposition in order, should we get to the end of the range of the Unicode before we have all required atoms, we increment the number of digits or characters representing the atomic propositions. Then create different combinations using an iterative loop, given a Unicode with range *n*, then the combinations would be *nxn*. In this way to not have to discard any atoms and it's much more efficient as opposed to selecting them randomly. This process is mainly handled by the *AtomCreation.java* class.

3.1.2 Simple and Complex propositional formula generation. Creation of propositional formulas or propositional statements requires the use of atomic propositions and connectives and as such one needs to understand how specific connectives work:

- Negation[$\neg$]: This is a unary Boolean operator, meaning that it operates on one element at a time. Given a propositional atom or formula which has an interpretation of true, application of negation operator on it converts it to false, and *vice versa*.
- Conjunction [$\wedge$]: This binary Boolean operator allows us to represent statements that indicate that two things (atomic propositions) must hold for the entire statement to be interpreted as true. In the case where one of the atomic propositions or propositional formulas is false, then the entire statement is interpreted as false.
- Disjunction[$\vee$]: This binary Boolean operator allows us to represent statements that indicate that at least one of the two things (atomic propositions) must hold for the entire statement to be interpreted as true. In the case where none of the atomic propositions or propositional formulas is true, then the entire statement is interpreted as false.
- conditional implication[$\rightarrow$]: This binary operator allows us to represent statements where interpretation of one statement (antecedent) as true implies the other (consequent) is true as well. Statements like these are Interpreted as true when the consequent is true or when the negation of antecedent is true.
- Biconditional implication[$\leftrightarrow$]: This binary operator allows us to represent statements where interpretation of one statement (antecedent) as true implies the other (consequent) is true, and *viceversa*. Statements like these are Interpreted

as true when both the consequent and antecedent are true or false.
- Defeasible implication[$\sim$]: This is a binary operator, mainly used to represent a weakened conditional implication, in the sense that one thing implies another in most cases, "typically", but not always the case.

These Connectives are used to create complex propositional statements used in the generation of the knowledge base to represent information in different forms. The conditional and biconditional implications are only used to create classical statements (making up the infinity rank), and so is the defeasible implication only used in defeasible statements, making up rank 0 to n, for a defeasible knowledge base with n ranks. Simple statements have a complexity of one on both antecedent and consequent, meaning they are made only of atomic propositions and use either the conditional implication or defeasible implication depending on the rank, $\alpha \rightarrow \beta$. Complex statements on the other hand are dependent on the atoms, antecedent complexity, consequent complexity, and selected connectives. Overall both the simple and complex's antecedent and consequent are created using the same function, the $getFormula()$ in the *KBGeneratorThreadedOPT.java* class, that takes as input the list of usable atoms, connectives, and complexity(*antecedent complexity or consequent complexity*). The resulting antecedent and consequent are then combined to form a propositional formula using a defeasible implication or a conditional implication depending on the rank. The $getFormula()$ uses an iterative loop constrained by the complexity to build the formulas, it randomly selects the number of atoms and connectives and combines them accordingly. Note that the complexity allows us to express more detail in one formula but is limited by the atoms and connectives, e.g $\alpha \vee \alpha$ has a complexity of 2 (atom count), and α has a complexity of 1, but share the same interpretation.

3.1.3 Distribution. This knowledge base generator produces defeasible knowledge bases that have statements distributed as per user selection, namely, the flat or uniform, linear incline, linear decline, exponential incline, exponential decline, random, and normal distribution. This is done using the number of ranks and the distribution, calculations are made to find the minimum number of statements needed to make up the distribution given the rank. This is an adoption of Lang's $Distribution.java$[14], with additional methods, *minDIsExp* and *minDIsNormal* for calculating minimum values; *distributeDIsExpDecline*, *distributeDIsExpIncline*, and *distributeNormal* for distributing number of statements per rank; and lastly the *getNewDistribution* method for finding a new distribution given two distributions. This method is used when an old knowledge base is used to create a new one.

3.1.4 Construction of the defeasible knowledge base. The current defeasible knowledge base generator constructs a defeasible knowledge base using a conservative technique, it uses one atom to create a chain of exceptions. It uses threads to speed up generation Like Lang's optimized generator, the difference is the current generator uses threads for specific tasks and for generating statements in ranks as opposed to just for the ranks (Threads not shown in algorithm 5). How the generator works is it assigns a thread to create a list of atoms used to create propositional statements, another thread to create defeasible statements that hold the structure of the ranks, another thread for creating the remaining defeasible Statements, and another thread for creating the classical Statements. A locking mechanism is used to avoid any bad interleaving or errors since the different threads are working on one defeasible knowledge, using one atom list. While the atom creation thread is still creating atoms, the thread responsible for building a chain of exceptions can use the existing atoms to build the defeasible implication that makes up the chain if there are enough atoms to build a complete defeasible implication, considering the complexity of the antecedent and consequent, the defeasible implication is constructed, otherwise the thread waits. After adding the resulting defeasible implication to the appropriate rank in the knowledge base it sets the completions status to true. The thread that adds the remaining defeasible statements uses the completion status to confirm if it should continue or wait, the thread branches to multiple threads to fill the ranks. The thread that builds classical statements checks if there are enough atoms to create the statement. So the overall knowledge base generation process is, that we create the atomic propositions, and we use them to create the defeasible statements that form the structure of the knowledge base, that is, those that maintain the number of ranks. When we do this, we decrease the number of statements in the distribution to show that a specific number of statements have already been created. This way we can maintain the distribution that was selected and the required statement. Then we fill in the remaining defeasible statements while decrementing the distribution accordingly and maintaining the rule that every defeasible statement created should contain the antecedent that forms the rank. Otherwise, if this is not done, what happens is that the distribution collapses. This process is illustrated using algorithm 5 in appendix A and is handled by the *KBGeneratorThreadedOPT.java* class through a *KBGenerate* method.

Transitivity was added as part of the parameters that are considered when creating a defeasible knowledge base. This parameter is focused on creating transitive statements within the same rank. This particular parameter was added to provide reasoning algorithms a platform to test statements that are not explicitly stated, for instance, given $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$ can the reasoners deduce that $\alpha \rightarrow \gamma$ as well? This was done taking into consideration that statements within defeasible ranks have been materialized. To create one transitive relation that can be extended within a rank one has to add four statements, among the four statements there should be a material implication relation between the antecedent that forms a rank and another atom. The other atom should have a material implication relation with the atom causing exceptions in the state in which it is appropriate in the current rank (The atom changes states based on rank, given α is the exception atom it can be α or $\neg\alpha$ depending on the rank, see rank R_m in figure 1 below), and the same atom must have a material implication relation with the antecedent forming the rank above, and lastly the atom must have a material implication relation with a new atom. To extend the transitive relation the new atom has to form material implication relations with the atom causing exceptions, the antecedent forming the rank above, and a relatively new atom, this process is continuous until a desired transitivity level is achieved. This is a rather conservative approach, a non-conservative approach may use different exception atoms to form the transitions.

Rank	Defeasible statements in rank
$\vdots$	$\cdots$
R_{m-1}	$\lambda \rightarrow \gamma, \lambda \rightarrow \neg\alpha \cdots$
R_m	$\beta \rightarrow \lambda, \beta \rightarrow \alpha, \beta \rightarrow \omega, \omega \rightarrow \lambda, \omega \rightarrow \alpha, \omega \rightarrow \tau \cdots$
R_{m+1}	$\cdots$
$\vdots$	$\cdots$

Figure 1: A a ranked defeasible knowledge base illustrating transitivity

The pseudo-code below gives a high-level explanation of how the defeasible knowledge base generator constructs the defeasible knowledge bases (for a more detailed algorithm see Algorithm 5 in Appendix A).

Generator Pseudo – code

1. Get parameters: *numformulas*,*classic-defeasible ratio*, *Ranks*, *distribution*,*transitivity*, *reuse consequent*, *anteComplexity* and *consComplexity*.
2. Calculate *Defeasible implications* and *classical statements* using *classic-defeasible ratio*.
3. Create an array of size *Ranks* and allocate *Defeasible implications* quantities into the array as per *distribution*.
4. Generate *atoms* based on the *numformulas*
5. Build a defeasible Knowledge base structure using *atoms*, based on *anteComplexity* and *consComplexity*.
6. Fill the structure with the missing defeasible implications according to the *distribution*, using *atoms* and based on *anteComplexity* and *consComplexity*.
7. generate and append classical statements to the structure using *atoms*, based on *classical statements*, *anteComplexity* and *consComplexity*.

3.2 Optimization Techniques

3.2.1 Space-Time Trade-off. This kind of optimization is aimed at improving generation time by using defeasible knowledge bases that have been created in the past, It uses the database to maintain the previously generated knowledge bases and the control parameters that were used in the creation process. So, this means that some control parameters are already fixed and cannot be changed, except for the number of ranks and the number of defeasible statements that have to be greater than those of the knowledge base that is being used in the generation process. The process involves finding a new distribution, finding the difference between the two distributions (old and new), and using the resulting distribution to guide the generation process. The atoms are checked to see if they are enough, If not, new atoms are added to the list. The generation of the structural defeasible Implications starts where the previous knowledge base ended, the generation of the remaining defeasible statements sweeps through the entire distribution checking to see if, each rank is already complete and completing the rank if necessary (see Algorithm 6).

Generator Pseudo – code by existing knowledge base

1. Select a knowledge base to use from the database.
2. Get parameters through user input: *numformulas* and *Ranks*.
3. Get parameters from database.*classic-defeasible ratio*, *Ranks*, *distribution*,*transitivity*, *reuse consequent*, *usedAtoms*, *remainingAtoms*, *anteComplexity* and *consComplexity*.
4. Calculate *Defeasible implications* and *classical statements* using *classic-defeasible ratio*.
5. Create an array of size *Ranks* and allocate *Defeasible implications* quantities into the array as per *distribution*, and existing knowledge base.
6. Generate *atoms* based on the *numformulas* and *remainingAtoms*
7. get the *conflictAtom*, atom causing exceptions in existing knowledge base.
8. Build a defeasible Knowledge base structure using *atoms*, based on *anteComplexity conflictAtom*, and *consComplexity*.
9. Insert the structure between the classical and defeasible ranks of the existing knowledge base.
10. Fill the entire structure with the missing defeasible implications according to the *distribution*, using *atoms* and based on *anteComplexity* and *consComplexity*.
11. Add classical statements to the classical/infinity rank, statements created using *atoms*, based on *classical statements*, *anteComplexity* and *consComplexity*.

3.2.2 Parallelization By Task. Multi-threading was done to reduce the execution time of the generator.

3.2.3 Atomic Proposition Generation Optimization. This form of optimization also aims to improve generation time by eliminating unnecessary operations that end up producing atoms that are already created.

3.2.4 Complex Proposition Generation Flexibility. This form of optimization allows us to have propositional statements that are more complex thus allowing us to express more details even though there are limitations to what you can express. It allows us to specify our complexity for both the antecedent and consequent, instead of being limited to 3 atoms and 2 connectives in an antecedent or consequent.

3.3 Defeasible Knowledge base generator features

The current knowledge base retains features from the previously optimized knowledge base generator, those which are for convenience and those that play a crucial role in defeasible knowledge base generation (control parameters).The convenience features being the command-line interface allowing the control parameters to be supplied via a text file, printing knowledge base on screen, saving it to a text file so it can be used by reasoners, character set selection, modification of adjustable Connective Symbols, and the option to regenerate, change parameter settings and quit[14]. The control parameters still in use are the Generator Type, Number of

Ranks, Distribution, number of Defeasible Implications, Reuse of Consequent, Antecedent complexity, Consequent complexity, and Connective Types, the Simple Only is excluded since it is directly linked to the complexity. [14].

3.3.1 Generator Type. This feature allows us to select From the different types of defeasible Knowledge base generators, the options being the currently optimized generator, and the generator that reuses old knowledge bases.

3.3.2 Distribution. Though the distribution is not a new feature, It has been extended to include a range of distributions to consider when creating a defeasible knowledge base. The distributions that are in use in this version of the defeasible knowledge base generator are Uniform or Flat, Random, Linear growth, Linear decline., Exponential growth, Exponential decline, and normal distribution.

3.3.3 Progress bar. Considering that in some cases, extremely large defeasible knowledge bases will be created, this feature is important since it allows us to see the progress in terms of the percentage value and a horizontal bar. To some extent giving an estimation of how long it would take to create the defeasible knowledge base or whatever is remaining of it.

3.3.4 Construction by reusing other knowledge bases. To promote efficiency existing defeasible knowledge bases are used to create new ones, this allows us to save time during generation. This feature works hand in hand with the defeasible knowledge base log which shows previously generated defeasible knowledge bases eligible for reuse. However, this is done with some constraints relative to the control parameters, the number of ranks and statements have to be greater than the previously generated knowledge base, and the remaining parameter settings remain unchanged.

3.3.5 Defeasible knowledge base log. This feature displays all divisible knowledge bases that have been created and saved in the database. This is to prevent unnecessary creation of defeasible knowledge bases and to provide selection details for defeasible knowledge base reuse.

3.3.6 Defeasible knowledge base testing. This feature allows us to test the generated defeasible knowledge base to see if it has the correct structure of a defeasible knowledge base by checking to see if the base rank can reproduce a defeasible knowledge base with the same number of ranks and possible number of statements per rank.

3.3.7 Saving in Database. This feature allows one to save the defeasible knowledge base, along with its associated parameters in a database.

3.3.8 Transitivity. To create a diverse set of relations in the defeasible knowledge base. The transitivity can be set to No for no transitivity amount statements, Yes meaning some statements in a rank should have a transitive relation to each other, or Random meaning within a rank some statements may have a transitive relation.

3.3.9 Classical and defeasible statement control. This controls the amount of defeasible propositional statements relative to classical ones by using a ratio, 'classical': 'defeasible'.

4 TESTING

To ensure that the created defeasible knowledge base generator works appropriately, a couple of tests were conducted, which are the correctness test, and the generation time test, and there was an analysis of the parameters used in the generation process. However, not all features could be quantitatively tested such as expressiveness, the use of transitivity, complexity, and formula formatting to express more or specific details in the knowledge base.

4.1 Correctness testing

This test checks whether the created defeasible statements can be used to create a defeasible knowledge base of the same number of rankings, statements, and distribution as it is expected to have. This is done using the base rank algorithm used in the SCADR project[21]. A total count of the defeasible knowledge bases (DKB) is kept, along with the pass and fail count base rank tests, the values are then used to calculate the Defeasible knowledge base *error rate* and *accuracy*. Pass means the exact amount of ranks is created and otherwise fails.

$$\textit{Error Rate} = \frac{\text{Number of fails}}{\text{Total Number of DKBs}} \times 100\%$$

$$\textit{Accuracy} = \frac{\text{Number of passes}}{\text{Total Number of DKBs}} \times 100\%$$

4.2 Generation time test

The generation time test compares the time taken to generate a defeasible knowledge base, produced by Lang's Optimised defeasible Knowledge Base Generator against the proposed defeasible Knowledge Base Generator, with the same control parameters. This evaluation will allow us to see which knowledge base performs better. The experiment is done by generating defeasible knowledge bases, with specific parameter settings and recording the generations for both Lang's generator and mine. For each parameter setting, the experiment is conducted n times to avoid bias and to eliminate errors.

4.3 Space-time trade-off optimization test

This test is aimed at evaluating how effectively reusing previously generated defeasible knowledge bases reduces generation time. For this test, the generator that generates knowledge bases without reusing other knowledge bases is compared to one that reuses the knowledge base, to see how long it takes to create a knowledge base of the same specification. The one that uses knowledge bases to create others will use one knowledge base as a reference, this will create a bounding region of performance for the creation of knowledge bases using subsequent knowledge bases produced by the reference. During the experiment the generation time and storage will be recorded, this will show the relation between time and space.

The size of the data that is stored in the Database is calculated using the Java standard data types size, int (4 bytes), Boolean (1 byte), char (2 bytes), long (8 bytes), double (8 bytes), Arrays have a size of 16 bytes, and a string of length n has the size n*2 bytes since its a combination of characters[8]. Once the calculation of the stored data size is done a conversion from bytes to MBs is done by dividing by 1 048 576 as per IEC units[10].

The data stored is an array of the knowledge base, the number of ranks, the type of generator used, the Generation time, a list of used atoms, a list of remaining atoms, the distribution, the number of defeasible implications, the used character set, a list of connectives, whether there are transitive statements, and whether consequents where reused.

4.4 Parameter analysis testing

Parameters are analyzed using the Monte Carlo simulation, it is used to understand how the different parameters affect the performance of the knowledge base generator. The parameters are the number of ranks, number of defeasible statements, distribution of statements, connectives, transitivity, antecedent complexity, consequent complexity, minimum statements per rank, and the reuse of consequent. The experiment is conducted by generating n different control parameters, generating the corresponding knowledge bases, and recording the generation time. Then using statistical analysis such as Correlation analysis and Sensitivity Analysis, we evaluate the effect of parameters on the generator's performance.

4.4.1 Correlation Analysis. Correlation analysis helps us understand the relation between individual parameters and the generation time, a correlation (r), of 0 meaning no relation between the parameter and the generation time, 0 to +1 indicating an increase in parameter increases the generation time and, 0 to -1 indicating an increase in parameter decreases the generation time. The equation below is used to calculate correlation, where n is the number of data points, X represents the parameter being evaluated, and Y is the generation time.

$$r = \frac{n(\sum XY) - (\sum X)(\sum Y)}{\sqrt{[n\sum X^2 - (\sum X)^2][n\sum Y^2 - (\sum Y)^2]}}$$

4.4.2 Sensitivity Analysis. The sobol Sensitivity analysis is used to determine which parameters highly affect the generator's performance and how the parameter interacts with other parameters. It does so using the first-order Sobol index S_i and the total Sobol index S_{Ti}. S_i shows how the individual parameter contributes to the variance of the variance of the output (generation time) and the S_{Ti} shows the total effect of the parameter on the output, directly and through interactions with other variables.

$$V_{X_i} = \frac{1}{n-1}\sum_{j=1}^{n}(Y_{X_i}^{(j)} - \overline{Y_{X_i}})^2$$

The variance of the output(generation time) $Var(Y)$ is used in the calculation of S_i and S_{Ti}.

$$S_i = \frac{V_{X_i}}{V},\ where\ V = Var(Y)$$

$$S_{Ti} = 1 - \frac{V - V_{X_i}}{V}$$

5 RESULTS

5.1 Correctness

The test was conducted on 10 different generated defeasible knowledge bases, with a different number of ranks, statements, antecedent and consequent complexity, and reuse of consequent Values. 10 out of 10 Defeasible knowledge bases passed, resulting in an error rate of 0% and accuracy of 100% for the 10 knowledge bases with parameters specified in Appendix C.

5.2 Generation time test

The values used to create the graphs below are in Appendix C.

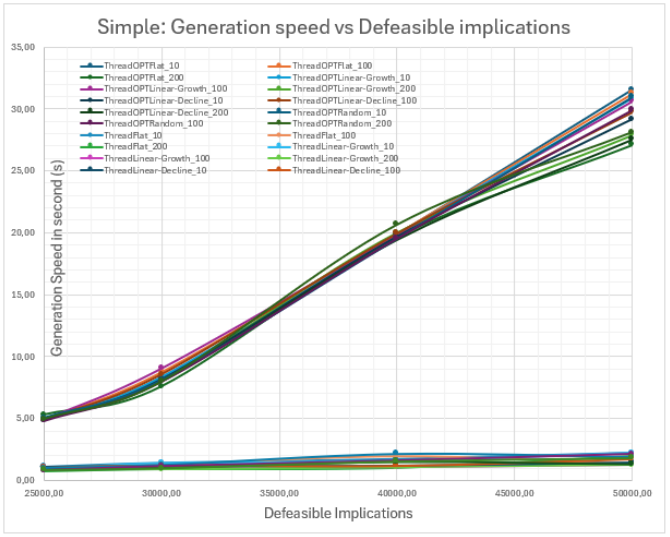

Figure 2: Comparison between the proposed knowledge base and Lang's generation time on simple statements

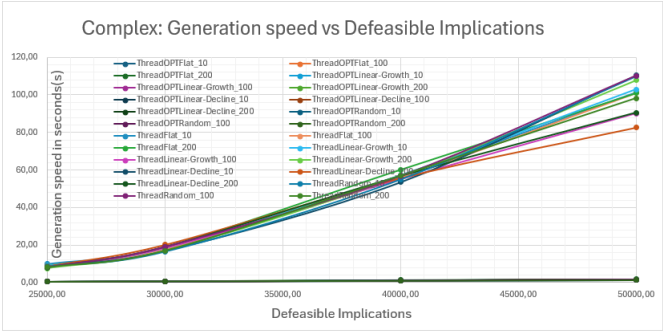

Figure 3: Comparison between the proposed knowledge base and Lang's generation time on complex statements

5.3 Space-time Trade-off optimization test

These are the results obtained when conducting this space-time trade-off test they are used to create the graph below. The test was conducted on simple defeasible implications.

Space-Time Trade-off results					
Ranks	Defeasible Implications	new (s)	reuse (s)	space (MBs)	$\Delta Time$ (reuse-new)
10	1000	0.0634	0.0342	0.136	-0.029
50	5000	0.2521	0.0569	0.352	-0.195
100	10000	0.6551	0.0998	1.506	-0.555
200	25000	5.4859	0.1588	4.416	-5.327
300	50000	26.1439	0.6898	10.776	-25.454

Table 1: Space-Time Trade-off results

Correlation per parameter(r)	
Ranks	0.040
Defeasible Implications	0.120
Minimum statement per rank	0.135
Reuse Consequent	-0.058
Transitivity	0.017
Antecedent Complexity	0.062
Consequent Complexity	-0.125
Connectives	0.119

Table 2: The correlation between the different parameters and the generation time

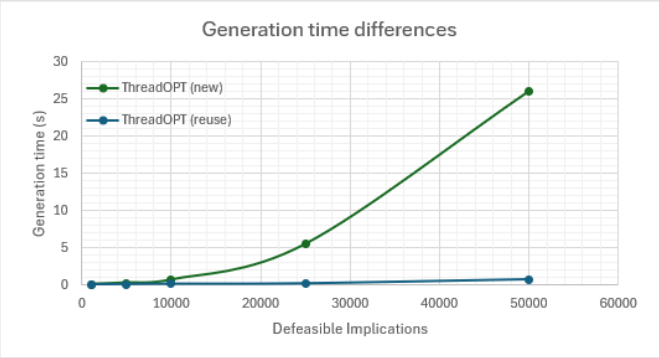

Figure 4: comparing generating time of the proposed generator when it creates a new knowledge base from scratch to when it uses an existing defeasible knowledge base

Correlation per distribution (r)	
Exponential decline	0.353
Exponential growth	0.517
Flat	0.278
Linear decline	0.760
Linear growth	0.241
Normal	0.459
random	0.267

Table 3: The correlation between the number of defeasible Implications and the generation time for a specific distribution

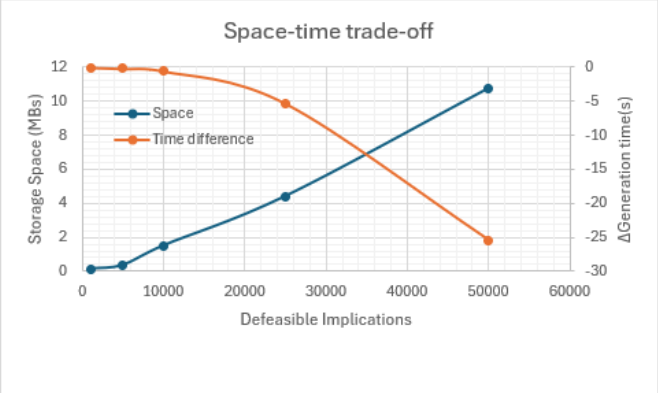

Figure 5: The trade-off between time and space when using existing knowledge bases as opposed to creating entirely new ones

5.4 Parameter analysis testing

5.4.1 Correlation Analysis. The Monte Carlo Simulation was conducted over 200 knowledge base generations, the input parameters that yield the resulting correlation can be found in Appendix D.

5.4.2 Sensitivity Analysis. The results could not be provided at this moment due low number of Monte Carlo Simulation trials and the random parameter selection. To calculate the first-order Sobol index S_i and the total Sobol index S_{Ti} for any parameter, its variance must be calculated on values where it's varied while other parameters are constant, and for these many parameters many trials (knowledge bases would be required).

6 DISCUSSION

6.1 Generation time test

Looking at the generation time test, we get to see that as the number of defeasible statements increases so does the generation time for all distributions. However, when looking at the difference between the outcome for complex and simple statements generations process, we can see that Lang's Generator is much faster when generating simple statements as opposed to the proposed generator. The reason for this is suspected to be the synchronization mechanism used in the generation of antecedents and consequents, for simple atoms the generating thread is expected to wait before removing atoms from the atom list due to fluctuations in the atom list size, this was done to minimize concurrency issues. This is not the case for the complex statements as they can reuse atoms within the same statements, so the level of fluctuations is minimal.

6.2 Space-time Trade-off optimization test

When looking at the space-time trade-off test, we can see that generating a knowledge base from scratch takes more time than using an existing one (see Figure 4). However, the decrease in generation

time causes an increase in the storage space needed (see Figure 5). The increase in space can be reduced by saving all the knowledge base over the one used to create it. But the problem with that is that the more you do that, you eliminate the knowledge bases that you would need to create other knowledge bases.

6.3 Parameter Analysis

6.3.1 Correlation Analysis. Based on the observed results there are no individual parameters that have a strong correlation with generation time. All parameters with the exception of the Reuse Consequent and Consequent Complexity, minimally increase the generation time by increasing, with the leading parameter being the minimum statements per rank given at r= 0.135. Not reusing consequent and decreasing consequent complexity increases generation time, the consequent complexity having the lowest correlation value, r= -0.125. Note the transitivity and the reuse of consequents were converted to 1 and 0 for the calculation of the correlation since they are boolean variables. The Connectives correlation was calculated using the number of connectives used in the knowledge base generation, if connective "1" or "2" were used that counts as one connective and otherwise 2. As expected the increase in the minimum number of statements/ defeasible implications per rank and number of defeasible implications increases the generation time, and not reusing the consequent increases the generation time. However, the complexity of both the consequent and antecedent were expected to increase the generation time, and this is not the case. Another, concerning issue is the weak correlation between the parameter and the generation time. The reason for all this is suspected to be the low number of trials (knowledge bases produced) of the Monte Carlo Simulation.

The correlation between a distribution and the generation time was conducted by calculating the correlation between the number of defeasible implications and the generation time for a specific distribution, this is because the distribution affects the number of defeasible implications required to produce the rank. As expected the flat and random distributions have a low to moderate correlation with the generation time, this is because for a multi-thread generator like this, which has some threads working on some ranks, the threads are likely to take around the same time to finish construction of the knowledge base, meaning an increase in the statements the flat distribution gradually increases the generation time. The random distribution may have impulsive increases along the ranks, but they will not be as dense as the other distributions. The remaining distributions are expected to have a strong correlation with the generation time due to defeasible implications being located more in one area (less number of threads work on more defeasible implications). On the contrary, only the Linear-decline has a somewhat strong correlation r= 0.760, and the exponential trailing it at r= 0.517. This as explained above is suspected to be due to the low number of Monte Carlo Simulation trials.

7 CONCLUSIONS

The goal of the project was to create an optimized pseudo-random defeasible knowledge base generator, that can express more detail, and study the parameters used in the knowledge base generation process. As a result, the generator produced can express more information through the extended complexity of the antecedent and consequent. The knowledge base generator created also creates defeasible knowledge bases with complex statements in a relatively optimal time compared to the existing generator. As for the generation of simple statements a better concurrency management system should be implemented. The study of parameters at this point did not give affirmative conclusions and would benefit of a lot from the generation of many defeasible knowledge bases or Monte Carlo Simulation trials in the future. Hence, this part of the project was a partial success.

8 FUTURE WORK

In terms of future work. I recommend the creation of non-deterministic knowledge bases. Based on some context or language, For instance, the English language. This may require learning and understanding fundamental concepts in the English language, verbs, adverbs, adjectives, etc. This would also require understanding which parts of the language can be translated to propositional statements or atomic propositions and connectives. This type of defeasible knowledge base generator would allow us to represent the world more realistically, which would allow the development of defeasible reasoners that best suit the environment. With it, we may learn some interesting patterns within the data that can be used to improve the previously generated knowledge bases. On the pseudo-random defeasible knowledge base generator side of things I recommend a propositional formula formatting feature that allows one to define the kind of propositional formulas required in the knowledge base as opposed to selecting the type of connectives to use and the complexity of the antecedent and consequent. This is particularly essential when one wants to test reasoners on defeasible knowledge bases with statements of a specific format to see if the appropriate conclusion is made given a query.

REFERENCES

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

[2] Aidan Bailey. 2021. Scalable Defeasible Reasoning. Project Paper.

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

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

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

[6] Michael Freund. 1998. Preferential reasoning in the perspective of Poole default logic. *Artificial Intelligence* 98, 1 (1998), 209–235. https://doi.org/10.1016/S0004-3702(97)00053-2

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

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2021. *The Java Language Specification, Java SE 17 Edition.* Addison-Wesley. https://docs.oracle.com/javase/specs/jls/se17/html/index.html Accessed: 2024-09-08.

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

[10] International Electrotechnical Commission. 2023. Prefixes for binary multiples. https://www.iec.ch/prefixes-binary-multiples Accessed: 2024-09-08.

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

[12] Adam Kaliski. 2020. *An Overview of KLM-Style Defeasible Entailment.* Master's Thesis. Faculty of Science, University of Cape Town.
[13] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence* 44, 1-2 (1990), 167–209. https://doi.org/10.1016/0004-3702(90)90101-5
[14] Alec Lang. 2023. Extending Defeasible Reasoning Beyond Rational Closure. Project Paper.
[15] Daniel Lehmann. 1995. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence* 15, 1 (01 03 1995), 61–82. https://doi.org/10.1007/BF01535841
[16] Daniel Lehmann and Menachem Magidor. 1992. What does a conditional knowledge base entail? *Artificial Intelligence* 55, 1 (1992), 1–60. https://doi.org/10.1016/0004-3702(92)90041-U
[17] Maqhobosheane Mohlerepe. 2024. Extending Defeasible Reasoning Beyond Rational Closure. Literature review.
[18] Maqhobosheane Mohlerepe. 2024. Investigating Optimisation Techniques for Rational Closure in Defeasible Reasoning. BSc Honours Project.
[19] Thabo Vincent Moloi. 2024. Extending Defeasible Reasoning beyond Rational Closure. BSc Honours Project.
[20] Thabo Vincent Moloi. 2024. Extending Defeasible Reasoning beyond Rational Closure. Literature Review.
[21] Evashna Pillay. 2022. An Investigation into the Scalability of Rational Closure V2. Honours Project.
[22] Mamodike Sadiki. 2024. A Literature Review on Extending Defeasible Reasoning Beyond Rational Closure. Literature review.
[23] Yoav Shoham. 1987. A semantical approach to nonmonotonic logics. In *Logic in Computer Science*. https://api.semanticscholar.org/CorpusID:117644704
[24] Luke Slater and Thomas Meyer. 2023. Extending Defeasible Reasoning Beyond Rational Closure. In *Artificial Intelligence Research*, Anban Pillay, Edgar Jembere, and Aurona J. Gerber (Eds.). Springer Nature Switzerland, Cham, 151–171.

Appendix A: Knowledge base generation algorithms

Algorithm 5: KBGeneratorThreadedOPT.KBGenerate

Input :Distribution, numClassical, numDefeasible,
transitivity, antComplexity, consComplexity,
connectives, reuseConsequent, charSet
Output:A defeasible knowledge base $\mathcal{K}$, $usedAtoms$, $atomList$

$\mathcal{K} = newArrayList < ArrayList < String >> ()$
$atomList = newArrayList < String > ()$
$usedAtoms = newArrayList < String > ()$
$completionStatus = newAtomicBoolean[Distribution.length]$
$setCharacters(charSet)$
$infinityCons = separateCon(connectives, "infinity")$
$defeasibleCons = separateCon(connectives)$
$generatAtoms(atomList, atomCount(numClassical + numDefeasible))$
$consequent = \emptyset$
$antecedent = \emptyset$
$j = 0 \;\&\; i = 0$
for $j < Distribution.length$ ***do***
  while$(atomList.size < antComplexity + consComplexity)$
    $\#wait$
  if $j == 0$ ***then***
    $cosequent = getFormula(consComplexity, defeasibleCons, atomList, usedAtoms)$
    $antecedent = getFormula(antComplexity, defeasibleCons, atomList, usedAtoms)$
    $KB.add(defeasibleImplication(antecedent, consequent))$
    $Distribution[j] = Distribution[j] - 1$
  else
    $prevAntecedent = antecedent$
    $consequent = negation(consequent)$
    $antecedent = getFormula(antComplexity, defeasibleCons,$
    $atomList, usedAtoms)$
    $KB.add(defeasibleImplication(antecedent, consequent))$
    $KB.add(defeasibleImplication(antecedent, prevAntecedent))$
    $Distribution[j] = Distribution[j] - 2$
  $completionStatus[j].set(true)$
  $j++$
if $distributionIncomplete(Distribution)$ ***then***
  for $i < Distribution.length$ ***do***
    while $completionStatus[i] == false$
      $\#wait$
    $rankAtom = getRankAtom(Distribution[i]])$
    while $Distribution[i] > 0$ ***do***
      $cosequent = getFormula(consComplexity, defeasibleCons, atomList, usedAtoms, rankAtom, transitivity, reuseConsequent)$
      $antecedent = getFormula(antComplexity, defeasibleCons, atomList, usedAtoms, rankAtom, transitivity, reuseConsequent)$
      $KB.add(defeasibleImplication(antecedent, consequent))$
      $Distribution[i] = Distribution[i] - 1$
    $i++$
$generateClassicalStatements(infinityCon, consComplexity, antComplexity, atomList, usedAtoms)$
return $(\mathcal{K}, usedAtoms, atomList)$

Algorithm 6: KBGeneratorThreadedOPT.newKBGenerate

Input :Distribution, knowledge base $\mathcal{K}$, atomList, numClassical, numDefeasible, transitivity, charSet, usedAtoms, connectives, reuseConsequent

Output:A defeasible knowledge base $\mathcal{K}$, $usedAtoms$, $atomList$

```
completionStatus = newAtomicBoolean[Distribution.length]
antComplexity = getComplexity(KB,"antecedent")
consComplexity = getComplexity(KB))
infinityCons = separateCon(connectives,"infinity")
defeasibleCons = separateCon(connectives)
count = numClassical + numDefeasible
if atomCount(count) > atomList.size do
    setCharacters(charSet)
    numAtoms = atomCount(count) − atomList.size
    generatAtoms(atomList, numAtoms)
consequent = ∅
antecedent = ∅
j = KB.size & i = 0
for j < Distribution.length do
    if j == KB.size then
        while(atomList.size < antComplexity + consComplexity)
            #wait
        if j == KB.size then
            prevAntecedent = getAntedent(KB)
            cosequent = getConsequent(KB)
            antecedent = getFormula(antComplexity, defeasibleCons, atomList, usedAtoms)
            KB.add(defeasibleImplication(antecedent, consequent))
            KB.add(defeasibleImplication(antecedent,
               prevAntecedent))
            Distribution[j] = Distribution[j] − 2
        else
            prevAntecedent = antecedent
            consequent = negation(consequent)
            antecedent = getFormula(antComplexity, defeasibleCons, atomList, usedAtoms)
            KB.add(defeasibleImplication(antecedent, consequent))
            KB.add(defeasibleImplication(antecedent, prevAntecedent))
            Distribution[j] = Distribution[j] − 2
        completionStatus[j].set(true)
    else
        completionStatus[j].set(true)
    j++
if distributionIncomplete(Distribution) then
    for i < Distribution.length do
        while completionStatus[i] == false
            #wait
        rankAtom = getRankAtom(Distribution[i]])
        while Distribution[i] > 0 do
            cosequent = getFormula(consComplexity, defeasibleCons, atomList, usedAtoms, rankAtom, transitivity, reuseConsequent)
            antecedent = getFormula(antComplexity, defeasibleCons, atomList, usedAtoms, rankAtom, transitivity, reuseConsequent)
            KB.add(defeasibleImplication(antecedent, consequent))
            Distribution[i] = Distribution[i] − 1
        i++
generateClassicalStatements(infinityCon, consComplexity, antComplexity, atomList, usedAtoms)
return (K, usedAtoms, atomList)
```

Appendix B: Previous knowledge base generation implementation algorithms

Algorithm 3: ConservativeRankedGenerate [2]

INPUT: A number of ranks r, total number of statements s, defeasible only Boolean $defeasibleOnly$ and a distribution function d.
OUTPUT: A defeasible knowledge base $\mathcal{K}$.

```
A := ∅
C := ∅
D := ∅
if defeasibleOnly then
    lowerBound := 2
else
    lowerBound := 1
j := 0
while j < r do
    A := A ∪ {α_j}
    if j > 0 then
        C := C ∪ {(α_j, α_{j−1})}
    if r = 1 or j = 0 then
        i := 1
    else
        i := 2
    while i < max(d(s, r, j + 1), lowerBound) do
        A := A ∪ {δ_{j,i}}
        D := D ∪ {(δ_{j,i}), α_{j−i})}
        i := i + 1
    j := j + 1
K := ConservativeDCGKB((A, C, D), defeasibleOnly)
return K
```

Algorithm 4: KBGeneratorThreaded.KBGenerate[14]

Input : dIDistribution, simpleOnly, complexityAnt, complexityCon, connectiveType
Output:A defeasible knowledge base $\mathcal{K}$

```
executor = Executors.newFixedThreadPool(numThreads)
K = newLinkedHashSet < LinkedHashSet <
DefImplication >> ()
anyRankAtoms = newArrayList()
rankBasedCons = generateAtoms()
rankBasedAns = newAtom[dIDistribution.length]
try :
    futures = newArrayList()
    forrank = 0; rank < dIDistribution.length; rank + +
    do
        future = executor.submit(()− >
        generateRank(rank, dIDistribution,
        simpleOnly, complexityAnt, complexityCon,
        connectiveType, rankBaseCons, rankBaseAnts,
        anyRankAtoms))
        futures.add(future)
    for future : futures do
        KB.add(future.get())
catch InterruptedException | ExecutionException e :
    e.printStackTrace()
finally :
    executor.shutdown()
for set : K do
    if!firstSetProcessedthen
        firstSetProcessed = true
        continue
    set.add(newDefImplication(rankBaseAnts[i],
    newAtom(rankBaseAnts[i − 1])))
    i + +
return K
```

Appendix C: Correctness and generation time test parameters

Correctness test parameters				
Ranks	numStatements	Distribution	reuseConsequent	Pass/Fail
10	60	Normal	Yes	Pass
20	300	Exponential-Decline	Yes	Pass
30	1500	Exponential-growth	No	Pass
50	1500	Linear-growth	No	Pass
80	3500	Linear-decline	Yes	Pass
100	900	Normal	No	Pass
130	900	Flat	No	Pass
150	1000	Random	Yes	Pass
200	1000	Flat	No	Pass
300	600	Flat	No	Pass
The correctness test was done with the complexity and antecedent complexity fixed at 1				

Generation time test parameters with recorded time in seconds								
25000 Simple Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	4.955	4.859	4.903	5.062	1.102	0.967	1.093	0.985
100	4.910	4.900	4.889	4.833	0.881	1.046	1.049	0.830
200	5.286	4.872	4.929	4.966	0.788	1.006	0.893	0.847
30000 Simple Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	8.300	8.328	8.179	8.092	1.413	1.409	1.177	1.237
100	8.763	9.057	8.581	8.006	1.199	1.019	1.240	1.153
200	7.574	8.004	8.044	7.994	0.955	0.901	1.003	0.998
40000 Simple Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	19.827	19.780	19.683	19.524	1.720	1.572	1.191	2.169
100	19.941	19.851	19.955	19.556	2.001	1.086	1.202	1.605
200	19.641	20.009	19.440	20.652	1.057	1.059	1.466	1.547
50000 Simple Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	31.539	30.829	29.159	30.957	2.205	2.210	1.460	1.911
100	31.256	30.637	29.748	29.921	1.322	1.280	1.725	2.133
200	27.108	27.893	27.519	28.133	1.977	1.258	1.280	1.833
The generation time test was done with the complexity and antecedent complexity fixed at 1, the values recorded are an average of 5 readings.								
*The readings on the left belong to the proposed generator, and on the right to Lang's threaded generator.								

Generation time test parameters with recorded time in seconds								
25000 Complex Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	0.489	0.581	0.529	0.489	9.942	9.134	8.695	8.595
100	0.508	0.525	0.485	0.504	9.112	8.591	8.652	8.314
200	0.480	0.454	0.491	0.504	9.052	7.646	8.100	8.163
30000 Complex Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	0.603	0.679	0.738	0.702	18.522	18.573	16.990	16.527
100	0.640	0.653	0.603	0.679	18.001	18.421	20.123	19.201
200	0.730	0.716	0.631	0.755	19.021	17.032	19.010	17.017
40000 Complex Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	1.059	1.105	1.051	1.200	57.091	57.250	53.489	55.001
100	1.168	1.118	1.079	1.085	55.923	55.041	55.529	56.790
200	1.040	1.125	1.103	1.079	60.117	56.699	56.721	56.920
50000 Complex Defeasible Implications								
Ranks	Flat	Linear-Growth	Linear-Decline	Random	Flat	Linear-Growth	Linear-Decline	Random
10	1.480	1.511	1.538	1.414	101.311	102.941	110.365	110.009
100	1.316	1.589	1.665	1.695	100.952	90.118	82.480	110.125
200	1.797	1.579	1.597	1.447	100.903	107.825	90.510	98.137
The generation time test was done with the complexity and antecedent complexity fixed at 2, the values recorded are an average of 5 readings.								
*The readings on the left belong to the proposed generator, and on the right to Lang's threaded generator.								

Appendix C: Classes used in the application

App.java – Class allows iteration with the end user, gather control parameters from the user.
AtomCreation.java - The class is responsible for creating atom for the Knowledge base.
BaseRankThreaded.java- Class used to test if the generated knowledge base is defeasible and of the appropriate ranking.
Connective.java- The class defines connectives used in the generation of the Knowledge base.
Distribution.java- The Class defines the methods different distribution for the defeasible statements and consists of methods that calculate minimum required values for distribution for specific number of ranks.
KBGeneratorThreadedOPT.java- Class responsible for generating the knowledge base.
Knowledge.java- Class defines all data that is store in the database, it provides get and set methods for all data.
KnowledgeDAO.java- Class responsible for CRUD operations on the database.
MongoDBConnection.java- Class responsible for communicating with the database.
MonteCarloSimulation.java- Class responsible for creating multiple input parameters used to perform the Monte Carlo simulation.

Appendix D: Monte Carlo Simulation Results

num Ranks	num DefImplications	distribution	min Statements	Reuse Consequent	Transitive	Antecedent Complexity	Consequent Complexity	Connectives	Generation time
5	150	eg	20	false	true	1	2	1	31.020
50	1200	f	20	false	false	1	1	2	53.215
5	35000	f	0	true	true	1	3	1,2	68.346
5	35000	eg	0	false	false	3	1	5	113.238
5	38000	n	50	false	true	5	7	1,2	129.489
5	40000	ed	80	false	false	1	5	1,2	109.483
5	45000	r	500	false	false	2	5	5	178.176
5	50000	ld	0	true	false	1	8	1,2	82.072
10	150	f	10	false	true	3	5	1,2	42.944
10	880	r	60	false	true	9	1	1,2	56.510
10	1800	ed	150	false	false	3	1	1	54.763
10	2500	eg	200	true	true	7	4	1	52.616
50	1800	lg	0	false	false	2	2	5	39.754
10	5500	lg	350	true	false	4	7	1,2	55.276
10	1000	f	0	true	true	4	1	1,2	49.790
10	800	n	0	false	false	1	3	1,2	43.392
10	5000	r	20	true	false	1	1	1,2	99.772
10	10000	ed	0	false	false	2	2	2	57.866
10	15000	eg	0	false	true	3	3	2	68.283
10	22000	lg	0	true	true	8	3	1,2	51.952
10	25000	f	0	true	false	3	8	1	48.561
10	25000	n	50	false	true	1	1	2	448.337
10	35000	eg	80	true	false	2	1	1,2	50.432
50	5000	f	0	true	true	3	3	1,2	33.373
10	35000	f	500	false	true	1	1	1	751.190
10	40000	lg	0	true	false	1	2	1,2	54.777
10	45000	f	10	true	false	1	2	2	55.498
15	13000	f	150	false	true	6	1	1	47.984
15	600	r	0	true	true	6	1	1	32.888
15	1500	r	35	true	false	1	3	2	34.946
15	500	ld	0	false	true	3	5	1,2	33.819
15	500	lg	0	true	false	3	5	1	32.304
15	2360	n	20	false	true	3	1	2	41.080
15	800	ld	0	true	false	1	5	5	32.478
5	10000	r	60	true	true	3	4	1	36.001
15	5000	ld	0	false	false	3	4	1	42.933
15	7000	n	0	false	true	2	3	1	52.216
15	9000	lg	0	true	false	2	3	1,2	37.361
15	10000	eg	50	false	true	4	4	1,2	44.816
15	15000	ed	80	true	false	1	1	1,2	206.449
20	15000	lg	500	true	false	1	2	1,2	37.159
20	20000	n	0	false	false	8	7	2	61.065
20	22000	eg	10	true	true	8	9	2	43.301
20	25000	f	60	false	true	6	9	1,2	64.860
20	25000	lg	150	false	false	1	3	2	72.942
50	47000	ed	0	false	false	8	3	1,2	95.780
20	30000	f	200	false	true	3	1	1	91.020
20	30000	eg	350	true	false	5	7	2	49.628
20	35000	r	0	true	true	1	5	1	49.394
20	35000	eg	0	true	false	2	5	2	50.841
20	38000	f	70	false	false	1	8	5	74.065

num Ranks	num DefImplications	distribution	min Statements	Reuse Consequent	Transitive	Antecedent Complexity	Consequent Complexity	Connectives	Generation time
20	40000	r	0	true	true	3	5	1,2	68.110
20	45000	ld	80	false	false	9	1	1,2	105.134
20	45000	n	0	false	true	3	1	1,2	96.773
20	50000	r	0	true	true	7	4	1,2	60.244
25	1000	f	20	true	false	4	7	1,2	34.470
50	9000	f	0	false	true	3	8	1,2	41.489
25	350	ld	0	false	true	4	1	5	37.171
25	50	r	0	true	false	1	3	1,2	32.160
25	500	r	0	false	true	1	1	1,2	39.998
25	500	ld	0	true	false	2	2	1,2	38.592
25	1750	lg	50	true	false	3	3	2	36.981
25	2300	r	80	false	true	8	3	1	108.078
25	12800	f	500	false	false	3	8	2	55.265
25	7000	r	0	false	true	1	1	1	117.169
25	9000	ld	10	false	true	2	1	2	49.759
25	10000	lg	60	true	false	1	1	5	152.498
50	10000	r	50	false	true	1	1	1,2	356.143
25	15000	r	150	false	true	5	7	1,2	207.914
25	15000	eg	200	false	false	1	5	1,2	47.651
25	20000	ed	350	false	true	2	5	1,2	56.479
25	22000	r	0	true	false	1	8	1,2	42.110
25	25000	r	0	true	false	3	5	1,2	43.033
25	25000	ld	20	true	true	9	1	5	43.234
25	25000	lg	0	false	true	3	1	1,2	67.919
30	1200	lg	0	true	true	7	4	1,2	49.885
30	20000	f	0	true	false	4	7	5	54.013
30	35000	n	0	false	false	1	1	1,2	981.617
50	15000	ld	80	true	false	2	1	1,2	44.765
30	38000	eg	50	true	true	2	3	1,2	61.715
30	50000	f	80	false	false	3	1	1,2	115.721
40	20820	lg	500	true	true	1	1	1,2	339.630
40	500	r	0	false	true	1	3	1	46.716
40	7900	ed	10	true	false	3	1	1	38.440
40	8933	eg	60	false	true	5	7	1,2	49.144
40	6820	lg	150	false	false	1	5	1,2	44.873
40	8200	f	200	true	false	2	5	1,2	39.225
40	15320	n	350	true	true	1	8	1,2	128.996
120	300	f	0	true	true	3	1	1,2	39.402
50	10000	r	0	true	false	3	5	1	44.362
120	5930	r	10	true	false	7	4	1	80.636
120	25000	ld	60	false	false	4	7	1	65.153
120	50100	f	350	true	true	1	2	5	51.634
130	9000	ld	0	false	false	8	3	1,2	42.214
130	635	r	0	true	true	3	8	1,2	31.840
130	35000	r	20	true	false	1	1	1	691.882
130	35000	ld	0	true	false	2	1	1,2	45.262
130	13800	f	0	true	false	1	3	5	41.115
130	40000	ld	0	false	true	3	1	2	101.711
130	40000	f	50	false	false	5	7	1	116.443
50	15000	n	10	false	true	9	1	2	100.526
130	35000	n	80	true	true	1	5	2	303.102
140	18000	ld	0	true	false	1	8	2	54.814
140	1777	f	10	false	true	3	5	5	31.596
140	19000	lg	60	false	false	9	1	1,2	74.426
140	29000	r	150	true	true	3	1	1,2	4133.818
2000	47700	f	23	false	false	3	1	1,2	133.218
150	45069	lg	20	false	true	3	8	1,2	88.463
500	50000	f	80	true	true	9	1	1	66.155

num Ranks	num DefImplications	distribution	min Statements	Reuse Consequent	Transitive	Antecedent Complexity	Consequent Complexity	Connectives	Generation time
220	500	f	0	false	false	3	5	1	41.141
250	700	f	0	false	false	7	4	1,2	42.495
5	9000	f	10	false	false	1	5	2	48.274
50	15000	f	60	true	false	3	1	5	47.593
200	4400	r	20	true	false	3	5	5	358.305
50	20000	r	150	true	false	7	4	1,2	201.227
50	22000	r	200	false	true	3	1	1,2	864.516
5	15000	ed	150	false	false	2	3	2	64.958
50	25000	ld	350	true	false	7	4	1	51.356
50	25000	lg	0	false	true	4	7	1,2	82.120
50	3000	lg	20	true	true	2	3	5	32.646
60	3000	n	0	true	true	1	2	1,2	36.237
60	35000	lg	0	false	false	8	3	1	80.747
60	350	r	0	false	true	3	8	2	45.695
40	10000	eg	0	false	false	3	5	2	42.539
60	38000	r	50	true	false	2	1	5	102.384
5	15000	eg	200	false	true	2	3	1	54.538
60	40000	f	500	false	false	1	3	2	85.488
60	45000	lg	0	false	true	3	1	1	94.564
60	45000	ld	10	true	false	5	7	2	51.712
70	7600	lg	60	true	false	1	5	1	40.068
70	10900	f	150	true	true	2	5	2	37.891
70	500	f	0	true	false	9	1	1,2	36.359
70	7000	lg	0	false	true	3	1	1,2	51.126
70	9000	ld	70	true	false	7	4	1,2	44.467
40	150	f	0	true	true	9	1	2	31.545
5	20000	lg	350	true	false	4	4	2	56.856
70	28000	f	0	false	true	4	7	1,2	77.350
70	15000	r	80	true	true	4	1	5	226.442
70	9000	ld	0	false	false	1	3	1,2	45.531
70	10000	f	0	true	true	1	1	1,2	121.922
70	15000	n	20	false	true	2	2	5	254.045
70	15000	r	0	false	false	3	3	1,2	62.937
70	20000	ld	0	true	true	8	3	1,2	40.705
70	22000	n	0	true	false	3	8	1,2	46.064
70	25000	lg	0	false	true	1	1	1	409.845
40	9000	lg	70	false	true	3	1	1,2	41.136
70	25000	f	50	true	false	2	1	2	42.558
5	22000	f	0	true	false	1	1	2	304.339
70	45000	lg	10	false	true	2	5	1	188.306
70	40000	f	60	false	false	1	8	1,2	152.482
70	45000	r	150	false	true	3	5	1,2	2451.159
90	20000	r	200	false	false	9	1	5	5634.771
90	9000	lg	0	false	true	7	4	1,2	41.986
90	10000	lg	0	false	false	4	7	1,2	44.183
5	25000	n	0	false	false	1	9	1,2	57.751
90	4700	f	20	false	true	1	1	2	79.567
40	10000	f	0	true	false	7	4	2	34.755
90	9000	n	0	true	true	2	3	2	44.831
90	25000	lg	0	true	false	3	1	1,2	42.387
90	15000	f	0	true	true	1	1	1	176.891
90	15000	ld	0	false	false	1	3	1	45.945
90	20000	f	50	true	true	3	1	1,2	39.531
90	22000	n	80	true	false	5	7	1,2	430.051
90	50000	ld	500	false	false	1	5	1,2	92.450
90	25000	f	0	true	true	2	5	1,2	41.921
5	30000	eg	20	true	true	8	7	1,2	50.494
5	30000	eg	20	true	true	8	7	1,2	77.164
40	35000	eg	80	true	true	4	7	1	44.514

num Ranks	num DefImplications	distribution	min Statements	Reuse Consequent	Transitive	Antecedent Complexity	Consequent Complexity	Connectives	Generation time
90	30000	f	150	true	false	9	1	1,2	57.815
90	35000	ld	200	true	true	3	1	2	62.052
90	37000	lg	350	false	false	7	4	1	101.634
90	50000	f	0	true	true	4	7	2	68.545
90	38000	n	0	true	true	4	1	1	74.800
90	40000	f	70	false	false	1	3	2	93.823
90	45000	lg	80	false	true	2	2	1,2	111.874
110	7000	ld	0	false	false	3	3	1,2	53.956
5	30000	f	0	false	true	8	9	1,2	113.607
110	5000	f	0	true	true	8	3	1,2	46.807
50	500	n	0	false	false	4	1	2	58.260
110	7000	n	20	true	false	3	8	1,2	193.687
110	15000	lg	0	true	true	1	1	1,2	226.359
110	10000	r	0	true	false	2	1	5	49.797
110	35000	n	0	true	false	1	1	1,2	735.918
110	9000	ld	0	false	true	3	5	1,2	57.335
110	50000	lg	50	true	false	9	1	1,2	61.800
110	45000	f	80	false	true	3	1	2	100.010
120	300	f	0	true	true	3	1	1,2	44.094
5	35000	lg	0	true	false	6	9	1,2	77.422
120	5930	r	10	true	false	7	4	1	90.739
50	50000	eg	0	false	true	1	3	1	106.588
120	25000	ld	60	false	false	4	7	1	85.548
5	50	lg	0	true	true	1	1	1,2	76.909
5	200	f	0	false	true	1	2	1	60.387
5	500	f	0	false	false	6	1	1,2	63.006
5	500	eg	0	false	false	6	1	1,2	63.711
5	1000	n	50	false	true	1	3	1,2	72.830
5	800	ed	80	true	false	3	5	2	41.447
5	5000	r	500	true	true	3	5	2	39.729
5	7000	ld	0	true	true	3	1	1,2	44.443
5	35000	lg	0	true	false	6	9	1,2	74.821