CS/IT Honours Project
Final Paper 2022

Title: Extending Defeasible Reasoning Beyond Rational Closure

Author: Alec Lang

Project Abbreviation: EXTRC

Supervisor(s): Professor 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**	

Extending Defeasible Reasoning Beyond Rational Closure

Alec Lang
University of Cape Town
Cape Town, South Africa
lngale007@myuct.ac.za

ABSTRACT

Defeasible reasoning, an important branch of knowledge representation and reasoning (KRR), offers a powerful framework for modeling and handling uncertain, incomplete, and conflicting information. Rational closure has assumed a significant role in the domain of defeasible reasoning research and forms the foundation for multiple rational defeasible entailment relationships. This project aims to provide a parameterised, non-deterministic tool based off of Rational Closure's BaseRank algorithm, to facilitate the generation of complex defeasible knowledge bases. Furthermore, an optimised variant of the generator is presented. We evaluate the performance of both generators and analyze how different complexities of defeasible implications impact the generation time. Additionally, the project aims to provide a tool that advances the process of testing and validating new defeasible entailment relations.

CCS CONCEPTS

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

KEYWORDS

artificial intelligence, knowledge representation and reasoning, defeasible reasoning, rational closure, defeasible knowledge base generation

1 INTRODUCTION

Knowledge Representation and Reasoning (KRR) is a sub-field of artificial intelligence where knowledge can be represented in a structured way by using formal logic. The represented knowledge can then be reasoned on, whereby it is manipulated using a set of rules to infer new information[2].

Knowledge bases are collections of information and facts represented in a structured manner. In particular, the project addresses the need for complex defeasible knowledge bases. Defeasible reasoning involves dealing with information that is not always certain and might be subject to exceptions or conditions. Defeasible knowledge bases provide a framework for modeling and reasoning about uncertain or incomplete information. An extension to propositional logic, put forward by KLM [8], is used to represent the information in these defeasible knowledge bases. This project aims to contribute to the analysis and evaluation of new entailment relations within the realm of defeasible reasoning by generating sophisticated knowledge bases.

This paper introduces a comprehensive and parameterized non-deterministic defeasible knowledge base generator, along with an optimized variant. Furthermore, the paper presents a project centered around assessing the influence of various defeasible implication configurations on the computation time of their generation, as well as comparing the performance of the two generators. The generators are designed to create knowledge bases with a wide range of configurations, allowing for a comprehensive exploration of different knowledge base structures. They are based on the Rational Closure's ranking algorithm known as BaseRank. This means that the generated knowledge bases will be structured in a manner consistent with how Base Rank would rank a knowledge base.

2 BACKGROUND

2.1 Propositional Logic

2.1.1 Grammar. Propositional logic is a logical system that is used to reason about, and model knowledge of the world. The language of propositional logic $\mathcal{L}$ is built up from atoms and boolean connectives. An atom is a statement, or a representation of a statement, and are often denoted using small Latin alphabet letters $\mathcal{P} = \{b, o, r, \ldots\}$. Each atom is attributed a truth value of either true (T) or false (F). These atoms can be combined together using a set of boolean connectives, $\{\neg, \land, \lor, \rightarrow, \leftrightarrow\}$, to create propositional formulas[1].

Name	Type	Symbol
Negation	Unary	$\neg$
Conjunction	Binary	$\land$
Disjunction	Binary	$\lor$
Implication	Binary	$\rightarrow$
Equivalence	Binary	$\leftrightarrow$

Figure 1: Propositional logic boolean operators

2.1.2 Semantics. The semantics of propositional logic involves the concept of *validity*, which refers to the property of a statement where the truth of the premises guarantees the truth of the conclusion. In propositional logic, a statement is valid if and only if its conclusion follows logically from its premises. The assignment of truth values to atoms is referred to as interpretations or worlds. Mathematically, this can be expressed as $u : \mathcal{P} \rightarrow \{T, F\}$, where $\mathcal{P}$ is the set of propositional atoms and $\{T, F\}$ represents the set of truth values.

If a valuation assigns a truth value of true to a propositional atom, then the atom is said to be satisfied in that valuation. *Satisfaction* is denoted with the symbol $\Vdash$, such that for some valuation u, if it is the case that for some $p \in \mathcal{P}$, $u(p)$ = T, then $u \Vdash p$, and if $u(p)$ = F, then $u \nVdash p$[1].

2.1.3 Entailment. A logical consequence, known as an *entailment*, is an outcome that arises from a statement or set of statements. Entailment is denoted by the symbol $\models$. If we have two formulas α and β, both belonging to $\mathcal{L}$, we say that β is a logical consequence of α, denoted by $\alpha \models \beta$, when for every $u \in \mathscr{U}$ such that u satisfies α (i.e., $u \Vdash \alpha$), u also satisfies β (i.e., $u \Vdash \beta$). In other words, $\alpha \models \beta$ holds if and only if $\hat{\alpha} \subseteq \hat{\beta}$ ($\hat{\alpha}$ is a subset of $\hat{\beta}$).

A defined set of propositional statements is known as a propositional knowledge base. A propositional knowledge base, $\mathcal{K}$ entails any α if every model of $\mathcal{K}$ is also a model of α[1]. The issue with propositional logic is that it is *monotonic*, meaning that the addition of new information might help draw new conclusions, but it will never lead to the retraction of a previously drawn conclusion[13].

2.2 Defeasible Reasoning

2.2.1 KLM approach. Defeasible reasoning is *non-monotonic*, meaning conclusions are drawn based on incomplete or uncertain information, where the conclusions can be subjected to revision in the occurrence of new conflicting information[7, 11]. This allows for systems to reason in a way which is closer to humans. Kraus, Lehmann and Magidor created the KLM-framework for defeasible reasoning[8, 10], by extending propositional logic to include the preferential consequence relation symbol $\mid\sim$. This introduced the defeasible implication, being a statement $\alpha \mid\sim \beta$, where $\alpha, \beta \in \mathcal{L}$ [6]. This equates to mean "α typically implies β", so if α is true, then β is probable to also be true. A defeasible knowledge base is said to be a set of defeasible implications.

2.2.2 Entailment. Unlike in propositional logic, in defeasible reasoning there is no set method to determine defeasible entailment (denoted $\approx$). KLM[8, 10] proposed the following properties that had to be satisfied by a defeasible entailment relation, with any such form of defeasible entailment that satisfies these properties being known as $LM - Rational$:

$$\text{(Ref)}\ \mathcal{K} \models\!\!\approx \alpha \mid\sim \alpha \qquad \text{(LLE)}\ \frac{\alpha \equiv \beta,\ \mathcal{K} \models\!\!\approx \alpha \mid\sim \gamma}{\mathcal{K} \models\!\!\approx \beta \mid\sim \gamma}$$

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

$$\text{(Or)}\ \frac{\mathcal{K} \models\!\!\approx \alpha \mid\sim \gamma,\ \mathcal{K} \models\!\!\approx \beta \mid\sim \gamma}{\mathcal{K} \models\!\!\approx \alpha \vee \beta \mid\sim \gamma} \qquad \text{(CM)}\ \frac{\mathcal{K} \models\!\!\approx \alpha \mid\sim \beta,\ \mathcal{K} \models\!\!\approx \alpha \mid\sim \gamma}{\mathcal{K} \models\!\!\approx \alpha \wedge \beta \mid\sim \gamma}$$

2.3 Rational Closure

Rational closure was the first non-monotonic entailment relation defined by Lehmann and Magidor[10] and is a form of defeasible entailment that satisfies LM-Rationality. It is a pattern of *prototypical reasoning*[9], and is the most conservative of all the defeasible entailment relations in the KLM framework. This means that members of a world only inherit the properties of that world if they are the most typical. Two approaches to computing rational closure will be defined. The first one, *minimal ranked entailment*, defines rational closure semantically using a unique ranked model of $\mathcal{K}$[6]. The second defines an algorithm to compute the rational closure of a defeasible knowledge base $\mathcal{K}$, first put forward by Lehmann and Magidor[10]

2.3.1 Minimal Ranked Entailment. Minimal ranked entailment involves defining a partial ordering of all ranked models of some knowledge base $\mathcal{K}$, with this denoted by $\preceq_{\mathcal{K}}$. A definition to impose an ordering $\preceq_{\mathcal{K}}$ on the typicality of the ranked interpretations in a knowledge base $\mathcal{K}$ was given by Casini et al.[3] and is as follows: $\mathcal{R}_1 \preceq_{\mathcal{K}} \mathscr{R}_2$ if for every $v \in \mathscr{U}$, $\mathcal{R}_1(v) \leq \mathcal{R}_2(v)$. Further Giordano et al. [5] showed that the partially ordered set $\langle \mathcal{R}, \preceq_{\mathcal{K}} \rangle$ has a minimal element, $\mathcal{R}^{\mathcal{K}}_{RC}$ for $\mathcal{K}$, with minimal interpretations being "pushed down"[6] as much as is possible. Minimal ranked entailment $\models\!\!\approx$ can be defined given a defeasible knowledge base $\mathcal{K}$ and the minimal ranked interpretation satisfying $\mathcal{K}$, $\mathcal{R}^{\mathcal{K}}_{RC}$: for any defeasible implication $\alpha \mid\sim \beta$, $\mathcal{K} \models\!\!\approx \alpha \mid\sim \beta$ iff $\mathcal{R}^{\mathcal{K}}_{RC} \Vdash \alpha \mid\sim \beta$.

2.3.2 Algorithmic approach. Before the algorithm can be looked at the *materialisation* of the knowledge base must first be defined: The material counterpart of a defeasible implication $\alpha \mid\sim \beta$ is the propositional formula $\alpha \to \beta$. Given a defeasible knowledge base $\mathcal{K}$, the material counterpart, denoted $\overrightarrow{\mathcal{K}}$, is the set of material counterparts, $\alpha \to \beta$, for every defeasible implication $\alpha \mid\sim \beta \in \mathcal{K}$[6]. A propositional statement α is then said to be exceptional with regards to a knowledge base $\mathcal{K}$ if and only if $\mathcal{K} \models \neg\alpha$ (i.e., α is false in all the most typical interpretations in every ranked model of $\mathcal{K}$)[3, 6].

2.3.3 BaseRank Algorithm. The first step in calculating rational closure is the *BaseRank* algorithm. BaseRank is an algorithm that separates formulas in a knowledge base into ranks, based on how general they are, with the most general statements being in the lowest rank. Each propositional formula in $\mathcal{K}$ is mapped to the set of natural numbers and infinity: $\{0, 1, 2, 3, ...\} \cup \infty$ (Most typical in rank 0 and so forth). The input to the algorithm is a defeasible knowledge base with the output being a tuple of sets of classical implications that are the material counterparts to the defeasible implications in $\mathcal{K}$, corresponding to the sequence $\mathcal{E}^{\mathcal{K}}_n$ of exceptional subsets of $\mathcal{K}$[6].

Algorithm 1: BaseRank

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

$i := 0$;
$E_0 := \overrightarrow{\mathcal{K}}$;
repeat
  $E_{i+1} := \{\alpha \to \beta \in E_i \mid E_i \models \neg\alpha\}$;
  $R_i := E_i \setminus E_{i+1}$;
  $i := i + 1$;
until $E_{i-1} = E_i$;
$R_\infty := E_{i-1}$;
if $E_{i-1} = \emptyset$ **then**
  $n := i - 1$;
else
  $n := i$;
return $(R_0, \ldots, R_{n-1}, R_\infty, n)$

2.3.4 RationalClosure Algorithm. The second step in calculating the rational closure is the *RationalClosure* algorithm. RationalClosure determines if a statement is defeasibly entailed by the knowledge base. It takes as input a knowledge base $\mathcal{K}$ and a defeasible implication $\alpha \mid\sim \beta$, and returns true if and only if the implication $\alpha \mid\sim \beta$ is in the rational closure of $\mathcal{K}$ ($\mathcal{K} \models_{RC} \alpha \mid\sim \beta$)[4].

Algorithm 2: `RationalClosure`

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

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

3 PROJECT AIMS

The main aims of the project was to:

- Create a non-deterministic defeasible knowledge base generator, that is capable of efficiently producing knowledge bases with different configurations for the purpose of testing defeasible entailment relations.
- Develop an optimised variant of the knowledge base generator.
- Compare the performance of the standard and optimised generators and evaluate the impact of defeasible implication complexity on the computation time of generation.

4 DEFEASIBLE IMPLICATION GENERATION

Defeasible implications, DIs, are the information that make up a defeasible knowledge base, with it consisting of two key parts: the antecedent and the consequent. These two components are connected by a $\mid\sim$ to form a defeasible implication, with a collection of these defeasible implications forming a defeasible knowledge base. The DefImplicationBuilder class handles the generation of defeasible implications for the knowledge base which we can split into 3 classes: structure DIs, simple DIs and complex DIs.

4.1 Atom

An atom represents the fundamental building block of the defeasible implications in the knowledge base, with atom generation being handled by the AtomBuilder class.

To add a degree of pseudo-randomness to the atom generation, the AtomBuilder class generates atoms by repeatedly selecting a random character from a chosen character-set, based on the length of the atom, to ensure that each new KB generated is unique in its information. A list of generated atoms is maintained to ensure that there are no duplicate atoms generated, as these would break the desired structure of the knowledge base. The countChecker function periodically updates the atoms length (the number of characters) according to how many atoms have already been generated for a knowledge base. This is done to ensure that unique atoms are continually generated from a finite character set.

The setCharacters function within the AtomBuilder class enables the selection of different character sets for atom generation. These being upperlatin [capital Latin alphabet], lowerlatin [lowercase Latin alphabet], altlatin [an assortment of alternate Latin characters] and greek [lowercase Greek alphabet].

Furthermore, the class was built to adhere to the Singleton pattern, ensuring that only a single instance of the AtomBuilder is created throughout the life-cycle of KB generation. This ensures that the changing list of already generated atoms and the length of said atoms remains constant when multiple threads are generating new atoms at once.

4.2 Structure Defeasible Implications

At Rank 0, two base atoms are initially generated to form the baseline defeasible implication for Rank 0 of the KB, with the antecedent and consequent being named rankBaseAnt and rankBaseCons. This becomes the lynch pin around which a rank is built. At each rank thereafter, a minimum of two defeasible implications are needed to build the structure of said rank. These baseline defeasible implications are once again generated before all else. Further defeasible implications can be connected onto the baseline defeasible implications to build out a rank with more defeasible implications. This is done using two different functions:

4.2.1 ***rankBuilderConstricted.*** This function constricts the generator to use the minimum amount of DIs to constitute a rank. It is called if a rank only has 2 DIs or if the user chooses to reuse the rankBaseCons from the rank before. A new rankBaseAnt is generated and acts as the antecedent in the formation of two DIs. In the first DI, the rankBaseAnt is linked up to the rankBaseAnt from the previous rank and in the second it is linked to the negated rankBaseCons from the previous rank.

Rank 0	$A \mid\sim B, \ldots$	rankBaseAnt = A, rankBaseCons = B
Rank 1	$C \mid\sim A, C \mid\sim \neg B, \ldots$	rankBaseAnt = C, rankBaseCons = $\neg B$
Rank 2	$D \mid\sim C, D \mid\sim B, \ldots$	rankBaseAnt = D, rankBaseCons = B
Rank 3	...	...

Figure 2: rankBuilderConstricted DIs

4.2.2 ***rankBuilder.*** This function generates the DIs to form the structure of a rank. It is called if the user chooses to generate a new rankBaseCons for each rank. A new rankBaseAnt is generated and acts as the antecedent in the formation of three DIs. In the first DI, the rankBaseAnt is linked up to the rankBaseAnt from the previous rank, in the second it is linked to the negated rankBaseCons from the previous rank and in the third DI it is linked to the new rankBaseCons.

Rank 0	$A \mid\sim B, \ldots$	rankBaseAnt = A, rankBaseCons = B
Rank 1	$C \mid\sim A, C \mid\sim \neg B, C \mid\sim D, \ldots$	rankBaseAnt = C, rankBaseCons = D
Rank 2	$E \mid\sim C, E \mid\sim \neg D, E \mid\sim F, \ldots$	rankBaseAnt = E, rankBaseCons = F
Rank 3	...	...

Figure 3: rankBuilder DIs

4.3 Simple Defeasible Implications

Simple defeasible implications are ones in which there are only one atom per antecedent and consequent. In the context of the program it can be said that they have a complexity of 0. There are 3 distinct functions for generating simple defeasible implications:

4.3.1 ***recycleAtom***. This function will generate a DI with a newly generated atom as the antecedent and reuse a random curRankAtom as the consequent. A curRankAtom is an antecedent with a base rank that is equal to the current rank.

4.3.2 ***negateAntecedent***. This function will generate a DI with a newly generated, negated atom as the antecedent and reuse a random curRankAtom as the consequent. The negated atom is then saved as a anyRankAtom and may then be used in a later rank in a DI generated by the reuseConsequent function. An anyRankAtom is an atom that may be used as a consequent in any rank greater than its own base rank.

4.3.3 ***reuseConsequent***. This function will generate a DI by reusing a random curRankAtom as the antecedent and a random anyrankAtom as a consequent.

4.4 Complex defeasible implications

A significant contribution of the program is its ability to generate defeasible implications characterized by differing levels of complexity. The term complexity is used to denote the degree of sophistication within defeasible implications, measured by the number of connectives present in both the antecedent and the consequent. In particular, the antecedent and consequent can each possess a maximum complexity of 2. The connectives are logical operators that link atoms together, enabling the formation of compound defeasible implications. The available connective types are the binary operators; conjunction, disjunction, implication, and bi-implication.

Five types of complex defeasible implications can be generated using:

(1) disjunctionDefImplication
(2) conjuntionDefImplication
(3) implicationDefImplication
(4) biImplicationDefImplication
(5) mixedDefImplication

Each function handles the generation of defeasible implications of varying complexities using their aforementioned connectives, with the exception of mixedDefImplication, which pseudo-randomly chooses from available connectives when generating a defeasible implication. These functions all use a key to determine the structure of the DI to be generated.

4.5 Rules

The Rules class guides the generation of complex defeasible implications within the knowledge base. It operates using a set of valid keys stored in a hashmap, which represents the combination of connection types, antecedent complexities, and consequent complexities that are permissible for generating DIs.

4.5.1 ***Key Map Creation and Storage***. The class stores a hashmap named keyMap of all possible keys. A key is a combination of connection type, antecedent complexity, and consequent complexity, with each key corresponding to a unique DI structure. For example the key "2,1,2" would signify conjunction connective types, an antecedent complexity of 1 connective and a consequent complexity of 2 connectives, eg: ($A \wedge D \mid\sim B \wedge E \wedge F$). A key's corresponding value indicates the minimum number of curRankAtoms [refer to previous explanation] required to successfully generate a DI with that key. The keyMap effectively stores the criteria that must be met for generating different types of DIs.

4.5.2 ***Key Generator Function***. The keyGenerator function generates a key by randomly selecting from the available connection types, antecedent complexities, and consequent complexities. The generator ensures that the key is valid by calling the checker function.

4.5.3 ***Checker Function***. The keyGenerator and checker functions work in tandem to provide a valid key for generating DIs in a given rank. The keyGenerator function first generates a random key, then the checker function determines if the key is valid by comparing the number of available curRankAtoms in the rank with the associated value from the keyMap. If the number of available curRankAtoms is greater than or equal to the required minimum, the function returns the original key to the generator as valid. Otherwise, a simple DI will be generated in place of a complex one.

5 KNOWLEDGE BASE CONSTRUCTION

5.1 Distribution

Distribution shapes the arrangement of defeasible implications within a knowledge base. The Distribution class was created to handle various functions for controlling the organization of DIs across different ranks. It takes a user defined number of DIs, number of ranks and distribution type, and outputs an array of the distribution over the ranks for the knowledge base. Four distributions were implemented as follows:

5.1.1 ***Flat Distribution***. This distribution divides the total number of DIs evenly across all ranks. Any remaining DIs are distributed starting from the last rank and moving upwards.

5.1.2 ***Linear Growth Distribution***. In this distribution, the number of DIs in each rank increases linearly, with higher ranks receiving more DIs.

5.1.3 ***Linear Decline Distribution***. This distribution follows a linear decline pattern, wherein the number of DIs in each rank decreases linearly.

5.1.4 ***Random Distribution***. This distribution assigns DIs randomly to ranks.

5.2 Standard Generator

KBGenerator class is responsible for constructing the knowledge bases by controlling the generation of defeasible implications using the KBGenerate function. KBGenerate builds the KB, rank by rank, beginning with Rank 0. It is given arrays for the DI distribution, complexity of the antecedent, complexity of the consequent and connective types, and two booleans for simple only DI generation [simpleOnly] and rankBaseCons reuse [reuseConsequent]. The time complexity of KBGenerate is O(r * d), with r being the ranks and d being the defeasible implications. The implementation of KBGenerate involves the following steps:

5.2.1 ***Initialisation**.* Initialise the knowledge base (KB) as a LinkedHashSet of LinkedHashSets, as well as generating the base atoms for both the consequent (rankBaseCons) and antecedent (rankBaseAnt) of Rank 0. A list is kept to hold atoms that are reusable in any rank (anyRankAtoms).

5.2.2 ***Rank Generation Loop**.* Iterate through ranks from Rank 0 until the desired number of ranks is reached. For each rank:

(1) Keep track of lists to store generated defeasible implications, reusable atoms for the current rank's antecedent (curRankAtoms), and reusable atoms usable in any rank temporarily (anyRankAtomsTemp). rankBaseAnt is added to curRankAtoms list for reuse in other DI in the current rank.

(2) If the current rank is 0, build the baseline DI using the rankBaseAnt and rankBaseCons as the consequent and antecedent respectively.

(3) If the rank is greater than 0, generate the initial structure DIs for the rank based on reuseConsequent flag and the number of DIs needed for the rank. If reuseConsequent is false and there are at least 3 DIs for this rank, the rankBuilder function is used, with a new rankBaseCons being generated. Else rankBuilderConstricted is run.

(4) Iterate through the remaining number of DIs for this rank:
- If simpleOnly is true, randomly choose between the three simple DI functions: recycleAtom, negateAntecedent or reuseConsequent. If there are no atoms that can be reused in any rank, anyRankAtoms is empty, then recycleAtom will be called over reuseConsequent. Both negateAntecedent and reuseConsequent will add an atom to anyRankAtomsTemp, which will be added to anyRankAtoms once the rank is completely generated for reuse in higher ranks of the knowledge base.
- If simpleOnly is false, a key is generated using the keyGenerator function. The connection type is determined from the key and the corresponding complex DI function is called from DefImplicationBuilder.

(5) Negate the rankBaseCons to prepare for the next rank.

(6) Add the set of generated DIs to the KB, update anyRankAtoms with anyRankAtomsTemp and increment the rank counter.

5.2.3 ***Return KB**.* After generating DIs for all ranks, return the knowledge base.

Algorithm 3: `KBGenerator.KBGenerate`

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

```
rank = 0
K = newLinkedHashSet < LinkedHashSet <
  DefImplication >> ()
rankBaseAnt, rankBaseCons = generateAtom()
anyRankAtoms = newArrayList()
while rank! = dIDistribution.length do
    DIs, curRankAtoms, anyRankAtomsTemp =
    new ArrayList()
    dINum =
    dIDistribution[rank]
    if rank == 0 then
        rankZero(DIs, rankBaseCons, rankBaseAnt)
        dINum − −
    else
        if reuseConsequent == false and dINum >= 3
         then
            rankBuilder(gen, DIs, rankBaseCons, rankBaseAnt)
            dINum − −
        else
            rankBuilderConstricted(gen, DIs, rankBaseCons,
            rankBaseAnt)
            dINum = dINum − 2
    curRankAtoms.add(rankBaseAnt)
    while dINum! = 0 do
        if simpleOnly == true then
            decision = random.nextInt(3)
            simpleDI(decision, generator, DIs, anyRankAtoms,
            curRankAtoms, anyRankAtomsTemp)
        else
            key = keyGenerator(connectiveType,
            complexityAnt, complexityCon,
              curRankAtoms.size())
            complexDI(key, gen, DIs, curRankAtoms)
        dINum − −
    rankBaseCons.negateAtom()
    K.add(newLinkedHashSet < DefImplication >
      (DIs))
    anyRankAtoms.addAll(anyRankAtomsTemp)
    rank + +
return K
```

5.3 Optimised Generator

The KBGeneratorThreaded is an optimized version of the knowledge base generator that leverages multi-threading to enhance the efficiency of DI generation. The main idea behind KBGeneratorThreaded is to assign each rank to a separate thread for DI generation. Once all ranks have been generated, the final DI for each rank is computed with the current rank's rankBaseAnt and the previous rank's rankBaseAnt as the antecedent and consequent respectively. This final DI ties the ordering together to form the complete knowledge base. Knowledge bases built using the optimised generator always reuse the rankBaseCons between ranks. The time complexity of KBGeneratorThreaded.KBGenerate is O(d + r), with d being the number of defeasible implications run in parallel and r being the number ranks. The optimised implementation of KBGenerate involves the following steps:

5.3.1 ***Initialisation***. The main executor service is created with a fixed thread pool size equal to the number of available threads. A LinkedHashSet of LinkedHashSets to store the generated knowledge base is created. The base atom for the consequent (rankBaseCons) is generated, and an array to store the antecedents of each rank, rankBaseAnts, is initialised.

5.3.2 ***Threaded Rank Generation***. A loop iterates through each rank in the DI distribution. For each rank, a new thread is submitted to the executor service. The thread is assigned the task of generating DIs for that rank by running the function generateRank. It is given arrays for the DI distribution, complexity of the antecedent, complexity of the consequent, connective types and rankBaseAnts, a boolean for simple only DI generation [simpleOnly], as well as the current rank number and the rankBaseCons.

5.3.3 ***DI Generation Logic***.

(1) The antecedent for the rank, rankBaseAnt, is generated and a synchronized block ensures that it is correctly stored in the rankBaseAnts array.

(2) If the remainder of the current rank with 2 is equal to 0, then build the baseline DI using the rankBaseAnt and rankBaseCons as the consequent and antecedent. Else, negate the rankBaseCons and build the baseline DI using the rankBaseAnt and the negated rankBaseCons.

(3) The generation process for each DI within a rank follows the same structure to the original generator. If simpleOnly is enabled, simple DIs are generated using random choices. If simpleOnly is disabled, complex DIs are generated using the key-based approach to determine the DI structure.

(4) The generated DIs are returned as a LinkedHashSet.

5.3.4 ***Rank Handling***. After all ranks are generated and the threads complete, the main thread processes each rank of generated DIs. The final DI of each rank (except the first) is created by connecting the previous rank's rankBaseAnt with the current rank's rankBaseAnt. The knowledge base is then returned

Algorithm 4: `KBGeneratorThreaded.KBGenerate`

```
Input   : dIDistribution, simpleOnly, complexityAnt,
          complexityCon, connectiveType
Output  : A defeasible knowledge base 𝒦
executor = Executors.newFixedThreadPool(numThreads)
  𝒦 = newLinkedHashSet < LinkedHashSet <
  DefImplication >> ()
anyRankAtoms = newArrayList()
rankBaseCons = generateAtom()
rankBaseAnts =
new Atom[dIDistribution.length]
try:
    futures = newArrayList()
    for rank = 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|ExecutionExceptione:
    e.printStackTrace()
finally:
    executer.shutdown()
for set : 𝒦 do
    if !firstSetProcessed then
        firstSetProcessed = true
        continue
    set.add(newDefImplication(rankBaseAnts[i],
    new Atom(rankBaseAnts[i-1])))
    i ++
return 𝒦
```

Algorithm 5: `KBGeneratorThreaded.generateRank`

```
Input: rank, dIDistribution, simpleOnly, complexityAnt,
       complexityCon, connectiveType, rankBaseCons,
       rankBaseAnts, anyRankAtoms
Output: A rank of DIs
rankBaseAnt = generateAtom()
anyRankAtoms = newArrayList()
synchronized rankBaseAnts:
    rankBaseAnts[rank] = rankBaseAnt
DIs, curRankAtoms, anyRankAtomsTemp =
new ArrayList()
dINum =
dIDistribution[rank]
dINum − −
if rankmod2 = 0 then
    rankZero(DIs, rankBaseCons, rankBaseAnt)
else
    rBCNegated = newAtom(rankBaseCons)
    rBCNegated.negateAtom()
    rankZero(DIs, rBCNegated, rankBaseAnt)
curRankAtoms.add(rankBaseAnt)
if !(rank == 0) then
    dINum − −
while dINum! = 0 do
    if simpleOnly == true then
        decision = random.nextInt(3)
        simpleDI(decision, generator, DIs, anyRankAtoms,
        curRankAtoms, anyRankAtomsTemp)
    else
        key = keyGenerator(connectiveType,
        complexityAnt, complexityCon,
        curRankAtoms.size())
        complexDI(key, gen, DIs, curRankAtoms)
    dINum − −
anyRankAtoms.addAll(anyRankAtomsTemp) return
  newLinkedHashSet(DIs)
```

6 SYSTEM DESIGN AND IMPLEMENTATION

6.1 System Implementation and Architecture

The system was developed in Java, with Maven being used to manage the software. The software's UML diagrams and class descriptions are attached in the Appendix.

6.2 Generator Features

The following is a run through of the features and usage of the software:

6.2.1 Command-line Interface. A command-line interface is implemented for the software. The user can either directly enter the specifications they desire using the interface or they can provide the input in a text file and run using a single command.

6.2.2 No. Ranks. The user is first prompted to enter a non-negative number of ranks for the knowledge base.

6.2.3 Distribution. Then the distribution for the defeasible implications is chosen. The user may enter either 'f' (flat), 'lg' (linear-growth), 'ld' (linear-decline) or 'r' (random)

6.2.4 No. Defeasible Implications. The number of defeasible implications is then entered. The user will be given a minimum and may enter any number greater than or equal to it. This is done so that the structure of the knowledge base is maintained for a chosen amount of ranks and distribution type.

6.2.5 Simple Only. Determines if the knowledge base contains only simple defeasible implications or a mixture of simple and complex.

6.2.6 Reuse Consequent. Determines if the knowledge base will reuse the rankBaseCons for all ranks, or if it generates a new one for each rank.

6.2.7 Antecedent complexity. Determines the allowed amount of connectives in the antecedent of a complex DI. The user may enter any assortment of 0, 1 and 2, separated by commas.

6.2.8 Consequent complexity. Determines the allowed amount of connectives in the consequent of a complex DI. The user may enter any assortment of 0, 1 and 2, separated by commas.

6.2.9 Connective Types. Determines the allowed connective types in a complex DI. The user may enter any assortment, separated by commas, of 1 (disjuntion), 2 (conjunction), 3 (implication), 4 (bi-implication) and 5 (mixture).

6.2.10 Adjustable Connective Symbols. The user may change the connective symbols to suit their preferences. The defeasible implication, disjuntion, conjunction, implication, bi-implication and negation symbols may all be altered.

6.2.11 Adjustable Character Set. The user may select an available character set from which to generate atoms. These being upperlatin [capital Latin alphabet], lowerlatin [lowercase Latin alphabet], altlatin [an assortment of alternate Latin characters] and greek [lowercase Greek alphabet].

6.2.12 Generator Type. Determines which generator type is used for creating the knowledge base. The options are 's' (Standard Generator) and 'o' (Optimised Generator).

6.2.13 Print and Export. The knowledge base may then be printed out to the screen and exported to a text file, to be used to test an entailment relation.

6.2.14 Regenerate, Change settings and Quit. The user may then regenerate another knowledge base of the same configuration, which would re prompt them to choose a generator type, else they may choose to change the knowledge base specification or quit the program.

7 EXPERIMENT DESIGN AND EXECUTION

Testing was done using a desktop PC with a 6-core Ryzen 5 3600 with 16GB RAM. The correctness of the generators was evaluated by ranking the knowledge bases using the BaseRank algorithm developed by Evashna Pillay as part of last years SCADR2 project [12].

7.1 Correctness Testing

Knowledge bases were generated and ranked using the BaseRank algorithm. The output obtained from BaseRank was compared to that produced by the generator in order to assess its accuracy.

7.2 Execution Time Testing

Tests were conducted using both the original and optimized generators. A consistent set of settings were applied across all tests, with the reuseConsequent flag set to true.
Both simple defeasible implications and complex defeasible implications were tested. The mixedDefImplication function was used for generating complex DIs, with the antecedent and consequent both being a complexity of 2.
The generated ranks varied in number, specifically 10, 50, 100, 150, and 200. For each rank, a variety of defeasible implications were generated (25000, 30000, 35000, 40000 and 45000 DIs). Only repeatable distributions were tested, those being flat, linear-growth, and linear-decline.
In order to obtain reliable performance metrics, each test scenario was executed five times. The results from these repetitions were averaged, with each run preceded by a warm-up phase to stabilize the system.

8 RESULTS AND ANALYSIS

The tabulated data for the tests can be found in Appendix A.

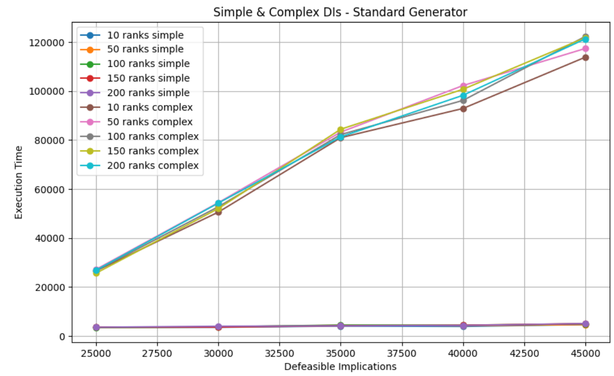
8.1 Correctness

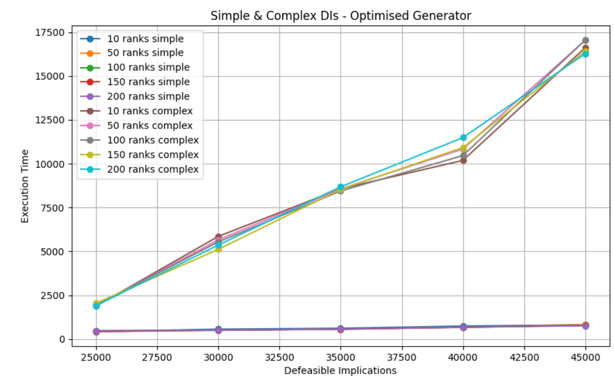
The output from the generators was found to be correct every time when tested using the BaseRank algorithm.

8.2 Analysis of Results

Based on the tabulated data in Appendix A, it's clear that variations in the number of ranks and their distribution have minimal impact on the generators' execution times. We can also see that as the number of generated DIs increases, the execution time for creating simple DIs doesn't increase as drastically as it does for generating complex DIs.

8.2.1 Simple vs Complex DI Comparison. The following graphs show the execution time vs defeasible implication count for generating simple and complex DIs, using both the standard and optimised generators. The average execution times for the distributions were used in these graphs.



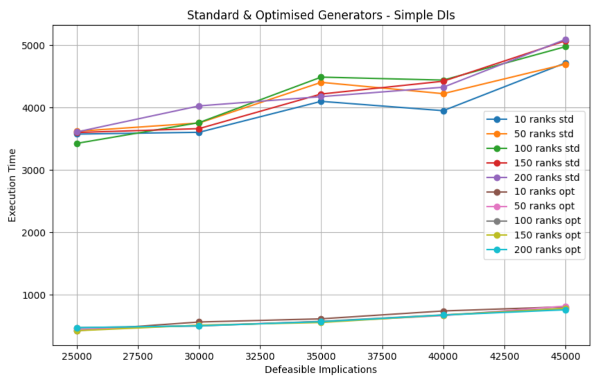


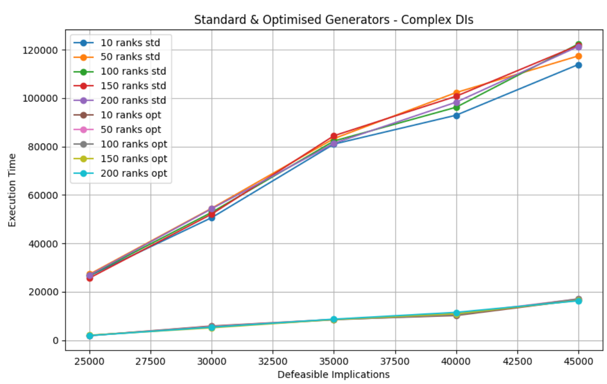
The numbers used to calculate the following percentage increase in execution times was the average execution time over the ranks, per the number of DIs.

- The percentage increase in execution time when generating 25000 simple vs 25000 complex DIs using the standard generator is [(26658-3566)/3566]*100 = 647.5 percent increase.
- The percentage increase in execution time when generating 45000 simple vs 45000 complex DIs using the standard generator is [(119331-4908)/4908]*100 = 2331.3 percent increase.
- The percentage increase in execution time when generating 25000 simple vs 25000 complex DIs using the optimised generator is [(1941-449)/449]*100 = 332.2 percent increase.
- The percentage increase in execution time when generating 45000 simple vs 45000 complex DIs using the optimised generator is [(16673-790)/790]*100 = 2010.5 percent increase.

From these results we can see the execution time to generate complex DIs is a significant increase over generating only simple DIs, with this being true for both the standard and optimised generator. We can see that the greater the number of DIs generated leads to an even greater difference between the generation of simple and complex DIs. It is also evident that the standard generator had a larger percentage increase in execution time between generating simple and complex DIs than the optimised generator for both 25000 and 45000 DIs.

8.2.2 Standard vs Optimised Gen Comparison. The next graphs show the execution time vs defeasible implication count between the standard and optimised generators, both simple and complex DIs were graphed. The average execution times for the distributions were used in these graphs.





The numbers used to calculate the following percentage increase in execution times was the average execution time over the ranks, per the number of DIs.

- The percentage increase in execution time when generating 25000 simple DIs vs 45000 simple DIs using the standard generator is [(4908-3566)/3566]*100 = 37.6 percent.
- The percentage increase in execution time when generating 25000 simple DIs vs 45000 simple DIs using the optimised generator is [(790-449)/449]*100 = 75.9 percent.
- The percentage increase in execution time when generating 25000 complex DIs vs 45000 complex DIs using the standard generator is [(119331-26658)/26658]*100 = 347.6 percent.
- The percentage increase in execution time when generating 25000 complex DIs vs 45000 complex DIs using the optimised generator is [(16673-1941)/1941]*100 = 758.9 percent increase in execution time.
- The percentage increase in execution time when generating 25000 simple DIs using the standard and optimised generators [(3566-449)/449]*100 = 693.5 percent.
- The percentage increase in execution time when generating 45000 simple DIs using the standard and optimised generators [(26658-1941)/1941]*100 = 1270.5 percent.
- The percentage increase in execution time when generating 25000 complex DIs using the standard and optimised generators [(4908-(4908/790)/790]*100 = 620.2 percent.
- The percentage increase in execution time when generating 45000 complex DIs using the standard and optimised generators [(119331-16673)/16673]*100 = 617.2 percent.

From the results we can see that the optimised generator was considerably faster than the standard generator at generating both simple and complex DIs, with the optimised generator being 693.5 percent and 1270.5 percent faster when generating 25000 simple and complex DIs, and 620.2 percent and 617.2 percent faster when generating 45000 simple and complex DIs respectively. However, the optimised generator had a larger percentage increase in execution time between generating 25000 DIs and 45000 DIs than the standard generator, with this being the case in both the generation of simple (75.9 vs 37.6 percent increase) and complex defeasible implications (758.9 vs 347.6 percent increase).

9 CONCLUSIONS

The aims for this project were reached, with a parameterized non-deterministic defeasible knowledge base generator that generates according to Rational Closures BaseRank successfully implemented. As well as this, the optimised variant was also developed. The performance of the generators was analysed, and the optimised generator was found to have significant speedup over the standard generator, however it was also found to have a larger percentage increase in execution time than the standard generator, when the number of defeasible implications was increased. This was true for both the generation of simple and complex defeasible implications. Knowledge bases with complex defeasible implications were found to take longer to generate than those with only simple ones, with the standard generator having a larger percentage increase in execution time between generating simple and complex DIs than the optimised generator.

10 FUTURE WORK

With the ability to generate knowledge bases of complicated defeasible implications, future researchers can use this tool to perform more in-depth analysis on defeasible entailment relations. Further improvements can be made to the pseudo-random reuse of atoms in DIs, as this could lead to large performance gains in the generation of complex defeasible implications. As well as this, improvements to the optimised generator's thread management could be prove to make the drop off in performance when generating large numbers of defeasible implications less prevalent.

REFERENCES

[1] Mordechai Ben-Ari. 2012. *Mathematical Logic for Computer Science (3 ed.)*. Springer Science Business Media, Rehovot, Israel.
[2] Ronald J. Brachman and Hector J. Levesque. 2004. Chapter 1 - Introduction. In *Knowledge Representation and Reasoning*, Ronald J. Brachman and Hector J. Levesque (Eds.). Morgan Kaufmann, San Francisco, 1–14. https://doi.org/10.1016/B978-155860932-7/50086-8
[3] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2019. Taking Defeasible Entailment Beyond Rational Closure. In *Logics in Artificial Intelligence*, Francesco

Calimeri, Nicola Leone, and Marco Manna (Eds.). Springer International Publishing, Cham, 182–197.
[4] 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
[5] Laura Giordano, Valentina Gliozzi, Nicola Olivetti, and Gian Luca Pozzato. 2015. Semantic characterization of rational closure: From propositional logic to description logics. *Artif. Intell.* 226 (2015), 1–33.
[6] A. Kaliski. 2020. *An overview of KLM-style defeasible entailment.* Master's thesis. University of Cape Town, Cape Town, South Africa.
[7] Robert Koons. 2022. Defeasible Reasoning. In *The Stanford Encyclopedia of Philosophy* (Summer 2022 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.
[8] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence* 44, 1 (1990), 167–207. https://doi.org/10.1016/0004-3702(90)90101-5
[9] Daniel Lehmann. 2002. Another perspective on Default Reasoning. arXiv:cs/0203002 [cs.AI]
[10] 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
[11] Drew McDermott and Jon Doyle. 1980. Non-monotonic logic I. *Artificial Intelligence* 13, 1 (1980), 41–72. https://doi.org/10.1016/0004-3702(80)90012-0 Special Issue on Non-Monotonic Logic.
[12] Evashna Pillay. 2022. *An Investigation into the Scalability of Rational Closure V2.* Honours Project. Faculty of Science, University of Cape Town. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2022/pillay_thakorvallabh.zip/files/PLLEVA005_SCADR2_FinalPaper.pdf
[13] Stuart J. Russell and Peter Norvig. 2022. *Artificial Intelligence: A modern approach.* Pearson Education Limited.

Appendix A TESTING RESULTS

Ranks	flat	linear-growth	linear-decline	average
25000 DIs				
10	3554ms	3624ms	3547ms	3575ms
50	3769ms	3591ms	3500ms	3620ms
100	3312ms	3409ms	3562ms	3427ms
150	3408ms	3750ms	3642ms	3600ms
200	3468ms	3618ms	3745ms	3610ms
30000 DIs				
10	3697ms	3484ms	3631ms	3604ms
50	3844ms	3887ms	3529ms	3753ms
100	3942ms	3610ms	3724ms	3759ms
150	3767ms	3527ms	3697ms	3663ms
200	4032ms	4140ms	3909ms	4027ms
35000 DIs				
10	4115ms	4217ms	3969ms	4100ms
50	4430ms	4581ms	4201ms	4404ms
100	4268ms	4583ms	4614ms	4488ms
150	4314ms	4418ms	3921ms	4217ms
200	4199ms	4110ms	4216ms	4175ms
40000 DIs				
10	3822ms	4125ms	3903ms	3950ms
50	4115ms	4238ms	4316ms	4223ms
100	4386ms	4482ms	4452ms	4440ms
150	4481ms	4549ms	4237ms	4422ms
200	4399ms	4197ms	4383ms	4326ms
45000 DIs				
10	4605ms	4650ms	4887ms	4714ms
50	4727ms	4576ms	4782ms	4695ms
100	4869ms	5130ms	4926ms	4975ms
150	5047ms	5071ms	5088ms	5069ms
200	5145ms	5372ms	4758ms	5091ms

Figure 4: Original gen - Simple

Ranks	flat	linear-growth	linear-decline	average
25000 DIs				
10	427ms	431ms	417ms	425ms
50	450ms	435ms	456ms	447ms
100	437ms	451ms	535ms	474ms
150	438ms	425ms	414ms	425ms
200	450ms	527ms	446ms	474ms
30000 DIs				
10	561ms	553ms	578ms	564ms
50	522ms	508ms	498ms	509ms
100	496ms	524ms	482ms	500ms
150	522ms	529ms	487ms	513ms
200	498ms	502ms	509ms	503ms
35000 DIs				
10	636ms	599ms	609ms	615ms
50	603ms	538ms	567ms	569ms
100	565ms	570ms	592ms	575ms
150	598ms	481ms	588ms	556ms
200	569ms	564ms	574ms	569ms
40000 DIs				
10	735ms	743ms	746ms	741ms
50	658ms	686ms	654ms	666ms
100	682ms	697ms	656ms	678ms
150	687ms	703ms	634ms	674ms
200	673ms	660ms	692ms	675ms
45000 DIs				
10	807ms	802ms	816ms	808ms
50	809ms	813ms	835ms	819ms
100	783ms	797ms	772ms	784ms
150	796ms	768ms	779ms	781ms
200	765ms	774ms	740ms	760ms

Figure 5: Optimised gen - Simple

Ranks	flat	linear-growth	linear-decline	average
25000 DIs				
10	27946ms	26902ms	25010ms	26619ms
50	26338ms	28183ms	27191ms	27271ms
100	25795ms	27852ms	26731ms	26759ms
150	25721ms	26525ms	24993ms	25713ms
200	26181ms	27860ms	26755ms	26932ms
30000 DIs				
10	48398ms	52691ms	50893ms	50627ms
50	54649ms	55263ms	53583ms	54498ms
100	50957ms	53940ms	53774ms	52824ms
150	51277ms	52374ms	52950ms	52100ms
200	53806ms	55881ms	53269ms	54319ms
35000 DIs				
10	77391ms	86347ms	79257ms	80998ms
50	83348ms	87027ms	79424ms	83266ms
100	81992ms	80663ms	84223ms	82259ms
150	87063ms	81911ms	84265ms	84446ms
200	79175ms	82650ms	82169ms	81331ms
40000 DIs				
10	92449ms	95432ms	91075ms	92919ms
50	102920ms	105785ms	97114ms	102273ms
100	92727ms	98441ms	97458ms	96209ms
150	94433ms	110407ms	95406ms	100749ms
200	91326ms	109536ms	93027ms	98296ms
45000 DIs				
10	117306ms	110251ms	113960ms	113839ms
50	118895ms	119253ms	115077ms	117408ms
100	121739ms	120915ms	125292ms	122315ms
150	123683ms	120602ms	121062ms	121782ms
200	119047ms	121085ms	124802ms	121311ms

Figure 6: Original gen - Complex

Ranks	flat	linear-growth	linear-decline	average
25000 DIs				
10	1748ms	1821ms	2088ms	1886ms
50	1785ms	1987ms	1979ms	1917ms
100	1920ms	2002ms	1929ms	1950ms
150	1850ms	2280ms	2014ms	2048ms
200	1783ms	1856ms	2085ms	1908ms
30000 DIs				
10	5671ms	5980ms	5921ms	5857ms
50	5524ms	5912ms	5553ms	5663ms
100	5183ms	5811ms	5637ms	5544ms
150	5087ms	5128ms	5136ms	5117ms
200	5057ms	5222ms	5819ms	5366ms
35000 DIs				
10	8488ms	8788ms	8427ms	8568ms
50	8552ms	8813ms	8328ms	8564ms
100	8297ms	8147ms	8888ms	8444ms
150	8887ms	8012ms	8713ms	8537ms
200	8948ms	8201ms	8893ms	8681ms
40000 DIs				
10	10560ms	10889ms	9906ms	10185ms
50	10013ms	11867ms	10531ms	10837ms
100	10009ms	10303ms	10969ms	10460ms
150	9995ms	11507ms	11248ms	10917ms
200	11396ms	10880ms	11986ms	11487ms
45000 DIs				
10	16463ms	17029ms	16340ms	16611ms
50	17009ms	17264ms	16817ms	17030ms
100	16405ms	17606ms	17128ms	17046ms
150	16096ms	16166ms	16962ms	16408ms
200	16515ms	15926ms	16375ms	16272ms

Figure 7: Optimised gen - Complex

Appendix B CLASS DESCRIPTIONS

- *App* class: used to control the overall execution and coordination of various components, including knowledge base generation and analysis.
- *Atom* class: represents an atomic proposition and encapsulates the properties and behavior associated with individual atoms
- *AtomBuilder* class: provides functions for generating and keeping track of atoms in a knowledge base.
- *Connective* class: contains logical connectives used to build complex DIs within the knowledge base.
- *DefImplication* class: represents a defeasible implication and provides functions to manipulate and access its components
- *DefImplicationBuilder* class: provides functions for generating diverse types of DIs.
- *Distribution* class: contains functions to control the distribution of DIs.
- *KBGenerator* class: controls the generation of defeasible knowledge bases.
- *KBGeneratorThreaded* class: provides an optimised version of defeasible knowledge base generation.
- *Rules* class: defines and manages the rules controlling the generation of complex defeasible implications.

Appendix C UML DIAGRAM