UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS Honours Project Final Paper 2024

Title: Defeasible Conditionals in Answer Set Programming

Author: Jack Mabotja

Project Abbreviation: DCASP

Supervisor(s): Professor Thomas Meyer and Dr Jesse Heyninck

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	20
Experiment Design and Execution	*0*	*20*	10
System Development and Implementation	*0*	*20*	0
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Defeasible Conditionals in Answer Set Programming

Jack Mabotja
University of Cape Town
Cape Town, Western Cape, South Africa
mbtjac003@myuct.ac.za

ABSTRACT

Defeasible logic extends propositional logic by enabling the handling of statements that can be overridden or defeated by more specific information. This capability is crucial for dealing with exceptions and conflicting rules. The KLM framework provides a formal foundation for defeasible reasoning, and within this framework, rational closure offers a structured method to prioritise and reason with defeasible statements. Conclusions drawn under rational closure are the most "cautious" or "rational" possible, given the available knowledge. While rational closure has been predominantly implemented imperatively, declarative implementations remain underexplored.

In this work, we present a declarative implementation of rational closure using Answer Set Programming (ASP). Through multiple approaches, we found that a recursive solution outperforms others in terms of efficiency, particularly compared to the search-based approach commonly advocated in introductory ASP literature. Our results highlight the benefits of leveraging declarative techniques in this context, offering a more efficient alternative to imperative implementations.

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

Knowledge Representation and Reasoning (KRR), a subfield of Artificial Intelligence (AI), explores methods for encoding, storing, and processing information in computer systems to facilitate intelligent problem-solving and reasoning [5]. KRR aims to organise large volumes of domain-specific information—including facts, rules, concepts, and relationships—to enable computers to comprehend and utilise it for decision-making and problem-solving.

The goal of KRR is to develop computer systems with reasoning capabilities that approach human-level cognition. Humans process information using sophisticated mental models and conceptual frameworks. KRR seeks to emulate aspects of these cognitive processes in computer systems.

Two example approaches in KRR are the Logical Representation Scheme and the Network Representation Scheme [17]. The Logical Representation Scheme utilises formal logic to encode knowledge and derive new conclusions through inference rules. Conversely, the Network Representation Scheme models knowledge as an interconnected graph, where nodes represent concepts or entities, and edges depict their relationships. This network-based approach effectively represents complex hierarchical structures and relationships within a knowledge domain.

Our work focuses on the logical representation approach which provides a systematic framework for encoding knowledge using logical symbols and operators. We concentrate on two logical systems: classical propositional logic and defeasible logic. Propositional logic is a formal reasoning system that uses a set of connectives and propositional statements to form a knowledge base [19]. A defining feature of propositional logic is its monotonicity—once conclusions are drawn, they remain valid even when new information is added [19]. However, this feature presents challenges when dealing with incomplete or exceptional information.

Defeasible logic addresses this challenge by allowing reasoning that accommodates exceptional information. The KLM Framework [21], a well-known system for defeasible logic, extends propositional logic, making it possible to handle exceptions. Rational closure is one of the most prominent defeasible entailment relations in the KLM Framework, which we describe in detail in Section 2 [19]. While multiple implementations of rational closure in imperative languages have been attempted, there has been little exploration of alternative implementation approaches. This presents an opportunity to investigate whether a different approach might outperform existing ones.

In this project, we implement rational closure, as defined in the KLM Framework, using a declarative programming paradigm called Answer Set Programming (ASP). The declarative programming approach aims to separate the logic of algorithms from the control mechanisms required to implement these algorithms in imperative languages [24]. This separation is intended to enhance programmer efficiency. We will also explore this potential benefit and whether there are other qualitative benefits to using ASP over an imperative language. We will then conduct a performance analysis to determine if our implementations are more efficient than the existing imperative ones and explore potential optimisation strategies.

This work is part of a larger project in collaboration with Racquel Dennison and Sibusiso Buthelezi. Racquel Dennison's contribution focuses primarily on implementing a knowledge base generator using ASP and assessing potential performance improvements [9]. Sibusiso Buthelezi's work centres around developing an application to explain the reasoning process and the outcome of a given entailment query [7].

The remainder of the paper is organised as follows: Section 2 overviews the foundational knowledge necessary to understand our work. Section 3 briefly discusses related works. Section 4 details the theory behind our implementation, including the encoding and optimisation strategies. Section 5 presents the experimental evaluation of our implementations, followed by a discussion of the qualitative benefits and potential drawbacks of using ASP to implement rational closure in Section 6. Finally, Section 7 summarises our conclusions and suggests directions for future research based on our findings and other areas we could not fully explore.

2 BACKGROUND

2.1 Propositional logic

Propositional logic is a basic formal system used in KRR to represent and reason about information. Its core components are propositional *atoms* [19, 20], which correspond to basic statements such as "The sky is blue" or "The sun is shining" that cannot be divided into simpler parts.

The set of all propositional atoms is denoted by $\mathcal{P}$ [19]. Additionally, there is a set of connectives: $\{\rightarrow, \leftrightarrow, \land, \lor, \neg\}$. By combining these two sets, more complex statements, commonly referred to as *formulas*, can be constructed. The set of all such formulas is typically represented by $\mathcal{L}$ (referred to as the language), with individual formulas usually denoted by lowercase Greek letters such as α and β.

The semantics of propositional logic defines the interpretation of logical expressions, including propositions and formulas. Its goal is to determine the truth or falsity of any logical expression by interpreting the symbols and rules within the system [19].

An *atom* can have a truth value of either true (T) or false (F). When an atom is assigned the value T, it is said to be *satisfied*. The assignment of truth values to atoms is done by a function called a *valuation*. A valuation is a function that maps each propositional atom $p \in \mathcal{P}$ to a truth value $u \in \{T, F\}$. If a valuation u assigns the truth value T to an atom p (i.e., $u(p) = T$), the valuation is said to satisfy the atom, denoted as $u \Vdash p$. A valuation u that satisfies a formula α is called a *model* of α.

Let $\mathcal{U}$ represent the set of all possible valuations for the language $\mathcal{L}$. The set of all models of a formula α is denoted by $[\![\alpha]\!]$ and is defined as $[\![\alpha]\!] := \{u \in \mathcal{U} | u \Vdash \alpha\}$. Essentially, $[\![\alpha]\!]$ is the set of all valuations $u \in \mathcal{U}$ that satisfy α.

A finite set $\mathcal{K} \subseteq \mathcal{L}$ of propositional formulas is referred to as a *knowledge base*. A valuation u satisfies $\mathcal{K}$, denoted $u \Vdash \mathcal{K}$ if and only if it satisfies every formula $\alpha \in \mathcal{K}$. A formula $\alpha \in \mathcal{L}$ is said to be entailed by $\mathcal{K}$ (denoted as $\mathcal{K} \models \alpha$) if and only if for every $u \in \mathcal{U}$ such that $u \Vdash \mathcal{K}$, it is the case that $u \Vdash \alpha$. This means that a knowledge base entails a formula if every valuation that satisfies the knowledge base also satisfies the formula. Another way to express this is $\mathcal{K} \models \alpha$ *iff* $[\![\mathcal{K}]\!] \subseteq [\![\alpha]\!]$, which is the definition of *classical entailment* [19].

Classical entailment is effective for many scenarios, but it has a limitation known as *monotonicity* [19]. Monotonicity states that adding new information to a knowledge base should not change or retract previous conclusions.

For instance, consider a knowledge base $\mathcal{K}$ with statements like $b \rightarrow f$ (birds fly), $p \rightarrow b$ (penguins are birds), $p \rightarrow \neg f$ (penguins don't fly), and $t \rightarrow b$ (Tweety is a bird). From this, we can infer that Tweety can fly. However, if we add $t \rightarrow p$ (Tweety is a penguin), we then infer that Tweety cannot fly. Monotonicity prevents us from retracting the previous conclusion, leading to a contradictory result: that Tweety doesn't exist.

This issue highlights why *non-monotonic* logics are needed. Unlike propositional logic, non-monotonic logics allow for conclusions to be revised or withdrawn when new information is introduced. Defeasible logics are a type of non-monotonic logics that address this challenge. Our work focuses on the KLM-style framework, which is a formal system in non-monotonic reasoning that provides a foundational approach to defeasible reasoning [8]. We introduce this framework in the following subsection.

2.2 KLM Approach to Defeasible Reasoning

Defeasible logics are a specific form of non-monotonic logics designed to handle reasoning with rules that are generally true but can be overridden by specific conditions or exceptions [8]. These logics aim to model real-world reasoning, where conclusions are based on general principles but can be revised with new information or exceptions.

There are various approaches to defeasible reasoning, but the most widely recognised and the one we used is the KLM framework [21]. Developed by Kraus, Lehmann, and Magidor, this framework extends propositional logic to incorporate non-monotonic features. It is noted for its elegance and robustness [25].

The KLM framework introduces a new operator, $\mid\!\sim$, to represent defeasible implications. This operator enables the construction of conditional assertions, which can be challenged by other statements in the knowledge base. A conditional assertion is expressed as $\alpha \mid\!\sim \beta$ where $\alpha, \beta \in \mathcal{L}$, meaning that β defeasibly follows from α.

Several types of defeasible entailment relations are defined within the KLM framework [19]. Our focus is on the Rational Entailment relation, which is recognised as the most intuitive and conservative of these relations [19].

2.2.1 *Rational Closure.*

In the KLM framework, rational closure is employed to perform defeasible reasoning for defeasible knowledge bases. Defeasible entailment is denoted by the symbol $\models\!\approx$ [19]. The process of reasoning under rational closure involves two steps. First, we take a defeasible knowledge base $\mathcal{K}$ and compute its materialisation (denoted $\overrightarrow{\mathcal{K}}$). This materialisation process converts defeasible statements into their classical equivalents [8, 19]. For instance, the materialisation of $\alpha \mid\!\sim \beta$ becomes $\alpha \rightarrow \beta$. The materialisation of a defeasible knowledge base $\mathcal{K}$ is achieved by replacing each defeasible implication $\alpha \mid\!\sim \beta \in \mathcal{K}$ with a propositional implication $\alpha \rightarrow \beta$, resulting in $\overrightarrow{\mathcal{K}}$ [19].

Next, each defeasible statement is assigned a rank. This ranking is based on the concept of *exceptionality*, which indicates the specificity of each statement. Given $\overrightarrow{\mathcal{K}}$, the materialisation of a knowledge base $\mathcal{K}$, a propositional formula α is said to be *exceptional* for $\overrightarrow{\mathcal{K}}$ if and only if $\overrightarrow{\mathcal{K}} \models \neg\alpha$ [19]. In natural language, this says that a formula is exceptional for a materialised knowledge base if the materialised knowledge base entails its negation. The algorithm used for ranking is shown below.

Algorithm 1 `BaseRank`

```
1: Input: A knowledge base $\mathcal{K}$
2: Output: An ordered tuple ($\mathcal{R}_0$,..., $\mathcal{R}_{n-1}$,$\mathcal{R}_\infty$,$n$)
3: $i = 0$;
4: $\mathcal{E}_0 = \overrightarrow{\mathcal{K}}$;
5: while $\mathcal{E}_{i-1} \neq \mathcal{E}_i$ do
6:     $\mathcal{E}_{i+1} = \{ \alpha \rightarrow \beta | \mathcal{E}_i \models \neg\alpha \}$ ;
7:     $\mathcal{R}_i = \mathcal{E}_i \backslash \mathcal{E}_{i+1}$;
8:     $i = i + 1$
9: end while
10: $\mathcal{R}_\infty = \mathcal{E}_{i-1}$;
11: if $\mathcal{E}_{i-1}$ = then
12:     $n = i - 1$;
13: else
14:     $n = i$;
15: end if
16: return ($\mathcal{R}_0$,..., $\mathcal{R}_{n-1}$,$\mathcal{R}_\infty$,$n$);
```

After ranking, we can then perform entailment queries on the final ranked knowledge base. The algorithm used for this is shown below.

Algorithm 2 `RationalClosure`

```
1: Input: A knowledge base $\mathcal{K}$ and a defeasible implication $\alpha | \sim \beta$.
2: Output: true, if the query is entailed by the knowledge base, else false.
3: ($\mathcal{R}_0$,..., $\mathcal{R}_{n-1}$,$\mathcal{R}_\infty$,$n$) = $BaseRank(\mathcal{K})$;
4: $i = 0$;
5: $\mathcal{R} = \bigcup_{i=0}^{j<n} \mathcal{R}_j$
6: while $\mathcal{R}_\infty \cup \mathcal{R} \models \neg\alpha$ and $\mathcal{R} \neq \emptyset$ do
7:     $\mathcal{R} = \mathcal{R} \backslash \mathcal{R}_i$;
8:     $i = i + 1$;
9: end while
10: return $\mathcal{R}_\infty \bigcup \mathcal{R} \models \alpha \rightarrow \beta$;
```

In summary, computing the rational closure of a defeasible knowledge base $\mathcal{K}$ involves the following steps: first, compute the materialisation $\overrightarrow{\mathcal{K}}$; next, assess the exceptionality of all statements in $\overrightarrow{\mathcal{K}}$ and rank them accordingly. This process results in a ranked knowledge base, which can then be used to answer defeasible entailment queries. We will demonstrate the usage of these algorithms with an example.

Example 2.1. Let $\mathcal{K} := \{b \mid\!\sim f, p \mid\!\sim b, p \mid\!\sim \neg f, s \mid\!\sim p, s \mid\!\sim f, d \rightarrow b\}$ and $\alpha := s \mid\!\sim b$.

In order to check if $\mathcal{K} \mathrel{|\!\approx} \alpha$, we start by ranking $\mathcal{K}$. After running `BaseRank` on $\mathcal{K}$, we will end up with the following ranked knowledge base:

$\mathcal{R}_\infty$	$d \rightarrow b$
$\mathcal{R}_2$	$s \rightarrow p, s \rightarrow f$
$\mathcal{R}_1$	$p \rightarrow b, p \rightarrow \neg f$
$\mathcal{R}_0$	$b \rightarrow f$

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

After ranking knowledge base $\mathcal{K}$, we can check if the ranked knowledge base entails $s \rightarrow b$ (the materialisation of the query α). We start this process by removing the lowest ranks until the antecedent of the query is no longer exceptional in the ranked knowledge base.

(1) We check if the ranked knowledge base entails $\neg s$. We find that it does, and we then remove the lowest rank.

$\mathcal{R}_\infty$	$d \rightarrow b$
$\mathcal{R}_2$	$s \rightarrow p, s \rightarrow f$
$\mathcal{R}_1$	$p \rightarrow b, p \rightarrow \neg f$

Figure 2: Base Ranking of Knowledge Base $\mathcal{K}$ after removing $\mathcal{R}_0$

(2) We check again whether the ranked knowledge base entails $\neg s$ and find that it still does. We then remove the lowest rank.

$\mathcal{R}_\infty$	$d \rightarrow b$
$\mathcal{R}_2$	$s \rightarrow p, s \rightarrow f$

Figure 3: Base Ranking of Knowledge Base $\mathcal{K}$ after removing $\mathcal{R}_0$ and $\mathcal{R}_1$

(3) We continue to check if the remaining ranked knowledge base entails $\neg s$. We find that it does not. We then check if it entails α. We find that it does not, allowing us to conclude that $\mathcal{K} \not\mathrel{|\!\approx} \alpha$.

2.3 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm designed for tackling complex search problems, especially those that are NP-hard [2, 23]. It is based on *answer set semantics* [4].

In ASP, problems are formulated as logic programs made up of rules and facts. Rules describe how new facts can be derived from existing ones, while facts are assumed to be true within the context of the program. The objective is to compute *answer sets* (or *stable models*), which are sets of facts that satisfy all the rules and provide potential solutions to the problem [23].

The process of solving problems in ASP requires these two tools: a *grounder* and a *solver*. A *grounder* is a pre-processing tool that converts an input high-level, abstract ASP program containing variables into an equivalent variable-free program [10]. This process is known as grounding and the resulting program is said to be *ground.* A *solver* is a tool that takes a ground program and computes the answer sets of that program [10]. These correspond to the solutions of the input program [1].

There are several grounders and solvers available today, but our work uses the grounder *gringo* [16] and the solver *clasp* [15]. We used the ASP system *clingo* [12], which integrates the functionalities of gringo and clasp. All these tools are part of the Pottasco open source project [14].

3 RELATED WORK

Joel Hamilton implemented both `BaseRank` and `RationalClosure` in [18] using Java and the TweetyProject library [27]. The primary focus was to explore ways to improve the scalability of rational closure in practice, ensuring it remains efficient for larger knowledge bases. To achieve this, a basic version of `RationalClosure` was implemented, followed by three optimisations.

The three optimisations include using binary search to find the lowest rank at which the antecedent of a query stops being exceptional in a ranked knowledge base, using a hashtable to store antecedents of queries and the lowest ranks at which they stop being exceptional and a combination of the two. Evashna Pillay added to these optimisations by using ternary search in `RationalClosure`, and by combining ternary search with multithreading [26] using Java and the TweetyProject as well.

All these efforts showed significant potential to help lead to a more scalable implementation of `RationalClosure`. In both these cases, little effort was put into making `BaseRank` more scalable as well.

4 IMPLEMENTATIONS AND OPTIMISATIONS

We used two approaches to implement `BaseRank` and `RationalClosure` in ASP. The second approach was developed after the first was evaluated and its bottlenecks were identified.

This section describes the theory behind our implementations and provides illustrative examples for both methods. We also provide an overview of how they were encoded in ASP and how the encoding was further optimised. It should be noted that our implementations only focused on statements of the form $\alpha \rightarrow \beta$ and $\alpha \mid\!\sim \beta$, where α, β are atomic formulas.

4.1 Search-Based Approach

The algorithms for `BaseRank` and `RationalClosure` describe the process of "building" the solutions we seek. For example, the `BaseRank` algorithm describes how to rank individual statements to get the final ranked knowledge base. In contrast, introductory ASP literature [6, 11, 22, 23] suggests treating problems such as `BaseRank` and `RationalClosure` as search problems. In this approach, rather than describing how to build a solution, we define the characteristics of the solution we seek and search for it.

We start by enumerating the entire search space and then systematically process all the solution candidates to see if they satisfy the characteristics of the solution we seek.

After analysing multiple `BaseRank` and `RationalClosure` solutions, we identified their common characteristics. We describe these characteristics in the following two subsections and provide illustrative examples.

4.1.1 `BaseRank`. Below, we list the common characteristics of the example solutions we analysed.

(1) All classical statements are in the infinite rank.
(2) Two statements with the same antecedent cannot be in two different ranks.
(3) There are no gaps between ranks.
(4) The antecedent of every statement must not be exceptional in the rank of the statement.
(5) For every statement in some rank i (where i is greater than zero), the statement's antecedent is exceptional in the union of ranks zero up to and including rank i and the infinite rank.
(6) For every statement in some rank i, the statement's antecedent is not exceptional in the union of ranks i up to and including rank n (where n is the highest finite rank) and the infinite rank.
(7) Every statement is assigned the lowest possible rank (which is the lowest rank at which all other characteristics are satisfied).

We use the following example to illustrate the characteristics above.

Example 4.1. Let $\mathcal{K} := \{b \mid\!\sim f, p \mid\!\sim b, p \mid\!\sim \neg f, s \mid\!\sim p, s \mid\!\sim f, d \rightarrow b\}$.

The associated ranking of the formulas is given in figure 1.

In example 4.1, the only classical statement, $d \rightarrow b$, is in the infinite rank. All the statements that share antecedents are in the same rank. The ranked knowledge base does not have any gaps between its ranks. None of the antecedents are exceptional in the ranks of their statements.

To illustrate characteristic (5), we focus on $\mathcal{R}_1$. We start by taking the union of $\mathcal{R}_0$, $\mathcal{R}_1$, and $\mathcal{R}_\infty$. This union forms this set: $\mathcal{S}_1 := \{b \rightarrow f, p \rightarrow b, p \rightarrow \neg f, d \rightarrow b\}$. The only antecedent in $\mathcal{R}_1$ is p, and $\mathcal{S}_1 \models \neg p$. Therefore, characteristic (5) is satisfied.

We again use $\mathcal{R}_1$ to illustrate characteristic (6). We start by taking the union of $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_\infty$. This union forms this set: $\mathcal{S}_2 := \{p \rightarrow b, p \rightarrow \neg f, s \rightarrow p, s \rightarrow f, d \rightarrow b\}$. Since $\mathcal{S}_2 \not\models \neg p$, characteristic (6) is satisfied.

If we take any of the statements in the ranked knowledge base and rank them in a lower rank, then not all of the characteristics (1)-(6) will be satisfied. Therefore, every statement is in the lowest possible rank, and characteristic (7) is satisfied.

We first implemented `BaseRank` using a straightforward encoding and then it was followed by an optimised encoding. In the straightforward encoding, we start by ranking all classical statements in the infinite rank. The rest of the encoding is then divided into three parts: a search space generator, constraints, and an optimisation part.

The search space generator uses a *choice rule* [23] to generate all possible solution candidates. A choice rule is a statement that allows one to choose any subset of a given set of atoms to include in a model [23]. It is often used in ASP as a generator to enumerate a search space. The second part has all the constraints to remove candidates not satisfying the first six characteristics. Each of the first six characteristics has a constraint (and auxiliary rules where necessary) encoding it. Characteristics (5) and (6) are implemented like their illustration in Example 4.1.

After removing all the solution candidates that do not satisfy characteristics (1)-(6), we end up with a collection of solution candidates where some statements are ranked higher than they should be. Some may even have more ranks than the solution we seek. In the third part of the encoding, we encode characteristic (7) with a *minimize directive* [23] to find the solution with all the statements in the lowest possible rank. A minimize directive is an example of an optimisation statement, which is a type of statement used to find the most optimal answer set based on some criterion [23].

When encoding characteristics (5) and (6), we start by building the respective unions. We then add each antecedent to its unique union and derive all the possible atoms. If a given union has a contradiction afterwards, then the antecedent is exceptional in that union. This is because for some knowledge base $\mathcal{K}$ and formula α, $\mathcal{K} \models \neg\alpha$ if and only if $\mathcal{K} \cup \{\alpha\}$ is unsatisfiable (i.e., leads to a contradiction) [19].

The encodings of characteristics (5) and (6) share some rules, resulting in redundancies. We optimised the search-based encoding by combining these encodings and reusing rules where possible. For example, the rules for adding classical statements to the unions and deriving all possible atoms are similar and can be reused. This reduced the total number of variables in our encoding, which can be expected to speed up the grounding process [23]. The solver also does less work due to the combination of some of the rules. This reduces the solving time.

4.1.2 `RationalClosure`.

The `RationalClosure` algorithm can be divided into two parts. In the first part, we eliminate the lowest ranks until the antecedent of the query is no longer exceptional in the ranked knowledge base. In the second part, we check whether the remaining ranks classically entail the query. The first part of the algorithm can be treated like a search problem. We are searching for a subset of the ranked knowledge base that starts at some rank i, extends up to rank n (where n is the last defeasible rank and $i \leq n$), includes the infinite rank, and has the following characteristics:

1. The subset always includes the infinite rank.
2. The subset may or may not include any defeasible ranks.
3. The antecedent of the query must not be exceptional in the subset.
4. The subset starts at the lowest possible rank i such that condition (3) is met.

After finding the correct subset, we move on to the second part, where we check whether this subset entails the materialisation of the query. We add the negation of the materialisation of the query to the remaining subset and check for satisfiability. The final result is "True" if the resulting set is unsatisfiable and "False" otherwise.

The encoding of `RationalClosure` has four parts. The first three parts are for the algorithm's first part and are similar to the ones described for `BaseRank`. They include the search space generator, the constraints to filter solution candidates, and an optimisation part. The fourth part is where we check for entailment. If $\mathcal{S}$ is the final subset and $\alpha \rightarrow \beta$ is the materialisation of the query, then adding α and $\neg\beta$ to $\mathcal{S}$ is equivalent to adding the negation of the materialisation of the query to $\mathcal{S}$. We then check for satisfiability and give the final result based on it.

4.2 Recursive Approach

After encoding our search-based approach for `BaseRank` and `RationalClosure`, we benchmarked our `BaseRank` implementation against the imperative implementation done by Hamilton in [18] to analyse its performance under certain conditions. Section 5 documents these benchmarks and their results.

From these benchmarks, we discovered that the most significant bottlenecks of the `BaseRank` implementation are the size of the search space and the use of the optimisation technique used to find the right solution. Based on these discoveries, we devised a different implementation—one that does not rely on searching the entire search space or using optimisation. Based on the original definition of `BaseRank`, we can see that it is possible to restate it recursively. This forms the basis of our second implementation approach, discussed in detail below.

4.2.1 `BaseRank`.

In the recursive approach to `BaseRank`, we start with a set equivalent to the materialisation of the input knowledge base. We first put all the classical statements in the infinite rank without removing them from the starting set. We then identify all the statements whose antecedents are not exceptional in this set and rank them in $\mathcal{R}_0$. We then remove the statements in $\mathcal{R}_0$ from the set. We continue the ranking process recursively. The new, smaller set serves as our recursive input. We identify all the statements in this new set whose antecedents are not exceptional and rank them in $\mathcal{R}_1$. This process continues until only classical statements remain in the set. At that point, we obtain a ranked knowledge base. We illustrate this new approach with the example below.

Example 4.2. Let $\mathcal{K} := \{b \mid\!\sim f, p \mid\!\sim b, p \mid\!\sim \neg f, s \mid\!\sim p, s \mid\!\sim f, d \rightarrow b\}$.

1. $\mathcal{R}_\infty = \{d \rightarrow b\}$
2. Let $\mathcal{E}_0 := \overrightarrow{\mathcal{K}} = \{b \rightarrow f, p \rightarrow b, p \rightarrow \neg f, s \rightarrow p, s \rightarrow f, d \rightarrow b\}$. Since $\mathcal{E}_0 \not\models \neg b$, b is not exceptional in $\mathcal{E}_0$ and $\mathcal{R}_0 = \{b \rightarrow f\}$.
3. $\mathcal{E}_1 = \mathcal{E}_0 \backslash \mathcal{R}_0 = \{p \rightarrow b, p \rightarrow \neg f, s \rightarrow p, s \rightarrow f, d \rightarrow b\}$. Since $\mathcal{E}_1 \not\models \neg p$, p is not exceptional in $\mathcal{E}_1$ and therefore, $\mathcal{R}_1 = \{p \rightarrow b, p \rightarrow \neg f\}$.
4. $\mathcal{E}_2 = \mathcal{E}_1 \backslash \mathcal{R}_1 = \{s \rightarrow p, s \rightarrow f, d \rightarrow b\}$. Since $\mathcal{E}_2 \not\models \neg s$, s is not exceptional in $\mathcal{E}_2$ and therefore, $\mathcal{R}_2 = \{s \rightarrow p, s \rightarrow f\}$.
5. $\mathcal{E}_3 = \mathcal{E}_2 \backslash \mathcal{R}_2 = \{d \rightarrow b\}$. The set contains only classical statements, so the process terminates.

In our encoding, we begin by placing all classical statements into $\mathcal{R}_\infty$, similar to the search-based approach encoding. The rest of the encoding is then broken into a base case and a recursive part. In the base case, statements with non-exceptional antecedents are placed into $\mathcal{R}_0$, while the remaining statements are moved to a new set. This new set serves as the input for the recursive step.

In the recursive step, we identify statements in the new set whose antecedents are not exceptional and place them into $\mathcal{R}_1$. This process continues recursively until only classical statements remain in the set.

For this approach, we modified the way we check for exceptionalities. In the search-based encoding, for a given set, we start with each antecedent and derive all the atoms we can, given the rules in the set. We then check to see if there are complimentary atoms in the set (i.e. if $\mathcal{K} \cup \{\alpha\}$ is unsatisfiable).

In this implementation, the set keeps getting smaller and smaller; each time it does, we have to check for the exceptionality of every remaining antecedent again. We must identify what each antecedent allows us to derive in every set and subset to which it belongs. ASP (in single-shot solving [13]) does not provide a way to remove atoms after they have been derived, meaning that we cannot derive the same atom more than once. Given this, we need a way to easily associate a given atom with the antecedent that allowed us to derive it and the set in which it was derived.

To check for exceptionality in this new encoding, we use the rules in a set to build trees whose roots are antecedents and mark them with the set in which they were built. We then check if the trees have contradictory atoms anywhere in their nodes. The presence of such atoms will be referred to as a *clash*. Those that do, have their roots (antecedents) marked as "exceptional" for that given rank. We show an example of such a tree with a clash below.

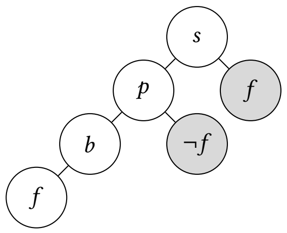

Figure 4: An example of a tree with a clash

The example tree in figure 4 allows us to conclude that the atom s is exceptional in the set that was used to create this tree due to the presence of the clash shown with the shaded nodes. To build the tree, we start with the root s; if there is a formula whose antecedent is s, we use it to add an edge from s to a node whose value is the consequent of the formula. This continues until there are no more formulas left to use.

4.2.2 `RationalClosure`. We extended the recursive approach used for `BaseRank` to `RationalClosure`. We mentioned in the previous subsection that we can break the `RationalClosure` algorithm into two parts. One is where we remove the lower ranks, and the other is where we check if the remaining ranked knowledge base entails the query. We use the recursive approach for the first part, and the second is the same as the search-based approach.

We start with the complete ranked knowledge base. To remove the lower ranks, we start by building a tree whose root is the antecedent of the query. We use the formulas in the knowledge base to build the tree as described in the previous subsubsection. After building the tree, we check for clashes in the tree. If any clash is found, the lowest rank is removed, and the process continues recursively. The recursion stops as soon as we have created a tree with no clashes or we have removed all defeasible ranks and are only left with the infinite rank. After the recursion has stopped, we check for entailment as before.

The first part of our recursive `RationalClosure` encoding has a recursive case and two base cases. The recursive case is encoded as described in the previous paragraph. The two base cases encode the two stopping conditions previously described. The second part of the encoding is where we check for entailment. This encoding is the same as it was in the search-based approach.

5 EXPERIMENTAL EVALUATION

In addition to the knowledge base generator, Racquel Dennison also implemented both `BaseRank` and `RationalClosure` [9]. These implementations also use a search-based approach, and they, along with our search-based implementations, will collectively be referred to as the "*search-based implementations*".

In this section, we present the experiments we have carried out to evaluate the performance of the search-based and recursive `BaseRank` encodings against Joel Hamilton's imperative `BaseRank` implementation [18] to determine if an ASP `BaseRank` implementation is computationally faster than an imperative one. We chose Joel Hamilton's implementation over Evashna Pillay's implementation [26] because it is basic and straightforward, whereas Evashna Pillay's implementation uses multithreading. Due to a lack of time, we did not get to evaluate `RationalClosure` as well.

5.1 Aims and Hypotheses

The main metrics that affect the performance of `BaseRank` are the number of defeasible statements and the number of ranks in the final ranked knowledge base. Based on this, we had two aims with our benchmarks:

- Determine how an increase in defeasible statement count impacts the runtime of each `BaseRank` implementation.
- Determine how an increase in rank count impacts the runtime of each `BaseRank` implementation.

For the search-based encodings, we hypothesised that the runtime of `BaseRank` would increase exponentially with a linear increase in the defeasible statement count. This is because all these implementations start by enumerating the entire search space, which increases exponentially with the number of defeasible statements. On the other hand, we expect the imperative implementation's runtime to increase linearly with the statement count. For this reason, we expect all the search-based implementations to perform better than the imperative one for smaller knowledge bases but worse for larger knowledge bases.

In our encodings, the number of ranks has no explicit impact on the encoding itself since we are using the defeasible statement count to enumerate the candidate solutions regardless of the rank count, and therefore, we expected it to not affect the performance. Unlike our search-based implementations, the imperative implementation ranks the statements sequentially. The more ranks the knowledge base has, the more ranks that need to be processed. For this reason, we expect the imperative implementation to perform better but get worse over time in comparison to the search-based approaches.

We also benchmarked the recursive `BaseRank` implementation against the imperative implementation for each of the evaluation aims. We hypothesise that the recursive implementation will perform better than the imperative implementation.

5.2 Experimental Design and Setup

We started by preparing test cases when setting up our experiments. We created two test cases, one for each of our evaluation aims. We used Aidan Bailey's knowledge base generator [3] for each of these instances to generate the required knowledge bases. We generated eight knowledge bases in the test case for the impact of the defeasible statement count. They each had one rank, and their defeasible statement count varied from five to 40 defeasible statements with an interval of five between consecutive knowledge bases.

We capped the defeasible statement count at 40 because all the search-based implementations took too long to rank knowledge bases with more than 40 defeasible statements. We believed that this collection of knowledge bases would still enable us to see the relationship between the defeasible statement count and the performance of each of the implementations, even though each knowledge base had relatively few defeasible statements.

For the rank count test case, we generated five knowledge bases with 30 statements each. Their rank counts were one, two, three, five, and ten. We needed to choose rank counts that allowed us to distribute the statements evenly over the ranks. We chose a uniform distribution when generating the knowledge bases because it was the only distribution that allowed us to have full control over the number of ranks and defeasible statements in our knowledge bases.

After all the knowledge bases had been generated, we ranked every one of them using each of the BaseRank implementations. Clingo automatically records the amount of time it takes for an ASP program to run, and this time is output to the console along with the final answer. Each encoding was run three times on each knowledge base, and the average of the three was recorded. We collected all these average times and plotted them. The results are in figures 5 and 7. The recursive BaseRank implementation was significantly faster than the search-based versions, so we had to plot its results separately to clearly observe its behaviour. We plotted the recursive BaseRank's results against the imperative BaseRank in figures 9 and 10 for test cases 1 and 2, respectively. In figure 6, we plotted the performance of Dennison's encoding against the defeasible statement count in isolation to clearly demonstrate its behaviour.

After plotting the results of the rank count test case, we noted that the number of ranks seems to impact the performance of the search-based BaseRank encodings. To confirm that this behaviour was not specific to the statement count we had chosen for the knowledge bases in test case 2, we created two more instances of the rank count test case with 20 and 32 statements. The results disproved our hypothesis that the number of ranks does not have an impact on the performance of BaseRank. After analysing the results of the search-based encodings, we found that for a given defeasible statement count, the number of candidate models passed to the optimisation statement changes based on the number of ranks of the final ranked knowledge base.

We hypothesised that for a given knowledge base, the number of candidate models left before optimisation had an impact on the runtime. To test this hypothesis, we recorded the number of models produced by each encoding before optimisation for each knowledge base. The results of these are shown in figure 8.

These benchmarks were conducted on an ASUS TUF A15 gaming laptop with an AMD Ryzen 7 6800H processor running at 3.2 GHz, 16 GB of RAM, and a 64-bit Windows 11 Home edition operating system. All other applications and unnecessary processes were closed before benchmarking to minimise their impact on the results.

5.3 Results

In the results shown below, the labels "Naive Search" and "Optimised Search" refer to the results for the straightforward search-based encoding and the optimised search-based encoding, respectively.

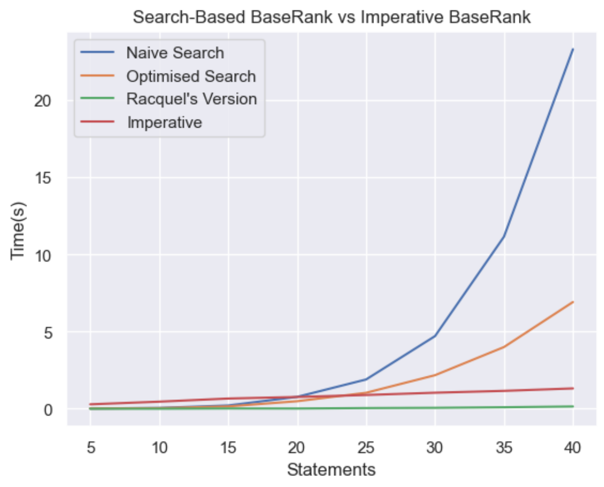

Figure 5: Impact of Statement Count on Search-Based BaseRank

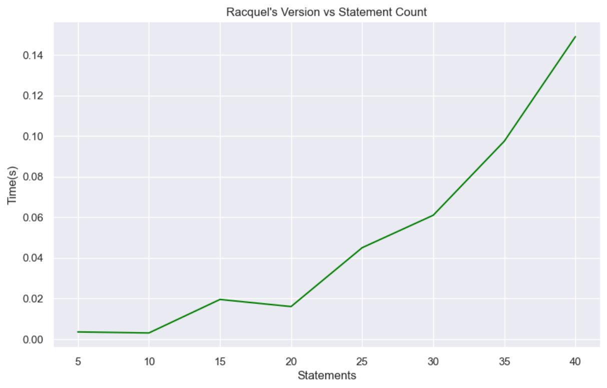

Figure 6: Impact of Statement Count on Racquel Dennison's Implementation

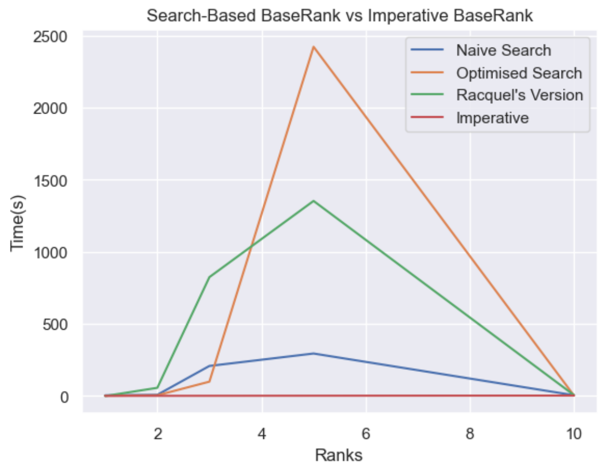

Figure 7: Impact of Rank Count on Search-Based BaseRank

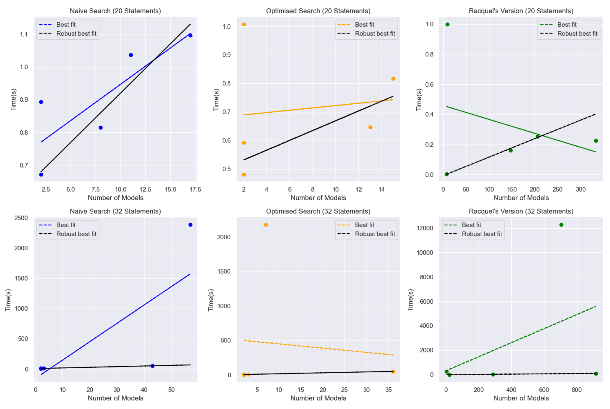

Figure 8: Impact of Model Count on Search-Based BaseRank

Figure 8 shows scatter plots of the number of models passed to the optimisation statement plotted against the total runtime for each knowledge base. Each scatter plot has a best-fit line to show the correlation between the number of models and the runtime for a given knowledge base. There is also a second best-fit line which is less impacted by outliers. The robust best-fit line was computed using Random Sample Consensus (RANSAC) to minimise the influence of outliers on the best-fit line and allow us to better see the general correlation between the two variables.

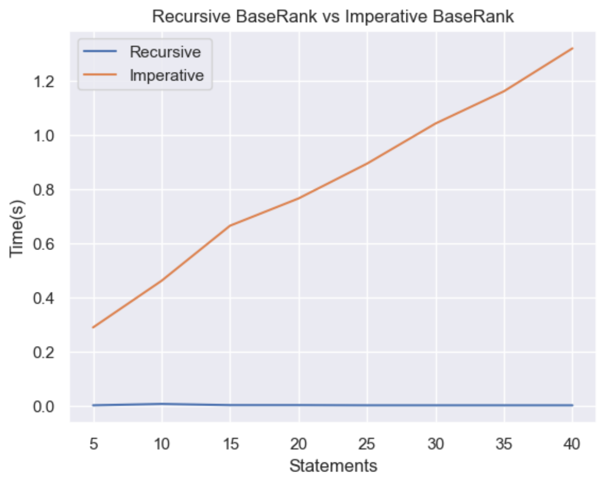

Figure 9: Impact of Statement Count on Recursive BaseRank

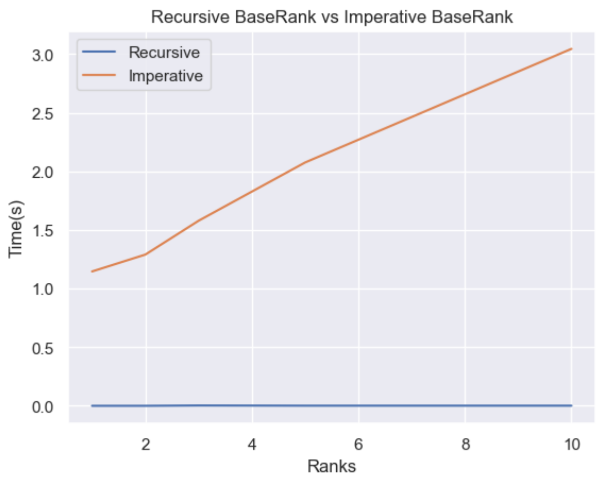

Figure 10: Impact of Rank Count on Recursive BaseRank

5.4 Discussion

We can see in figures 5 and 6 that the runtime of the search-based encodings increases exponentially with the statement count. This confirms our hypothesis and can be explained by the search space size. In both our search-based implementations, the search space size is $(n+1)^m$, where n is the maximum rank and m is the number of defeasible statements. However, because we do not know ahead of time how many ranks each ranked knowledge base is going to have, we have to use the highest possible value. With m defeasible statements, a ranked knowledge base cannot have more than m ranks. Therefore, in order to account for all possible rank counts, we had to set n to m. So in practice, the size of the search space ended being $(m+1)^m$. All the search-based encodings have search spaces whose sizes increase exponentially with the statement count.

We also see in figure 5 that the optimised version of our encoding performs better than our naive encoding. Based on our results, the optimised encoding only reduced the grounding time slightly. Therefore, the improved performance can be explained by the removal of redundant rules, which means that the solver no longer has to do duplicate work.

Dennison's encoding also performs better than our optimised search-based encoding. We believe that this is due to the encodings for characteristics (5) and (6) of a final ranked knowledge base. Dennison's approach differs from ours in that it has fewer and looser constraints than ours. This means that the additional time spent solving for candidate solutions that satisfy these characteristics puts both of our implementations at a disadvantage, leading to poorer performance.

We also see in figure 5 that the imperative implementation performs better than our search-based encodings but not better than Dennison's. This suggests that the search-based approach with looser constraints may be ideal for implementing BaseRank in ASP.

In figure 7, we see that the runtimes of all the search-based BaseRank encodings change with the increase in the rank count. This disproves the hypothesis that we initially had. To explain this behaviour, we will shift our focus to figure 8. In all six graphs, the runtime generally increases with the number of models, as shown by the robust best-fit line. This suggests that the number of models we end up with before clingo uses the minimize directive to find the optimal solution has a direct impact on the runtime. Despite at least one outlier in each plot, there seems to be a positive correlation between the runtime and the number of models that need optimisation. The minimize directive instructs clingo to keep searching for a better solution until the most optimal solution is found [23]. We can, therefore, expect this process to take longer the more models we have to optimise.

Another thing to note in figure 8 is that Dennison's encoding always ends up with significantly more models than both of our encodings. This can be explained by the fact that the constraints of our encodings are tighter than those of their encoding, which means that they can filter out more models before needing to use the minimize directive. However, despite having more models to optimise than our encodings, their encoding is still more efficient than ours. This suggests that the amount of work done to filter out unwanted models in our encodings is not worth the effort.

This also suggests that the search space is a more significant bottleneck than the minimize directive. This is because all the search-based implementations use identical optimisation statements, and Dennison's encoding is still performing better, despite having more models to process.

The larger the search space, the more work that needs to be done to filter candidate models before optimisation. It also means that we are more likely to end up with more models before optimisation, depending on the rank count of the final ranked knowledge base.

We also note in figure 7 that the imperative implementation outperforms all the search-based implementations. This shows that Dennison's search-based approach loses its advantage over the imperative approach as soon as we have more than one rank. We need a more efficient approach to implementing `BaseRank` in ASP. The best thing to do when looking for a more efficient way of ranking is to reduce the search space or remove the need for it entirely. This is what we did in our recursive approach.

The recursive approach performs significantly better than all the search-based and imperative implementations, as shown in figure 9. Clingo is able to finish solving the problem in under a second for all the instances of our knowledge bases. This is because we eliminated the need to search through a space of potentially millions of candidates, depending on the defeasible statement count.

The same behaviour can be observed in figure 10, with clingo solving all the problems in under a second there as well. We can see that the recursive approach is significantly more efficient than all the other implementations. This suggests that out of all the approaches we studied, the recursive approach is the most ideal way to solve base ranking in ASP.

6 DISCUSSION

In this section, we discuss some of the qualitative benefits and drawbacks we observed while using ASP to implement rational closure. The most significant benefit we noted was ASP's concise and compact encoding, which enabled us to implement our encodings with fewer lines of code than would be possible in most imperative languages. As a result, the final code was more readable and easier to debug.

Another advantage of ASP is its high degree of flexibility and modularity. It is easy to reuse or combine parts of the program in different ways, as they are not strictly coupled. This makes the programs easier to modify or extend.

However, a notable drawback of using ASP is its relatively steep learning curve. It has been well-documented that the declarative programming paradigm can be challenging for individuals with an exclusively imperative background [6], a difficulty we also encountered. This issue is compounded by the limited resources and small community surrounding ASP, which makes it harder to troubleshoot problems or find guidance when trying to express certain concepts in ASP.

7 CONCLUSIONS AND FUTURE WORKS

Many introductory ASP texts, such as [6, 11, 22, 23], present the ASP problem-solving approach as a search problem with two steps: a generation step and a filtering step to find the correct model. However, after using this approach as the basis for our implementation of `BaseRank` in ASP, we found it to be less than ideal. This is due to the exponential relationship between the number of defeasible statements and the size of the search space.

We found that a recursive approach to this problem is more effective. It resembles the imperative method of "building" a solution rather than searching for it. Additionally, ASP proved ideal for implementing `BaseRank` and `RationalClosure`, as the resulting implementations were more readable, less prone to errors, and easier to maintain and extend.

While the recursive approach appears promising, its limitations are not yet fully understood. We recommend that future researchers explore its behavior with larger knowledge bases. We also suggest benchmarking our `RationalClosure` implementations to assess their performance compared to previous implementations. Furthermore, extending our implementations of `BaseRank` and `RationalClosure` to handle more complex statements would be a valuable area for further study.

REFERENCES

[1] Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub. 2005. A Glimpse of Answer Set Programming. *Künstliche Intell.* 19, 1 (2005), 12. https://www.cs.uni-potsdam.de/~torsten/Papers/asp4ki.pdf

[2] Theofanis I. Aravanis and Pavlos Peppas. 2017. Belief Revision in Answer Set Programming. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics* (Larissa, Greece) *(PCI '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 5 pages. https://doi.org/10.1145/3139367.3139387

[3] Aidan Bailey. 2021. *Scalable Defeasible Reasoning.* BSc (Hons) Project. University of Cape Town, Cape Town, South Africa. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/BLYAID001_SCADR.pdf

[4] Chitta Baral. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving* (1st ed.). Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511543357

[5] Ronald J. Brachman and Hector J. Levesque. 2004. *Knowledge Representation and Reasoning* (1st ed.). Elsevier, San Francisco, USA. https://doi.org/10.1016/B978-1-55860-932-7.X5083-3

[6] Martin Brain, Owen Cliffe, and Marina De Vos. 2009. A Pragmatic Programmer's Guide to Answer Set Programming. In *Proceedings of the SEA09: Software Engineering for Answer Set Programming (Department of Computer Science Technical Report Series, CSBU-2009-20)*, M. De Vos and S. Torsten (Eds.). Bath, U.K., 49–63. https://purehost.bath.ac.uk/ws/portalfiles/portal/327621/CSBU-2009-20.pdf

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

[8] Victoria Chama. 2020. *Explanation for Defeasible Entailment.* Master's Thesis. University of Cape Town, Cape Town, South Africa. http://hdl.handle.net/11427/32206

[9] Racquel Dennison. 2024. *Defeasible Conditionals in Answer Set Programming.* BSc (Hons) Project. University of Cape Town, Cape Town, South Africa.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. 2008. A user's guide to gringo, clasp, clingo, and iclingo. Technical report. https://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/clingo_guide.pdf University of Potsdam, Unpublished manuscript.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice* (1st ed.). Springer Cham. https://doi.org/10.1007/978-3-031-01561-8

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR* abs/1405.3694 (2014). https://doi.org/10.48550/arXiv.1405.3694 arXiv:1405.3694

[13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1 (2019), 27–82. https://doi.org/10.1017/S1471068418000054

[14] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* 24, 2 (2011), 107–124. https://doi.org/10.3233/AIC-2011-0491

[15] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. *clasp* : A Conflict-Driven Answer Set Solver. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4483)*, Chitta Baral, Gerhard Brewka, and John S. Schlipf (Eds.). Springer, 260–265. https://doi.org/10.1007/978-3-540-72200-7_23

[16] Martin Gebser, Torsten Schaub, and Sven Thiele. 2007. GrinGo : A New Grounder for Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4483)*, Chitta Baral, Gerhard Brewka, and John S. Schlipf (Eds.). Springer, 266–271. https://doi.org/10.1007/978-3-540-72200-7_24

[17] Sampada Gulavani. 2019. Knowledge Representation Approaches in Artificial Intelligence. *International Journal of Science and Research* 8 (10 2019), 1699–1701. Issue 10. https://www.ijsr.net/getabstract.php?paperid=ART20202230

[18] Joel Hamilton. 2021. *An Investigation into the Scalability of Rational Closure.* BSc (Hons) Project. University of Cape Town, Cape Town, South Africa. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/SCADR_HMLJOE001.pdf

[19] Adam Kaliski. 2020. *An Overview of KLM-Style Defeasible Entailment.* Master's thesis. University of Cape Town, Cape Town, South Africa. http://hdl.handle.net/11427/32743

[20] Hans Kleine Büning and Theodor Lettmann. 1999. *Propositional logic - deduction and algorithms.* Cambridge tracts in theoretical computer science, Vol. 48. Cambridge University Press, New York, USA. https://dl.acm.org/doi/10.5555/554728

[21] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic Reasoning, Preferential Models and Cumulative Logics. *Journal of Artificial Intelligence* 44, 1-2 (1990), 167–207. https://doi.org/10.1016/0004-3702(90)90101-5

[22] Nicola Leone and Francesco Ricca. 2015. Answer Set Programming: A Tour from the Basics to Advanced Development Tools and Industrial Applications. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures (Lecture Notes in Computer Science, Vol. 9203)*, Wolfgang Faber and Adrian Paschke (Eds.). Springer, 308–326. https://doi.org/10.1007/978-3-319-21768-0_10

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

[24] John W. Lloyd. 1994. Practical Advtanages of Declarative Programming. In *1994 Joint Conference on Declarative Programming, GULP-PRODE'94 Peñiscola, Spain, September 19-22, 1994, Volume 1*, María Alpuente, Roberto Barbuti, and Isidro Ramos (Eds.). 18–30. https://www.researchgate.net/publication/242503635_1994_Joint_Conference_on_Declarative_Programming_GULP-PRODE'94_Peniscola_Spain_September_19-22_1994

[25] Kody Moodley, Thomas Meyer, and Ivan José Varzinczak. 2012. A defeasible reasoning approach for description logic ontologies. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (Pretoria, South Africa) *(SAICSIT '12)*. Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/2389836.2389845

[26] Evashna Pillay. 2022. *An Investigation into the Scalability of Rational Closure V2.* BSc (Hons) Project. University of Cape Town, Cape Town, South Africa. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2022/pillay_thakorvallabh.zip/files/PLLEVA005_SCADR2_FinalPaper.pdf

[27] Matthias Thimm. 2014. Tweety: A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter (Eds.). AAAI Press. http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7811