UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS Honours Project Final Paper 2024

Title: Defeasible Conditionals in Answer Set Programming

Author: Racquel Nina Dennison

Project Abbreviation: DCASP

Supervisor(s): Tommie Meyer, Jesse Heyninck

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	20
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	5
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*	0
Total marks		**80**	

Defeasible Conditionals in Answer Set Programming

Racquel Nina Dennison
dnnrac003@myuct.ac.za
University of Cape Town
Cape Town, Western Cape, South Africa

Abstract

Defeasible reasoning, a key area within Knowledge Representation and Reasoning (KRR), handles the challenges of modelling and inferring conclusions from incomplete information. A well-known approach to modelling defeasible reasoning is the KLM framework defined by Kraus, Lehmann and Magdior. KLM defines the fundamental properties necessary for modelling defeasible entailment relations. Traditionally, modelling defeasible entailment within the KLM framework has taken an imperative approach, with minimal study done on modelling defeasible entailment with declarative languages. This paper explores the feasibility and advantages of using a declarative approach to computing the reasoning patterns described in the KLM framework. The declarative language we investigate is Answer Set Programming (ASP). ASP is a logic language orientated towards solving complex problems. To analyse the usability of ASP within the KLM framework, we define a declarative approach to computing Rational Closure (RC). RC is a well-known defeasible reasoning algorithm that satisfies the KLM properties required for modelling entailment. To the best of our knowledge, our declarative definition of RC is the first approach to be defined using ASP. Additionally, we develop a parameterised knowledge base generator to create data for testing RC. Our experimental analysis evaluates the knowledge base generation times with different parameters, and we explore potential reductions in generation time through fine-tuning configurations provided by the ASP solver used in the study.

CCS Concepts

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

Keywords

Artificial Intelligence, Knowledge Representation and Reasoning, Defeasible Reasoning, Rational Closure, Answer Set Programming.

1 Introduction

Formal logic has a long-standing history in addressing reasoning problems within Artificial Intelligence (AI) [20]. Representing information with formal logic enables algorithmic modelling of reasoning patterns [34]. Traditionally, reasoning algorithms were monotonic, meaning that adding new information to a domain did not invalidate previously established conclusions [21]. However, monotonicity presented challenges when modelling a domain's characteristics, as new information sometimes contradicted or served as an exception to existing rules.

Nonmonotonic reasoning aimed to address the limitations of monotonic reasoning by incorporating mechanisms to handle exceptions [5]. Nonmonotonic reasoning involves making and revising assumptions previously made when new information is learnt. It is based on the notion of *typicality*, which means that not all properties of a given object necessarily hold [14]. For example, instead of asserting that "all birds fly," we instead say that "birds *typically* fly" as there are exceptional cases such as ostriches and penguins. By defining the characteristics of an object to "*typically*" hold, we are making provisions for exceptions that could arise.

Defeasible reasoning, a nonmonotonic approach, offers a framework for modelling this type of reasoning. Among the various approaches to defeasible reasoning, the KLM framework developed by Kraus, Lehmann, and Magidor [24] is well-known for defining the fundamental properties any defeasible reasoning framework should follow when dealing with incomplete information [12]. A notable reasoning pattern that satisfies the KLM properties is Rational Closure (RC). RC is a conservative form of defeasible entailment, meaning that it infers minimal assumptions from the knowledge base [8]. RC is defined both semantically and algorithmically.

The existing approaches for computing entailment relations that satisfy the KLM properties have been defined using imperative algorithms [8] and implemented in imperative languages like Java and Python.

In this paper, we explored the potential benefits of using a declarative approach for modelling defeasible entailment under the KLM framework. Declarative programming languages offer advantages over their imperative counterparts, such as simpler modification of program constraints without requiring a rework of the entire algorithm [30]. The declarative language we explored is Answer Set Programming (ASP). To assess the effectiveness of using ASP within the KLM framework, we defined and implemented a declarative approach to computing RC. Additionally, we designed and developed a parameterised knowledge base generator that produces data for testing RC. Our experiments evaluated the influence different parameterisations have on the generation time of knowledge bases. Moreover, we tested the knowledge generator's performance using predefined fine-tuning configurations provided by the ASP solver. Specifically, we evaluated four configurations tailored for particular computing problems in ASP. Our experiments allowed us to evaluate that specifying parameters such as a uniform distribution leads to a longer generation time. Increasing the number of statements and ranks in a knowledge base also impacted the generation time. The configurations used showed interesting results. We expected the configurations to influence the generation time by decreasing the generation time; however, their use demonstrated that the standard ASP solver used in this paper was sufficient for solving the problem of generating knowledge bases.

2 Background

In this section, we provide the relevant background material. Propositional logic is the foundational logic used in this paper and is outlined in Section 2.1. RC is defined in Section 2.3, and finally, the relevant background needed for ASP is defined in Section 2.4.

2.1 Propositional Logic

Propositional logic is a framework for modelling information [4]. Statements (otherwise known as formulas) are built up using *propositional atoms*, which are symbols that are either true or false and contain no logical connectives [13]. Propositional logic formalises information representation and facilitates deriving conclusions from a provided set of statements [4].

2.1.1 Language: We denote the set of all *propositional atoms* by the set $\mathcal{P}$. *Propositional atoms* are assigned a *truth value*, true or false, [4] and are combined with boolean operators to create well-defined propositional formulas, otherwise referred to as formulas. Formulas are denoted by Greek letters such as α, β, and γ. The language, denoted as $\mathcal{L}$, is a set of formulas which are represented as, for any given $p \in \mathcal{P}$ and $\tau, \alpha \in \mathcal{L}$, τ can take on the following form: $p, \tau, \neg\tau, \tau \wedge \alpha, \tau \rightarrow \alpha, \tau \leftrightarrow \alpha, \tau \vee \alpha$ [4]. A knowledge base, $\mathcal{K} \subseteq \mathcal{L}$, is a finite set of propositional atoms. Knowledge bases are used when modelling facts about a specific domain.

2.1.2 Semantics: To assign meaning to *propositional atoms*, we define an interpretation ω, which maps propositional atoms to truth values. An interpretation ω is defined as $\omega : \mathcal{P} \rightarrow \{T, F\}$ [4]. The total amount of interpretations for a given set $\mathcal{P}$ is $2^{|\mathcal{P}|}$, where $|\mathcal{P}|$ is the number of elements in $\mathcal{P}$ [4]. The set $\mathcal{U}$ represents all possible interpretations of $\mathcal{P}$ [21]. For example, given $\mathcal{P}_1 = \{p, q\}$, where p and q are atoms, p indicates p is true under some interpretation and $\overline{p}$ indicates that p is false under some interpretation. There are 2^2 possible interpretations for $\mathcal{P}_1$, namely, $\mathcal{U} = \{pq, \overline{p}q, \overline{pq}, p\overline{q}\}$. We denote the value of a formula under an interpretation as $\mathcal{I}(\alpha)$. If $\mathcal{I}(\alpha)$ is true for some formula α, we say that $\mathcal{I}$ satisfies α, denoted as $\mathcal{I} \Vdash \alpha$. From our previous example set, $\mathcal{P}_1$, the interpretation $\overline{p}q$ satisfies q denoted as $\overline{p}q \Vdash q$ as q is true in this interpretation. $\mathcal{I}$ satisfies a knowledge base $\mathcal{K}$ if, for all $\alpha \in \mathcal{K}$, $\mathcal{I} \Vdash \alpha$ [21]. If at least one interpretation $\mathcal{I}$ satisfies $\mathcal{K}$, then $\mathcal{I}$ is a *model* of $\mathcal{K}$. The set of all models of $\mathcal{K}$ is denoted as $\text{mod}(\mathcal{K})$. If there exists no $\mathcal{I} \in \mathcal{U}$ such that $\mathcal{I} \Vdash \mathcal{K}$, then $\mathcal{K}$ contains no interpretation satisfying all $\alpha \in \mathcal{K}$. If no $\mathcal{I} \Vdash \mathcal{K}$, $\mathcal{K}$ is said to be unsatisfiable.

2.1.3 Entailment: Classical reasoning allows us to formally define how inferences are drawn from a knowledge base and how to determine whether some formula, α, follows from a knowledge base $\mathcal{K}$. Drawing inferences from a knowledge base is defined as entailment. Entailment is denoted as $\models$. Formally, if some $\alpha \in \mathcal{L}$ is true in every $mod(\mathcal{K})$, i.e $mod(\mathcal{K}) \subseteq mod(\alpha)$, then $\mathcal{K} \models \alpha$.
Classical propositional logic laid the groundwork for many reasoning extensions [21] however, classical entailment is monotonic by definition. Monotonic means adding more information to the knowledge base does not lead to previous inferences being withdrawn [15]. Monotonicity presents challenges in modelling human reasoning patterns as they are nonmonotonic [23]. Defeasible reasoning aims to address this.

2.2 Defeasible Reasoning

Defeasible reasoning is a nonmonotonic framework that permits the retraction of previously established rules when new, contradictory information is introduced to a knowledge base [16]. Various formalisations have been developed to model defeasible reasoning, including belief revision [2], default logic [19], and the preferential approach [33]. The preferential approach has been extended into what is known as the KLM framework [24], established by Kraus, Lehmann, and Magidor. This framework is the paper's primary focus due to its well-defined properties and computationally efficient reasoning algorithms, one of which, RC [8], will be the main focus of this paper.

2.2.1 KLM Approach: The KLM framework introduces the concept of *defeasible implications*, which allows reasoning algorithms to deal with statements that hold for most cases but are exceptional in others. Defeasible implications take on the form of $\mid\sim$ and are expressed as $\alpha \mid\sim \beta$, which means that if α is true, then this is enough information to conclude that β is true [21]. Drawing inferences from a set of defeasible implications is called defeasible entailment. The notion of defeasible entailment ($\mid\approx$) is not unique; many acceptable ways of defining defeasible entailment have been outlined. The KLM approach is of interest in this paper due to its properties, which are known as the KLM properties [24]. The KLM properties suggest that any defeasible entailment procedure should adhere to several rationality properties. Defeasible entailment procedures that satisfy these properties are LM-rational. RC is an example of an LM-rational procedure.

2.3 Rational Closure

RC is the most conservative form of defeasible reasoning, which means that the reasoning algorithm infers as little as possible from a knowledge base [21]. The framework is defined semantically as well as algorithmically [8]. This paper will focus on the algorithmic definition of RC as outlined in [8]. Computing defeasible entailment with RC requires two algorithms, Base Rank and RC.

2.3.1 Base Rank: Base Rank is a procedure which assigns a numerical rank to statements in a knowledge base. More generalised statements, which apply broadly to a particular topic, are given lower ranks, while more exceptional statements which violate established rules are assigned higher ranks. For example, if given the following: $\mathcal{K} = \{man \rightarrow moral,\ Socrates \rightarrow man,\ Socrates \rightarrow \neg mortal\}$, in this knowledge base, the statement $Socrates \rightarrow \neg mortal$ is exceptional while $man \rightarrow moral$ is a generalised statement to the class of men. Before we outline Base Rank, we establish the following definitions:
Definition 2.3.1.1: The materialisation of a knowledge base $\mathcal{K}$ is defined as $\overrightarrow{\mathcal{K}} := \{\alpha \rightarrow \beta \mid \alpha \mid\sim \beta \in \mathcal{K}\}$ [21].
Definition 2.3.1.2 A formula α is exceptional in the set $\mathcal{K}$ if $\forall u \in \mathcal{U}$, where $\mathcal{U}$ is the set of all interpretations, $u \models \neg\alpha$ [21].
To rank statements based on their *exceptionality*, Base Rank first computes the materialisation of a defeasible knowledge base, then ranks statements using the materialised knowledge base. Statements are ranked based on how specific they are to a specified domain, with classical statements placed on the infinite rank. Algorithm 1 outlines this procedure.

Algorithm 1 BaseRank

Input: A knowledge base $\mathcal{K}$
Output: An ordered tuple (R_0,..., R_{n-1},R_∞,n). Each R_i, where $i \in [0, n]$ indicates the rank each statement is placed on.
$i = 0$;
$E_0 = \vec{\mathcal{K}}$;
while $E_{i-1} \neq E_i$ **do**
    $E_{i+1} = \{ \alpha \rightarrow \beta | E_i \models \neg\alpha \}$;
    $R_i = E_i \backslash Ei + 1$;
    $i = i + 1$
end while
$R_\infty = E_{i-1}$;
if $E_{i-1} =$ **then**
    $n = i - 1$;
else
    $n = i$;
end if
return (R_0,..., R_{n-1},R_∞,n);

2.3.2 Rational Closure: RC computes whether a knowledge base entails a given implication. For a given query $\alpha \mid\sim \beta$ or $\alpha \rightarrow \beta$, α is the antecedent and β is the consequence. To compute defeasible entailment for a given query, $\alpha \mid\sim \beta$, RC takes a defeasible knowledge base, computes Base Rank, then eliminates statements within the knowledge base which makes the antecedent, α, exceptional [21]. RC returns true if the materialisation of $\alpha \mid\sim \beta$ is classically entailed by $\vec{\mathcal{K}}$ once all the inconsistent statements have been eliminated. Else, it will return false. Algorithm 2 outlines the RC procedure.

Algorithm 2 RationalClosure

Input: A knowledge base $\mathcal{K}$ and a defeasible implication $\alpha \mid\sim \beta$.
Output: true, if the query is entailed by the knowledge base, else **false**.
(R_0,..., R_{n-1},R_∞,n) = $BaseRank(\mathcal{K})$;
$i = 0$;
$\mathcal{R} = \bigcup_{i=0}^{j<n} \mathcal{R}_j$
while $\mathcal{R}_\infty \cup \mathcal{R} \models \neg\alpha$ *and* $\mathcal{R} \neq \varnothing$ **do**
    $\mathcal{R} = \mathcal{R} \backslash \mathcal{R}_i$;
    $i = i + 1$;
end while
return $\mathcal{R}_\infty \bigcup \mathcal{R} \models \alpha \rightarrow \beta$;

To make sense of both algorithms, consider the following example. Given the knowledge base $\mathcal{K}_1 = \{fruit \mid\sim sweet, fruit \mid\sim seeds, lemon \rightarrow fruit, lemon \mid\sim \neg sweet\}$ and the query, $lemon \mid\sim seeds$. The antecedent of the query is $lemon$. The materialisation of $\mathcal{K}_1$ is $\{fruit \rightarrow sweet, fruit \rightarrow seeds, lemon \rightarrow fruit, lemon \rightarrow \neg sweet\}$ and the Base Rank for $\mathcal{K}_1$ is outlined in Table 1.

Table 1: BaseRank of knowledge base $\mathcal{K}_1$

$\mathcal{R}_\infty$	$lemons \rightarrow fruit$
$\mathcal{R}_1$	$lemon \rightarrow \neg sweet$
$\mathcal{R}_0$	$fruit \rightarrow sweet,\ fruit \rightarrow seeds$

The initial step is to check if $\mathcal{R} \cup \mathcal{R}_\infty \models \neg lemon$; this is done by computing the models of $\mathcal{K}_1$ and checking if all interpretations make $\neg lemon$ true. We denote lemon as l, sweet as w, seeds as s and fruit as f. The models of $\mathcal{K}$ is $mod(\mathcal{K}) = \{f\bar{l}ws, \overline{f}lws, \overline{f}l\bar{w}s, \overline{f}l\overline{ws}, \overline{flws}\}$. There exists no model u in $mod(\mathcal{K})$ where $u \Vdash lemon$, this means $\mathcal{R} \cup \mathcal{R}_\infty \models \neg lemon$. This results in the most preferred rank, i.e. rank 0, being eliminated from the Base Rank as shown in Table 2.

Table 2: BaseRank of knowledge base $\mathcal{K}_1$ with the lowest rank eliminated.

$\mathcal{R}_\infty$	$lemons \rightarrow fruit$
$\mathcal{R}_1$	$lemon \rightarrow \neg sweet$
$\mathcal{R}_0$	~~$fruit \rightarrow sweet,\ fruit \rightarrow seeds$~~

With the remaining statements in $\mathcal{R} \cup \mathcal{R}_\infty$, it is checked again if $\mathcal{R} \cup \mathcal{R}_\infty \models \neg lemon$. The revised models of $\mathcal{K}_1$ is $\{f\bar{l}ws, \overline{f}lws, \overline{f}l\bar{w}s, \overline{f}l\overline{ws}, \overline{flws}, f\bar{l}w\bar{s}\}$. We see that $\mathcal{R} \cup \mathcal{R}_\infty \not\models \neg lemon$ as there exists an interpretation which makes $lemon$ true. Computing classical entailment on $\mathcal{R} \cup \mathcal{R}_\infty$, we see that no model within the remaining ranks makes the statement $lemon \rightarrow seeds$ true, thus under RC, $\mathcal{K} \not\models lemon \mid\sim seeds$.

2.4 Answer Set programming

ASP is a modern approach to declarative programming, which has been designed to solve complex problems [28]. ASP is recognized for its ease of use, expressive constructs, and computational efficiency [30]. ASP consists of rules similar to those in Prolog [11] and has computational methods that leverage the concepts behind efficient satisfiability solvers [30]. Solving ASP programs involves determining stable models [27]. A stable model is a set of atoms which satisfy the rules specified in an ASP program.

2.4.1 Problem-solving in ASP paradigm: Solving problems in the ASP paradigm involves encoding problems into programs that can be processed by ASP tools like gringo [18] and clasp [17]. Finding the answer sets of an ASP-encoded problem involves two stages. The first stage, grounding, takes a logic program and replaces all variables with constants, producing a fully instantiated program. The grounder performs this. In this paper, we will be using gringo as the grounder. Grounded programs are then passed to the solver, which uses conflict-driven [30] techniques to compute the answer sets for a given program. Conflict-driven solving techniques resolve and handle integrity violations when finding answer sets [30]. The solver used in this paper is clasp [17]. Clingo [30] combines both gringo and clasp into a monolithic system and is a programming tool based on the ASP programming paradigm. We will be using clingo to compute ASP programs.

2.4.2 Language: Rules in ASP take the form:

$$a_0 :- a_1, a_2, ..., a_m, not\ a_{m+1}, .., not\ a_n.$$

Given a rule represented as r, we denote the $head(r) = \{a_0\}$ and the body $body(r) = \{a_1, a_2, ..., a_m, not\ a_{m+1}, .., not\ a_n\}$. Both the head and the body of the rule consist of literals. Literals are defined as atoms that are either positive or negative [30]. The set of positive literals in the body is denoted by $body^+(r) = \{a_1, a_2, ..., a_m\}$

while the set of negative literals is $body^{-}(r) = \{a_{m+1}, a_2, ..., a_n\}$. Intuitively, if we have a set $\mathcal{X}$ which consists of literals within the program then if $body^{+}(r) \subseteq \mathcal{X}$ and $body^{-}(r) \cap \mathcal{X} = \varnothing$ then $head(r) \in \mathcal{X}$ [30]. Answer sets are sets of literals that satisfy the rules of a program. Answer sets are the minimal stable models of an ASP program [3].

2.4.2.1 Choice Rules: ASP programs, which consist of several answer sets, utilise choice rules. Choice rules describe several alternative ways to form a stable model with the head of the rule consisting of an expression in braces [30], for example:

$$\{b(a); c(d)\}. \tag{1}$$

The rule above states all possible ways the two literals $b(a)$ and $c(d)$ can be included in the stable model. There are four possible choices, which is the power set of $\{b(a), c(d)\}$ namely,

$$\{\{\varnothing\}, \{b(a)\}, \{c(d)\}, \{b(a); c(d)\}\} \tag{2}$$

2.4.2.2 Optimisation: Clingo offers aggregate functions that instruct the solver to find a stable model based on some characteristic. The function we are interested in is the *#minimize* function. The *#minimize* takes the form $\#minimize\{X : p(X)\}$, where X represents the variable to be minimised, and $p(X)$ indicates the condition that must hold for the function to be considered.

For example, the travelling salesman problem is a combinatorial optimisation problem that finds the shortest path between cities with minimal cost [9]. Encoding the total path cost as $cost(C)$ to represent the sum of all paths between cities, we can use $\#minimize\{C : cost(C)\}$ to instruct the solver to improve each stable model, ensuring that the next stable model found has a lower cost than the previous one.

2.4.3 ASP encoding example: Idiomatic ASP problems are typically separated into problem instances and problem classes/encodings. The problem instance is a set of facts, while the problem class is a set of rules that will satisfy a class of facts. We will consider the graph colouring problem as an example. The graph colouring problem describes assigning a specified set of colours to nodes on a graph while ensuring that any nodes connected by an edge are not coloured with the same colour [31]. The problem instance in Listing 1 represents the data points of the graph 1 with a selected set of colours, blue, red and green, denoted as b,r,g.

```
1 node(1..3).
2 edge(1,2; 2,3; 3,1).
3 col(r;b;g).
```

Listing 1: Problem instance representing Figure 1

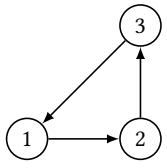

Figure 1: Graph coloring example

The encoding of the problem is outlined in Listing 2. The methodology many ASP programs follow is *generate, define and test* [26]. The *generate* method is meant to non-deterministically provide possible candidate solutions while *test* eliminates the solutions violating the requirements. *Define* provides definitions for auxiliary rules that make up the solution. Line 1 of Listing 2, the *generate* rule states that for each node, exactly one colour should be assigned to it, so given three colours, an assignment of 3 colours will be given to each node. The *test* rule in line 2 eliminates solutions where two nodes X, Y are connected and assigned the same colour. The *#show* directive on line 4 indicates to the solver to print the colouring of nodes and not the other predicates in the solution candidate.

```
1 {color(X,C) : col(C)} = 1 :- node(X).
2 :- edge(X,Y), color(X,C), color(Y,C).
3 #hide.
4 #show color/2.
```

Listing 2: Program encoding graph coloring

Listing 3 outlines a solution candidate.

```
1 Answer: 1
2 color(1,b) color(2,g) color(3,r)
```

Listing 3: Program encoding graph coloring

3 Project Aims

The project aimed to qualitatively assess the usefulness of the ASP paradigm when computing reasoning algorithms defined by the KLM framework. We implemented multiple versions of Base Rank and RC to achieve this. One version is outlined in this paper, while the others are outlined in [29]. Additionally, we created a parameterised knowledge base generation tool that generates test sets for RC. We ran experiments on our knowledge base. The aim was to evaluate the influence of parameter configurations on generation time. Furthermore, we explored different fine-tuning configurations offered by the ASP solver when testing our knowledge base. This evaluated if any configurations resulted in a reduction in the generation time of knowledge bases. The work done in this paper is of a larger project in collaboration with Jack Mabotja [29] and Sibusiso Buthelezi [6]. Sibusiso Buthelezi's contributions centre around developing an application to explain the reasoning process of RC. Jack Mabotja's contributions optimised the work done on the definitions of Base Rank and RC in this paper.

4 Declarative approach to Base Rank and Rational Closure

The following section outlines our ASP implementation of Base Rank and RC. We describe the problem instance and problem class of each implementation.

4.1 Base Rank

Base Rank ranks statements based on their level of generality, where statements that are more specific concerning a particular class of objects will be ranked on higher levels. When a knowledge base contains a formula, α and $\mathcal{K} \mid\!\!\sim \neg\alpha$, this means that there are a set

of statements that have resulted in a logical inconsistency allowing for the negation of α to be entailed by the knowledge base.
On a high level, our declarative definition of Base Rank takes a set of defeasible and classical implications as facts. This means that these predicates are always present in the answer sets. Our Base Rank will arbitrarily assign a numerical rank to the implications. For each candidate solution, an implication will be assigned one rank. For each implication on $rank_i$, we assume the existence of an interpretation that satisfies the antecedent, then derive all possible consequences from $rank_i$ to $rank_N$ where $0 \leq i \leq N$. We find the answer sets by eliminating any derived consequences that have resulted in a logical inconsistency. Section 4.1.1 describes the problem instances of Base Rank, while Section 4.1.2 describes the problem encoding. An example walk through of the declarative encoding of Base Rank is found in Appendix 12.1.

4.1.1 Problem Instance: We describe classical implications ($\alpha \rightarrow \beta$) and defeasible implications ($\alpha \mid\sim \beta$) as facts with the predicates $defeasible/2$ and $classical/2$ (the number indicates the arity of the predicate). Example, the fact $defeasible(\alpha, \beta)$ encodes statements of the form $\alpha \mid\sim \beta$. For example, the knowledge base from Table 1 is encoded in this format in Listing 4.

4.1.2 Problem Encoding: The problem encoding of Base Rank is proceeded via schematic rules. Schematic rules contain variables and allow for the generalisation of logic rules [30].
Our definition of Base Rank in Listing 5 follows the *generate, define and test* methodology.

```
1 defeasible(fruits,sweet).
2 defeasible(fruits,seeds).
3 defeasible(lemons,-sweet).
4 defeasible(lemons,fruits).
5 classical(lemons,fruit).
6 #const number_of_statements=4.
```

Listing 4: Problem instance of knowledge base $\mathcal{K}_1$

The preliminary rules on lines 1 and 2 convert all predicates from the problem instance, described in Section 4.1.1, to material implications. The *generate* part on line 4 arbitrarily assigns a numerical to a material implication. The predicate $rank/2$ expresses a numeral rank position assigned to an implication. The constant, $number\_of\_statements$ is an external fact set by the problem instance. We use this constant to indicate the number of possible ranks in the Base Rank model. Each statement will be assigned one rank per candidate solution. The *define* section on lines 5 and 6 derives all the possible consequences of antecedents within the knowledge base. Derving consequences for antecedents assumes that the antecedent is true, and by utilising transitivity rules [32] for implications, all possible consequences that should follow from a true antecedent are derived. On line 5, we derive the antecedent of a m_implication on a rank. Line 6 derives all possible consequences of an antecedent on its rank, N, and ranks above it, N1. We use the constraint $N >= N1$ to ensure that ranks below are not used to derive an antecedent's possible consequences. This is in line with the structure of Base Rank. The *test* part on line 7 is an integrity constraint which removes all answer sets where an antecedent is true but contains a contradictory consequence. The final step, on line 8, uses a *#minimize* directive to ensure the implications are assigned the lowest rank possible while preserving the distribution structure of implications.

```
1 m_implication(X,Y) :- defeasible(X,Y).
2 m_implication(X,Y) :- classical(X,Y).
3 rank(m_implication(X,Y),inf) :- classical(X,Y).
4 {rank(m_implication(X,Y),0..number_of_statements)}
      = 1:-m_implication(X,Y), not classical(X,Y).
5 derive(X,X,N) :- rank(m_implication(X,Y),N).
6 derive(X,P,N) :- rank(m_implication(Y,P),N1),
     derive(X,Y,N), N1 >= N.
7 :- derive(X,Y,N),derive(X,-Y,N).
8 #minimize{N,X,Y:rank(m_implication(X,Y),N)}.
```

Listing 5: Problem encoding of Base Rank

4.2 Rational Closure

The definition of RC in Section 4.2.1 and 4.2.2 consists of two parts: problem instance and problem encoding. The problem instance consists of facts which describe the implication that will be queried against a knowledge base and the ranked statements computed from Base Rank. The rank statements will be in the format shown in Listing 4.

4.2.1 Problem Instance: We define a query by the predicate $query/2$, while the ranked levels of the statements obtained from computing Base Rank are described via the fact predicate $rank/2$. Our definition of RC can handle queries of the form $p \mid\sim q$ and p where p, q are *propositional atoms*. When a query is a propositional atom with no defeasible conditional, it is encoded as $query(p, none)$. An example is found in Listing 6.

```
1 rank(m_implication(lemons,fruit),inf).
2 rank(m_implication(fruits,sweet),0).
3 rank(m_implication(fruits,seeds),0).
4 rank(m_implication(lemons,fruits),1).
5 rank(m_implication(lemons,-sweet),1).
6 query(lemons,sweet).
```

Listing 6: Problem instance of Rational Closure

4.2.2 Problem Encoding: The encoding of Rational Closure is defined in Listing 7. Lines 1, 2 and 3 are auxiliary predicates that extract the antecedent from all the ranked implications and turn our query fact into an $implication/2$. The *generate* method on line 5 arbitrarily assigns a rank to our query, while the $derived/3$ predicate on lines 6 and 7 follows the same logic as outlined in Base Rank. The intuition behind our test method determines the rank for which the antecedent of our query is exceptional, which is captured in the $exceptional/1$ predicate. The $exceptional/1$ predicate indicates that the antecedent is exceptional from $rank_0..rank_N$; thus, we should not consider these ranks when deriving consequences. All consequences are derived from lines 10 and 11, assuming the antecedent is true. We determine entailment in lines 12 to 14 by comparing the antecedent of our query and the consequences derived.

```
antecedent(X) :- rank(m_implication(X,Y),N).
rank(implication(X,Y),N) :- rank(m_implication(X,Y
    ),N).
implication(X,Y) :- query(X,Y).
{rank(implication(X,Y),0..number_of_statements)} =
     1 :- implication(X,Y).
derived(X,X,N) :- rank(implication(X,Y),N).
derived(X,P,N) :- rank(implication(Y,P),N1),
    derived(X,Y,N), N1 >= N.
exceptional(N):- derived(X,Y,N),derived(X,-Y,N).
consequence(X,Y,N) :- query(X,_), rank(implication
    (X,Y),N), rank(implication(X,Y),N1), N1>=N,
    not exceptional(N).
consequence(X,P,N) :- query(X,_), rank(implication
    (Y,P),N), rank(implication(X,Y),N1), N1>=N,
    not exceptional(N).
entailed(true) :- consequence(X1,_,N), query(X2,
    none), X1 = X2, antecedent(X2).
entailed(true) :- consequence(X1,Y1,N), query(X1,
    Y1), antecedent(X1).
entailed(false) :- not entailed(true).
#show entailed/1.
#show query/2.
```

Listing 7: Problem encoding of Rational Closure

5 Knowledge generator

The following sections outline our ASP implementation of a parameterised knowledge base. In section we present the underlying structure that results in multiple ranks in a knowledge base.

5.1 Underlying Structure of knowledge bases

A knowledge base that produces two defeasible ranks can be effectively represented with three straightforward implications. Knowledge base $\mathcal{K}_2$ illustrates this.

$$\mathcal{K}_2 = \{r \to p,\ r \mid\sim \neg y, p \mid\sim y\}$$

Table 3: Base Rank of $\mathcal{K}_2$

∞	$r \to p$
1	$r \mid\sim \neg y$
0	$p \mid\sim y$

The Base Rank of $\mathcal{K}_2$ is represented by Table 3. The implication $r \to p$ results in an unsatisfiable knowledge base $\mathcal{K}_2$ as there does not exist $u \in \mathcal{U}$ where $u \Vdash r \wedge y$ and $u \Vdash r \wedge \neg y$. We say that the literals r and p are in a clash as their relationship in the knowledge base has resulted in the knowledge base $\mathcal{K}_2$ being unsatisfiable.

We can define further implications between clashing atoms to introduce additional ranks in a defeasible knowledge base. For instance, a knowledge base structured with three ranks would take the following form:

$$\mathcal{K}_3 = \{r \to p,\ r \mid\sim \neg y, p \mid\sim y\, p \to q, q \mid\sim \neg t, p \mid\sim t\}$$

$\mathcal{K}_3$ results in three defeasible ranks containing six statements.

5.2 Paramterised knowledge base

In Section 5.1 the underlying conflict structure among antecedents was outlined. This section will look at how we can transform the conflict structure into ASP predicates to generate parameterised knowledge bases. We will be looking at the following parameters:

(1) Number of statements
(2) Number of ranks
(3) Classical statements included
(4) Distribution of statements among ranks
(5) Encoded statements

The problem will consist of two parts: the problem instance and the problem encoding. A subsection will describe how each parameter is generated in our knowledge base generator.

5.2.1 Problem Instance: We denote the predicate $clash/2$ to indicate the two literals in a clash with one another. The predicate $clashed\_atom/1$ acts as an auxiliary predicate which denotes the atoms involved in a clash. The predicate $atom/1$ will denote the literals present in a given problem instance. To derive a logical inconsistency from our clash literals, we define the predicate $inconsistency\_literal/1$. This will represent the literal that will form defeasible relations with the clash literals to form a contradiction. For example, consider the following problem instance for $\mathcal{K}_2$ from Section 5.1 outlined in Listing 8. The clash implication $r \to p$ is encoded in $clash(r, p)$.

```
clashed_atom(p).
clashed_atom(r).
clash(r,p).
atom(r).
atom(p).
inconsistency_literal(y).
classical_included.
```

Listing 8: Problem instance of knowledge base $\mathcal{K}_2$

The $inconsistency\_literal(y)$ will be used to generate the contradiction of $r \mid\sim \neg y$ and $p \mid\sim y$. The $inconsistency\_literal(y)$ is not included in the predicate $atom/1$. This was done to differentiate between atoms used to generate statements and atoms used to create additional ranks.

5.2.2 Problem Encoding: The problem encoding generates knowledge bases given the facts outlined in the problem instance in Section 5.2.1. The methodology followed in the problem encoding is *generate, define and test.*

5.2.2.1 Number of ranks: Listing 9 outlines the rules for generating a specified number of ranks in a knowledge base. The *generate* rule on line 1 adds one $defeasible/2$ predicate to the answer set for every $clashed\_atom/2$. This defeasible implication is of the form $a \mid\sim y$ where a is a $clashed\_atom$, and y is an $inconsistency\_literal$. The *define* rules on lines 2 and 3 encode $defeasible/2$ and $classical/2$

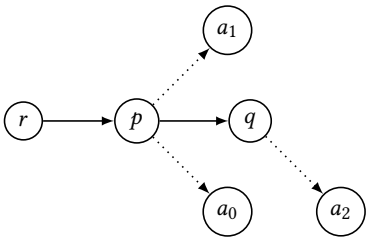

Figure 2: Graph illustration of enumerating numerous statements within a knowledge base.

statements into the answer set. The predicate *classical_included* acts as a guard condition added to our problem instance. This predicate indicates to the solver whether classical statements should be included in the answer sets or if all statements should be defeasible implications. Line 4 ensures that literals in a *clash* do not have the same *inconsistency_literal*.

```
1 {defeasible(X,L;X,-L):inconsistency_literal(L)}
      =1:-clashed_atom(X).
2 defeasible(X,Y):-clash(X,Y), not
      classical_included.
3 classical(X,Y):-clash(X,Y), classical_included.
4 :-clash(X,Y), defeasible(X,L1), defeasible(Y,L2),
      L1=L2.
```

Listing 9: Problem encoding of a ranked knowledge base with problem instances shown in Listing 8

5.2.2.2 Number of statements: Generating many defeasible statements within a knowledge base can take a variety of structures. The encoding followed in our knowledge base uses all *clashed_atoms*/1 as the antecedents. Defeasible implications are generated arbitrarily by assigning random *atom*/1 to *clashed_atoms*/1. The graph in Figure 2 encapsulates this idea. The dotted lines in the graph in Figure 2 indicate defeasible implications, while the solid lines indicate the atoms in a clash. The knowledge base, $\mathcal{K}_4$, corresponding to graph 2 is outlined below.

$$\mathcal{K}_4 = \{r \to p, p \mathrel{|\!\sim} y, r \mathrel{|\!\sim} \neg y,\ p \mathrel{|\!\sim} a_1,\ p \mathrel{|\!\sim} a_0, p \to q, q \mathrel{|\!\sim} \neg y, q \mathrel{|\!\sim} a_2\}$$

$\mathcal{K}_4$ has a total of 8 statements with three defeasible ranks.

```
1 {defeasible(X,Y):atom(Y),not clashed_atom(Y)}:-
      clashed_atom(X).
2 m_implication(X,Y):-defeasible(X,Y).
3 m_implication(X,Y):-classical(X,Y).
4 :-#count{(X,Y):m_implication(X,Y)} !=
      number_of_statements.
```

Listing 10: Problem encoding of a ranked knowledge base that generates additional statements.

To encode the generation of multiple statements in ASP, we utilise aggregate functions available in the clasp solver. Aggregate functions allow us to summarise or aggregate literals in our answer sets. Listing 10 outlines our encoding. Line 1, *the generate method* produces an arbitrary set of defeasible statements for every *clashed_atom* in the answer sets. Lines 2 and 3 imply that all *defeasible*/2 and *classical*/2 predicates are *m_implication*/2. The *test* rule on lines 4 and 5 ensures that the count of implications is within the specified number of statements.

5.2.2.3 Statement distributions: The distribution determines the structure of defeasible implications within the Base Rank of a knowledge base. Our knowledge base generator accommodates the following distributions:

(1) **Uniform Distribution:** This distribution evenly divides the total defeasible statements among the ranks. The remaining defeasible statements will be distributed to the last rank.
(2) **Linear Growth Distribution:**This distribution ensures that the number of statements distributed on each rank increases linearly, starting from the lowest and moving up.
(3) **Random Distribution:** This distribution will assign a random number of defeasible statements to each rank.

Each distribution requires calculating how many statements should be in each rank. The needed calculations proved to be a task that ASP was not well equipped to perform. To address this, script embeddings offered by the clingo were used [22].

5.2.2.4 Encoded Statements: When present in a knowledge base, certain defeasible statements are classical statements. For example, if given $a \mathrel{|\!\sim} b$ and $b \mathrel{|\!\sim} a$, this is $a \to b$ [21]. Identifying encoded statements when computing Base Rank may decrease computational time as subsets of statements are placed on the infinite rank and eliminate the need to compute entailment on the removed statements. Our knowledge generator allows for the generation of how many encoded should be in a knowledge base. Encoded statements generated are of the form shown in Listing 12.

```
1 {encoded(X,Y)}:-atom(X), atom(Y), not clash(X,Y),
      clashed_atom(X), not clashed_atom(Y),
      amount_of_ranks -1 > 0, encoded.
2 encoded(X,Y) :- encoded(Y,X).
3 :- #count\{Y : encoded(X,Y)\} >
      number_of_encoded_statements, encoded.
4 :- #count{Y:encoded(X,Y)} < 1, encoded.
5 defeasible(X,Y) :- encoded(X,Y).
```

Listing 11: Problem encoding generated encoded statements

Listing 11 outlined how encoded statements are generated in the knowledge base. Line 1 of listing 11 generates *encoded*/2 predicates with *clashed_atoms*/2 as the antecedent. The *define* predicate on line 2 swops the literals around to form the structure of encoded statements. Line 3 ensures that the number of encoded statements is within the specified count. Encoded statements are not included in the number of statement counts but are classified as defeasible statements. The predicates in Listing 12 describe defeasible implications of the form $a \mathrel{|\!\sim} b$ and $b \mathrel{|\!\sim} a$ which is $a \to b$.

```
1 defeasible(a,b).
2 defeasible(b,a).
```

Listing 12: Encoded statements in a knowledge base

6 System Design and implementation

In this section, we outline the features of our knowledge generator.

6.1 Generator Features

The system was designed using the ASP paradigm. Clingo, developed by Potassco [1], internally couples gringo and clasp; thus, it takes care of both grounding and solving, making it a suitable option to compute our generator. Clingo was the ASP tool used to develop both the declarative RC definition outlined in section 4 and the knowledge generator outlined in section 5.1. In the sections below, each feature of our generator is outlined.

6.1.1 Number of ranks: The number of ranks specifies how many defeasible ranks the user wants in the generated knowledge base.

6.1.2 Distribution type: The distribution type indicates how the statements should be distributed across the rank.

6.1.3 Number of statements: The number of statements specifies the number of defeasible statements within the knowledge base. This number can act as a minimal number of statements to maintain the structure of the knowledge base for the specified distribution and number of ranks.

6.1.4 Classical included: Classical included will indicate to the generator that classical and defeasible statements should be included in the knowledge base.

6.1.5 Encoded statements: This feature generates knowledge bases with encoded statements. It specifies the number of encoded statements to be generated in the knowledge base.

6.2 Architecture

The generator developed for this project is divided into three files. One generates the problem instances with the specified parameters, which include constants like the number of statements and ranks to be generated. The second file is the problem encoding, which generates the parameterised knowledge base. The third file contains the auxiliary functions used to determine the distribution specifications for the statements in the knowledge base. This is written using the programming language Lua. The knowledge generator can be used in two ways: through a Python application developed to take input from a user and run all three ASP files or the clingo environment. More details can be found on our GitHub [1].

7 Methods and Materials

Experiments were done on the computational generational time of our knowledge generator. We looked at the generation time of knowledge bases with varied numbers of ranks, statements and distributions. We also explored fine-tuning configurations offered by clingo. Efficiency testing was not performed for Base Rank and RC. Refer to [29] for efficiency testing of Base Rank and RC.
Tests were done on the generational time of our knowledge generator on a MacBook Air with an M1 processor, eight-core CPU and seven-core GPU. All experiments were done through the command line with bash scripting. The timing was captured from the statics produced by the clasp solver once a model was found; see Appendix 12.3 for an example of the output. We ran our experiments five times and took the average of the times to account for the run-to-run variations.

7.1 Correctness Testing

Knowledge bases generated were ranked using the Base Rank algorithm outlined in section 4.1.1. The output obtained from Base Rank was compared to that produced by the generator to assess its accuracy. Base Rank and RC were tested against other implementations done by past honours students, namely Jaron Cohen [10].

7.2 Experiment Design and Execution

The experiments tested the time it took for our system to generate a knowledge base with specified parameters. The parameters tested were the number of statements, number of ranks, and varied distributions. Only defeasible statements were generated. The distributions tested were uniform and random. The knowledge bases generated ranged from 100 to 350 statements, incremented by 50, with each statement count consisting of 1 to 20 ranks.
The clasp solver has several predefined configurations which have been fine-tuned to decrease the search time based on characteristics in the search space. Generational time refers to the time the solver takes to find an answer set for our knowledge generator. We looked at four configurations when experimenting with the generational time of our knowledge generator, namely Tweety, Jumpy, Handy and Trendy [1]. The configurations explored make adjustments to the heuristics and progress saving [1] of the solver. Progress saving indicates when the solver caches truth values when performing back jumps and reassigns known truth values to literals. [30].

8 Results and Analysis

8.1 Results

Graphical representations of the results can be found in Appendix 12.2. Figures 3, 4 and 5 summarise the data captured from the experiments for knowledge bases with 300 and 350 with varied ranks of 1 to 20. The experiments conducted varied the distributions, number of statements, and number of ranks.

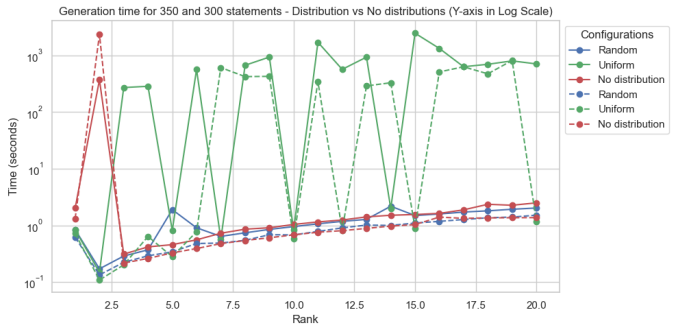

Figure 3: Generation time for 350 and 300 statements with distribution specified and varied ranks.

[1] https://github.com/RacquelNinaDennison/honors-project-

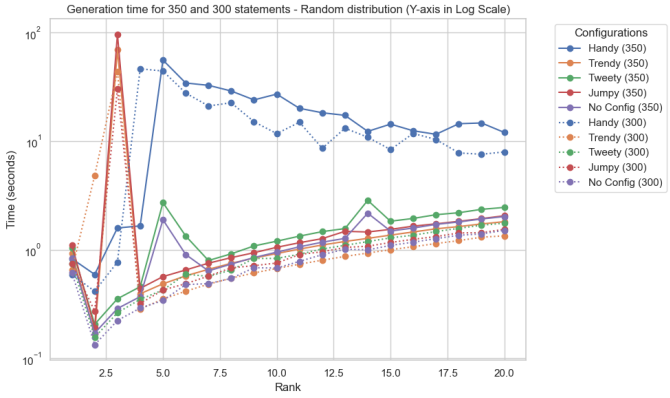

Figure 4: Generation time for 350 and 300 statements with distribution random, varied ranks and fine-tuning configurations specified.

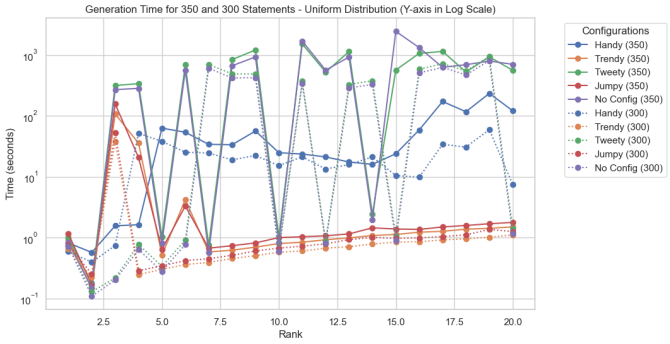

Figure 5: Generation time for 350 and 300 statements with distribution uniform, varied ranks and fine-tuning configurations specified.

8.2 Discussions

8.2.1 Distribution specified: The hypothesis outlined in Section 1 assumed that specifying different parameters, such as the number of statements and ranks for knowledge bases, leads to a slower generation time as the solver is required to ensure that a larger set of rules is satisfied. Figure 3 shows that looking for a uniform distribution among ranks leads to a slower generation time; however, when the amount of ranks evenly divides the number of statements, a uniform distribution performs just as well as when no distribution is specified. We assume this behaviour occurs because, at each rank, the distribution is calculated again based on the rank index when the number of ranks does not evenly divide the number of statements. This leads to more candidate solutions that the solver must explore when finding answer sets.
Random distribution does not affect the solver's generational time and, in some instances, performed better than specifying no distribution. When no distribution is defined, the generation time rises gradually as the ranks increase. The same goes for a random distribution. The behaviour of uniform distribution is erratic; we see no pattern when the number of ranks and the number of statements increase.

8.2.2 Fine-tuning configurations: As mentioned in Section 7.2, four predefined fine-tuning configurations were investigated. We analysed these configurations by looking at how their use affected the generation time of knowledge bases. In Figures 5 and 4, we see that using certain configurations leads to the solver performing worse than using no configuration. In Figure 4, Handy performed the worst as the ranks increased, while Jumpy and Trendy improved generation time as the ranks increased. In Figure 5, Tweety performed on par with No Config leading to little to no improvements in the generation time. Trendy and Jumpy performed better for higher ranks. When specifying a random distribution, in some cases, we notice that the default configurations performed better than using one of the four fine-tuned configurations.
From conducting the experiments, we examined that each configuration is very specific to a problem. We assumed that Handy would have performed the best in reducing the generation time as it is geared towards large problems, and generating knowledge bases was seen as a large problem. However, this proved not to be the case. Trendy and Jumpy performed better with higher rank levels and a uniform distribution. Tweety offered no optimisations to the generation time. In the case of a random distribution, on average, the default solver configurations were sufficient. The biggest takeaway was that the default solver is sufficient for generating knowledge bases.

8.2.3 Base Rank and Rational Closure algorithms: The efficiency of our Base Rank and RC implementation was not tested in this paper; however, a few points are worth nothing. Analysing the declarative algorithms of Base Rank and RC in Section 4, we see that the search space for our Base Rank implementation is m^n where m is the $number\_of\_statements$ while n is the $number\_of\_ranks$. The search space grows exponentially, resulting in a larger runtime to compute Base Rank. Furthermore, the optimisation statement to minimise the models adds to the search time as the solver attempts to find the optimal model. While the implementation is novel, reductions to the search space can be explored in both Base Rank and RC instances. A potential avenue would be to replace the minimisation statement with a constraint that ensures the statements that are not exceptional to a subset of statements are assigned the lowest ranks possible.

9 Discussions

This research project evaluated the utility of ASP for computing algorithms and frameworks defined by the KLM [24] framework, using qualitative data collected throughout the process.

9.0.1 Ease of Use: ASP's grounding and solving workflow simplifies problem-solving. Solvers can be easily installed via a terminal or accessed through a web browser, minimising installation and dependency issues. However, despite its ease of installation, ASP is a complex technology with a steep learning curve, particularly for newcomers. Despite resources such as [30] and [28], extensive background knowledge and a solid mathematical foundation are required when understanding the underlying concepts of stable models in ASP. The increasing complexity of the technology further

complicates the practical implementation of ideas in ASP [22] as it can be overwhelming to grasp all the new concepts. Additionally, adopting a declarative approach can be challenging, particularly for those with experience in imperative programming. Effectively addressing problems declaratively requires a clear understanding of the outlined goals.

Approaching KLM problems declaratively was not straightforward. We encountered several challenges while implementing Base Rank and Rational Closure, resulting in numerous iterations. Initially, our methods used ASP as an SAT solver, with a Python application managing the control flow. This approach constrained our ability to assess the language's usefulness and necessitated significant rework.

9.0.2 Running ASP programs: A notable advantage of ASP is its program compactness. For instance, as demonstrated by the graph colouring example in Listing 1 and 2, a problem that might require numerous lines of code in other languages can be expressed with just a few rules in ASP. This compactness reduces program dependencies and storage requirements, making ASP an efficient choice when computing programs that may have a densely specified imperative counterpart.

Using Clingo requires a simple command-line installation and a text editor, unlike other languages, such as Java, which requires a JVM and other installation setups. This is an appealing feature of Clingo, as you can immediately utilise the solver and the grounder out of the box. Furthermore, the programs are compact, reducing dependencies among programs that could lead to potential bugs.

9.0.3 Configurations Offered by Clasp: Clasp, the solver used in this study, offers a range of configurations and heuristics that enhance program efficiency and reduce runtime. When computing entailment relations, these can be explored to improve the efficiency of the declarative definitions. Clingo also provides built-in statistics for program analysis, eliminating the need for external analytical tools.

9.0.4 Distribution Calculations: When defining multiple distributions for our knowledge generator, we found that the ASP tool was not well-suited for intensive calculations, such as determining the modulo of two numbers, without extensive logic definitions. While it was possible to define operations like modulo in ASP, using external functions with Lua scripting proved to be a more efficient approach to solving this problem.

While ASP presents a learning curve, particularly for those accustomed to imperative programming, it offers a unique perspective on problem-solving. Embracing ASP within the KLM framework can provide new insights and require researchers in the field to approach problems from a different angle, deepening one's understanding of the framework. Taking a more abstract approach to the KLM framework could provide alternative perspectives for optimising the entailment relations defined in the framework.

10 Future Work

Some primary areas of interest for future work extending from the work done in this paper include extending the knowledge generator to accommodate more complex implications such as $\alpha \wedge \beta \mid\sim \gamma$. Configurations to our knowledge generator provided some insights into which settings lead to faster generation times; however, we did not investigate why this occurred. It can be worth investigating why some configurations performed better than others concerning this problem. This may offer insight into adjustments to the solver heuristics that may be used in other aspects of the KLM framework. One well-known pattern of defeasible reasoning, RC, was explored in this paper. However, many other forms of defeasible entailment outlined by the KLM framework can be investigated using ASP. These include Lexicographical Closure [25] and Relevant Closure [7].

11 Conclusions

This paper explored the usefulness of ASP when computing reasoning entailment relations within the KLM framework. To do this, we defined an ASP approach to computing RC and outlined a parameterised knowledge base generator implemented using ASP to generate test sets for RC. We tested the generational time of our knowledge base, evaluating which parameters impacted the generation time. Furthermore, we leveraged the fine-tuning configurations offered by the ASP solver when running our experiments. This aimed to evaluate which configurations lead to an increase or reduction in time.

The implementation of RC in ASP provided valuable insights into the language's applicability to the KLM framework. One key advantage of ASP is its declarative nature, which encourages a deep understanding of the problem being encoded. This abstraction process offered a new perspective on understanding RC.

The experiments conducted on our knowledge base allowed us to explore the fine-tuning configurations offered by clasp. We determined that for our specific problem, Tweety, Jumpy and Handy showed interesting characteristics when changing the distribution to uniform and random. Handy performed worse on average than the default solver configurations, while Jumpy and Tweety reduced generation time for a higher number of ranks and a uniform distribution specified. The takeaway from these experiments was that the default solver configurations for clasp were sufficient to generate knowledge bases.

The work done in this paper gave a fresh perspective on tackling KLM entailment relations. ASP offers several significant advantages in computing KLM reasoning patterns, such as compact programs and abstractions of RC, that better allow us to understand the overarching aim of the problem.

As a programming language, the compactness of ASP allows for more straightforward and efficient programs than imperative languages. The clasp solver gives users more control over how the solver should find stable models. Additionally, the ease of installation and the ability to run programs with minimal dependencies make ASP a practical choice when one wants to encode a problem easily.

ASP comes with a steep learning curve but does enhance one's perspective when looking at a problem. Effective use of the language, especially for complex reasoning frameworks like KLM, requires a solid understanding of the programming paradigm and a deep knowledge of the problem domain. This provides an advantage as it highlights a different perspective on the problems the KLM framework aims to address.

References

[1] 2019. *Potassco User Guide,2nd ed. University of Potsdam.* http://potassco.org

[2] Ga rdenfors P. Makinson D Alchourr on, C.E. 1985. On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic 50(2), 510–530* (1985).

[3] Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub. 2005. A Glimpse of Answer Set Programming. *Künstliche Intell.* 19 (2005), 12–. https://api.semanticscholar.org/CorpusID:185940

[4] Mordechai Ben-Ari. 2012. *Mathematical logic for computer science* (3 ed.). Springer, London. 4–47 pages. https://doi.org/10.1007/978-1-4471-4129-7

[5] Gerhard Brewka, Ilkka Niemelä, and Mirosław Truszczyński. 2008. Chapter 6 Nonmonotonic Reasoning. In *Handbook of Knowledge Representation*, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter (Eds.). Foundations of Artificial Intelligence, Vol. 3. Elsevier, 239–284. https://doi.org/10.1016/S1574-6526(07)03006-4

[6] Sibusiso Buthelezi. 2024. *Defeasible Conditionals in Answer Set Programming.* BSc (Hons) Project. University of Cape Town, Rondebosch, Cape Town.

[7] Giovanni Casini, Thomas Meyer, Kodylan Moodley, and Riku Nortjé. 2014. Relevant Closure: A New Form of Defeasible Reasoning for Description Logics. In *Logics in Artificial Intelligence*, Eduardo Fermé and João Leite (Eds.). Springer International Publishing, Cham, 92–106.

[8] Meyer T. Casini, G. and I. 2018 Varzinczak. 27-29 October 2018. Defeasible Entailment: from Rational Closure to Lexicographic Closure and Beyond. In *17th International Workshop on Non-Monotonic Reasoning (NMR) 2018.* Arizona, USA, pp. 109–118.

[9] Bikas K. Chakrabarti. 2005. *Statistics of Linear Polymers in Disordered Media.* Elsevier Science. https://doi.org/10.1016/B978-0-444-51709-8.X5000-2

[10] Jaron Cohen. 2022. *Model-Based Defeasible Reasoning.* Technical Report. University of Cape Town. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2022/cohen_combrinck.zip/assets/res/MBDR_Final_Paper_CHNJAR003.pdf

[11] Alain Colmerauer. 1990. An introduction to Prolog III. *Communications of the ACM, Volume 33, Issue 7* (1990), 69 – 90. https://doi.org/10.1145/79204.7921

[12] Menachem Magidor Daniel Lehmann. May 1992. What does a conditional knowledge base entail? *Journal of Artificial Intelligence* Vol. 55 no.1 (May 1992), 1–60.

[13] Alan K. Mackworth David L. Poole. 2023. *Artificial Intelligence: Foundations of Computational Agents, 3rd Edition.* Cambridge University Press.

[14] John Woods Dov M. Gabbay. (January 2, 2014). *Handbook of the History of Logic.* North Holland.

[15] Kenneth G. Ferguson. 2003. Monotonicity in Practical Reasoning. *Monotonicity in Practical Reasoning. Argumentation 17, 335–346* (2003). https://doi.org/10.1023/A:1025164703468

[16] G. Governatori M. J. Maher A.Rock G. Antoniou, D. Billington. 2000. A family of defeasible reasoning logics and its implementation. In *Proceedings of the 14th European Conference on Artificial Intelligence.* 459–463.

[17] Kaufmann B. Neumann A. Schaub T. Gebser, M. 2007. clasp: A Conflict-Driven Answer Set Solver. *Baral, C., Brewka, G., Schlipf, J. (eds) Logic Programming and Nonmonotonic Reasoning. LPNMR 2007. Lecture Notes in Computer Science(), vol 4483. Springer, Berlin, Heidelberg.* (2007).

[18] Schaub T. Thiele S. Gebser, M. 2007. GrinGo: A New Grounder for Answer Set Programming. *In: Baral, C., Brewka, G., Schlipf, J. (eds) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science(), vol 4483. Springer, Berlin, Heidelberg.* (2007).

[19] Pearl J Geffner, H. 1992. Conditional entailment: Bridging two approaches to default reasoning. *Artificial Intelligence 53(2-3), 209–244* (1992).

[20] Frank Harmelen, Vladimir Lifschitz, and Bruce Porter. 2007. The Handbook of Knowledge Representation. *Elsevier Science San Diego, USA* (01 2007), 1034.

[21] Adam Kaliski. 2020. *An Overview of KLM-Style Defeasible Entailment. Master's thesis.* Ph.D. Dissertation. Faculty of Science, University of Cape Town, Rondebosch, Cape Town, 7700.

[22] ROLAND KAMINSKI, JAVIER ROMERO, Torsten Schaub, and Philipp Wanko. 2021. How to Build Your Own ASP-based System?! *Theory and Practice of Logic Programming* 23 (12 2021), 1–63. https://doi.org/10.1017/S1471068421000508

[23] Timotheus Kampik. 2022. *Principle-based non-monotonic reasoning - from humans to machines.* Ph.D. Dissertation. Umea University, Sweden. https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-193460

[24] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic Reasoning, Preferential Models and Cumulative Logics. *Journal of Artificial Intelligence* 44 (1990), 167–207.

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

[26] V. Lifschitz. 2002. Answer set programming and plan generation. *Artificial Intelligence, Vol. 138, pp. 39-54* (2002). https://doi.org/10.1016/S0004-3702(02)00186-8

[27] Vladimir Lifschitz. 2008. Twelve Definitions of a Stable Model. *Logic Programming. ICLP 2008. Lecture Notes in Computer Science, vol 5366. Springer, Berlin, Heidelberg* (2008), 37–51.

[28] Vladimir Lifschitz. 2019. *Answer Set Programming.* Springer Cham. https://doi.org/10.1007/978-3-030-24658-7

[29] Jack Mabotja. 2024. *Defeasible Conditionals in Answer Set Programming.* BSc (Hons) Project. University of Cape Town, Rondebosch, Cape Town.

[30] Benjamin Kaufmann Martin Gebser, Roland Kaminski and Torsten Schaub. 2012. Answer Set Solving in Practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning* (2012). doi:10.2200/S00457ED1V01Y201211AIM019

[31] Panos M. Pardalos, Thelma Mavridou, and Jue Xue. 1998. *The Graph Coloring Problem: A Bibliographic Survey.* Springer US, Boston, MA, 1077–1141. https://doi.org/10.1007/978-1-4613-0303-9_16

[32] David Ripley. 2017. On the 'transitivity' of consequence relations. *Journal of Logic and Computation* 28, 2 (12 2017), 433–450. https://doi.org/10.1093/logcom/exx039

[33] Karl Schlechta. 2007. Nonmonotonic Logics: a Preferential Approach. In *The Many Valued and Nonmonotonic Turn in Logic*, Dov M. Gabbay and John Woods (Eds.). Handbook of the History of Logic, Vol. 8. North-Holland, 451–516. https://doi.org/10.1016/S1874-5857(07)80010-0

[34] Peter Norvig Stuart Russell. 2010. *Artifical Intelligence, A Modern Approach.* Pearsons.

12 Appendix

12.1 Illustration of Base Rank in ASP

This section presents a step-by-step example of computing Base Rank as outlined in Section 4.1. Consider the following knowledge base $\mathcal{K} = \{penguins \rightarrow birds, birds \mid\sim fly, penguins \mid\sim \neg fly\}$. This knowledge base will be encoded as outlined in Listing 17.

```
1 defeasible(penguins,-fly).
2 defeasible(birds,fly).
3 classical(penguins,birds).
4 #const amount_of_statements=3.
```

Listing 13: Problem instance of knowledge base $\mathcal{K}$

Lines 1-3 of Listing 5 converts all defeasible and classical statements to material implications. Line 3 will rank all classical statements on rank ∞. Executing the first three lines by calling *clingobase – rank.lp* through the command line results in the output in Listing 14.

```
1 Answer: 1
2 m_implication(penguins,birds) m_implication(birds,
      fly) m_implication(penguins,-fly) rank(
      m_implication(penguins,birds),inf)
3 SATISFIABLE
```

Listing 14: Problem instance of knowledge base $\mathcal{K}$

The *generate* rule on line 4 of Listing 5 generates all possible ways to rank the defeasible statements in the knowledge base. Listing 15 shows the first three models determined by the solver.

```
1  Answer: 1
2  rank(m_implication(penguins,birds),inf)
3  rank(m_implication(birds,fly),1)
4  rank(m_implication(penguins,-fly),0)
5  Answer: 2
6  rank(m_implication(penguins,birds),inf)
7  rank(m_implication(birds,fly),2)
8  rank(m_implication(penguins,-fly),0)
9  Answer: 3
10 rank(m_implication(penguins,birds),inf)
11 rank(m_implication(birds,fly),1)
12 rank(m_implication(penguins,-fly),1)
13 SATISFIABLE
```

Listing 15: Problem instance of knowledge base $\mathcal{K}$

Table 4 shows the ranking of the first model. The *derive*/3 predicate on lines 5 and 6 of Listing 5 will derive all consequences of the antecedent on each rank. For the model shown in Table 4 for $birds \rightarrow fly$ on rank one will derive all consequences from rank 1 to rank ∞, it will not include rank 0. Listing 16 illustrates all the possible derived consequences from the model illustrated in Table 4.

Table 4: BaseRank of knowledge base of first stable model in Listing 15

$\mathcal{R}_\infty$	$penguins \rightarrow birds$
$\mathcal{R}_1$	$birds \rightarrow fly$
$\mathcal{R}_0$	$penguins \rightarrow \neg fly$

```
1  Answer 1:
2  rank(m_implication(penguins,birds),inf) derive(
       penguins,penguins,inf)
3  derive(penguins,birds,inf)
4  derive(birds,birds,1)
5  rank(m_implication(birds,fly),1)
6  derive(penguins,penguins,0)
7  rank(m_implication(penguins,-fly),0)
8  derive(penguins,birds,0)
9  derive(penguins,-fly,0)
10 derive(birds,fly,1)
11 derive(penguins,fly,0)
```

Listing 16: Problem instance of knowledge base $\mathcal{K}$

The *test* predicate on line 7 will eliminate all models where an antecedent, X derives two contradictory consequences, Y and $-Y$. Thus, looking at the model in Listing 16, we see that this is rejected as a model as the program derives $derive(penguins, -fly, 0)$ and $derive(penguins, fly, 0)$.

12.2 Graph Results

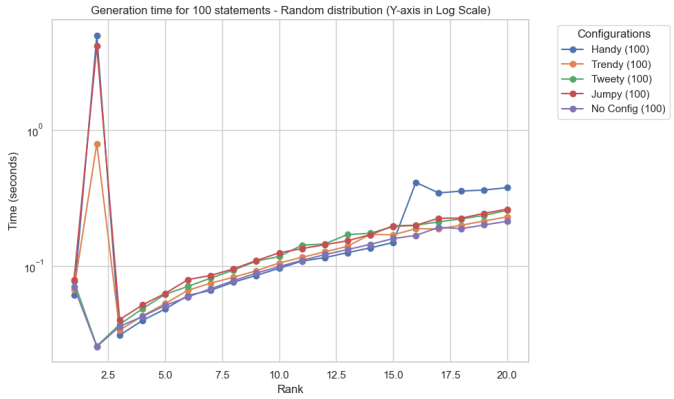

Figure 6: Generation time for 100 statements with distribution random and fine-tuning configurations specified.

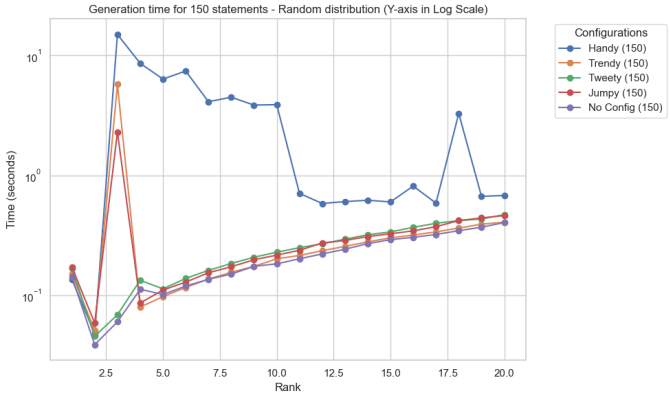

Figure 7: Generation time for 150 statements with distribution random and fine-tuning configurations specified.

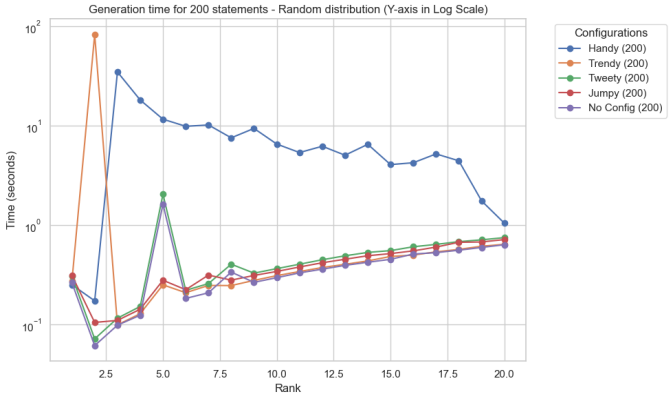

Figure 8: Generation time for 200 statements with distribution random and fine-tuning configurations specified.

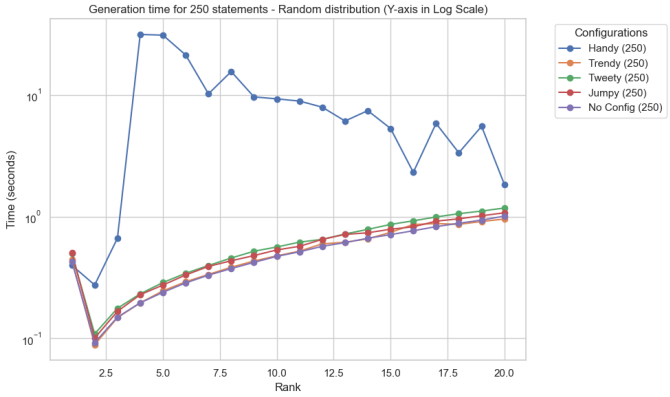

Figure 9: Generation time for 250 statements with distribution random and fine-tuning configurations specified.

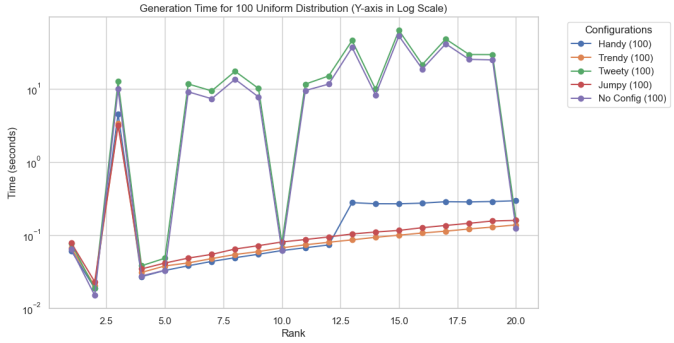

Figure 10: Generation time for 100 statements with distribution uniform and fine-tuning configurations specified.

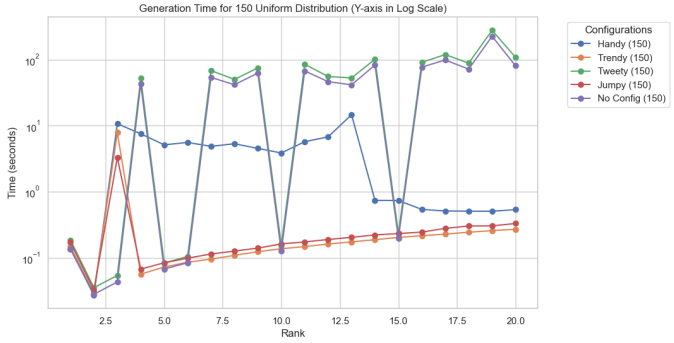

Figure 11: Generation time for 150 statements with distribution uniform and fine-tuning configurations specified.

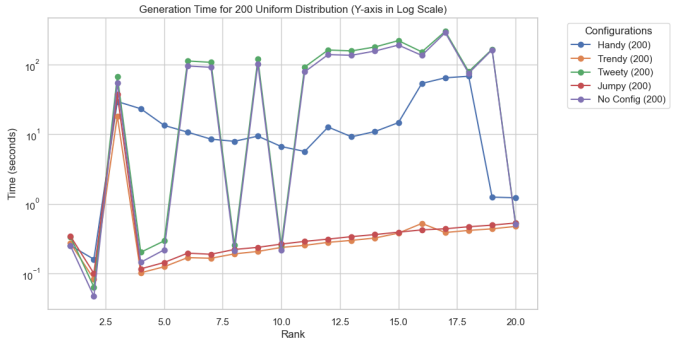

Figure 12: Generation time for 200 statements with distribution uniform and fine-tuning configurations specified.

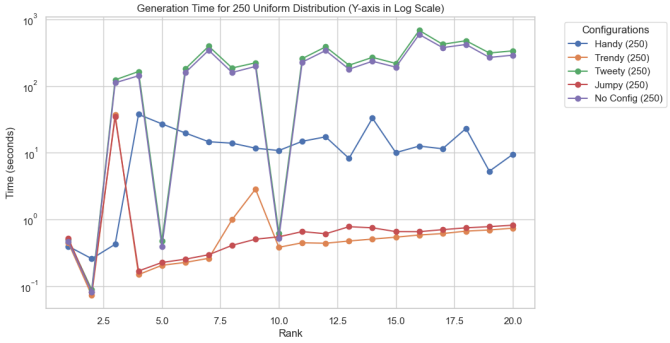

Figure 13: Generation time for 250 statements with distribution uniform and fine-tuning configurations specified.

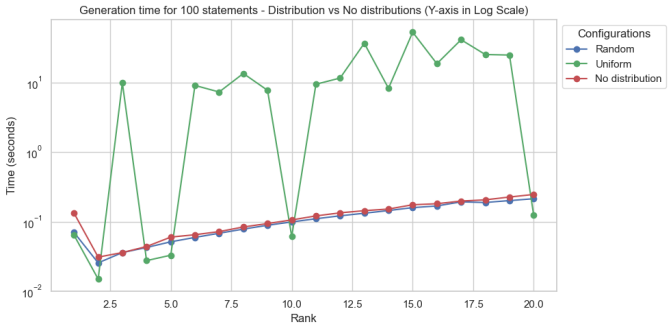

Figure 14: Generation time for 100 statements with distribution specified.

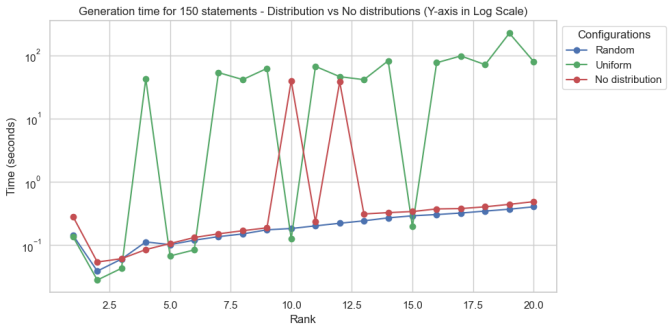

Figure 15: Generation time for 150 statements with distribution specified.

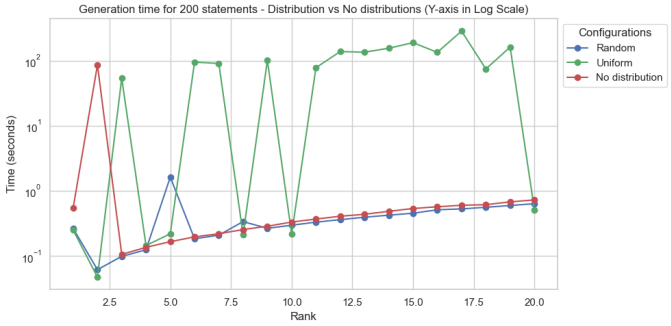

Figure 16: Generation time for 200 statements with distribution specified.

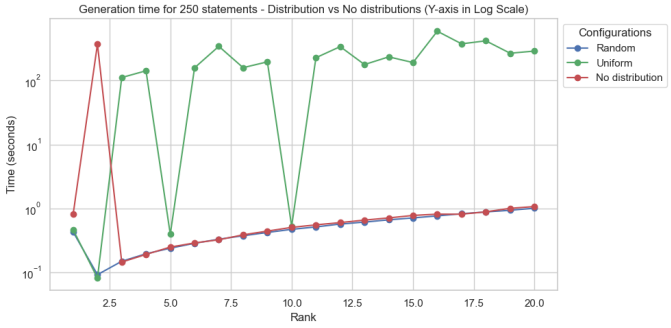

Figure 17: Generation time for 250 statements with distribution specified.

12.3 Output from a generated knowledge base

Below is an example of the output generated from the knowledge base generator. The following command generated a knowledge base with 100 statements, two ranks and a random distribution. Our experiments took the total time on line 24 of Listing 18.

```
clingo --outf=2 --quiet=1 knowledge-base instances.lp -c number_of_ranks=2 -c number_of_statements=10 -c
    random=1 knowledge_base_problem_class_2.lp functions.lp
```

Listing 17: Clingo command to generate a knowledge base of 10 statements, 2 ranks and random distribution


```
{
  "Solver": "clingo version 5.7.1",
  "Input": [
    "knowledge-base-instances.lp","knowledge_base_problem_class_2.lp","functions.lp"
  ],
  "Call": [
    {
      "Witnesses": [
        {
          "Value": [
            "defeasible(a(1),a(0))", "defeasible(a(0),a(2))", "defeasible(a(0),a(3))", "defeasible(a(0),a
                (4))", "defeasible(a(0),a(5))", "defeasible(a(0),a(6))", "defeasible(a(0),a(7))", "
                defeasible(a(1),a(8))", "defeasible(a(0),-l0)", "defeasible(a(1),l0)"
          ]
        }
      ]
    }
  ],
  "Result": "SATISFIABLE",
  "Models": {
    "Number": 1,
    "More": "yes"
  },
  "Calls": 1,
  "Time": {
    "Total": 0.009,
    "Solve": 0.000,
    "Model": 0.000,
    "Unsat": 0.000,
    "CPU": 0.007
  }
}
```


Listing 18: Output from generating knowledge base of 10 statements, 2 ranks and a random distribution