UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2023

Title: Justifications for Classical Entailment

Author: Orefile Morule

Project Abbreviation: CEE

Supervisor(s): Professor Tommie Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	10
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Justifications for Classical Entailment

Orefile Morule
mrlore001@myuct.ac.za
University of Cape Town
Cape Town, South Africa

ABSTRACT

Knowledge Representation and Reasoning allows for Artificial Intelligence systems to store knowledge in symbolic format which makes reasoning about that knowledge easier. Conclusions from reasoning systems can be hard to understand. Explanation services are crucial to reasoning systems as they provide both the users and engineers of knowledge bases an explanation as to why conclusions were made by the systems.

We present a type of explanation called a Justification which gives the minimal subset of knowledge that makes a conclusion hold. Given a query, the Classical Justification Algorithm uses its sub-algorithms to extract the necessary knowledge from a knowledge base and then outputs the minimal subset of the knowledge base that renders the query true.

Propositional Logic is the logic used in this paper and it has a property called monotonicity, which means that it cannot retract a conclusion even when new information is added to the knowledge base. This results in the inability to handle exceptions. Defeasible reasoning allows for exception handling and is the focus of the project done by my project partner. The combination of our two projects resulted in a system which can take in both exceptional and classical knowledge bases and use the appropriate algorithms to get an explanation.

We give a software application that implements the Classical Justification Algorithm and is accessible through a Graphical User Interface.

CCS Concepts

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

Keywords

artificial intelligence, knowledge representation and reasoning, defeasible reasoning, classical reasoning

1 INTRODUCTION

Formal logic in the field of *Artificial Intelligence* allows us to represent the state of the world in a way that is easy for automated systems to understand. Once this is achieved, we would like the systems to be able to infer new knowledge by reasoning using what it already knows. This is the concept of *Knowledge Representation and Reasoning* in Artificial Intelligence. In addition to reasoning, we would like to know how Artificial Intelligence systems come to conclusions when asked if something holds or is *entailed* in the world as it knows it. Explanations are an important part of reasoning services as they offer a way to debug any unwanted inferences and they explain why inferences we want to have, hold.

We will start this paper by giving background on *Propositional Logic* which is the formal logic this paper will be restricted to. We will then show how logic ties in with Knowledge Representation and Reasoning. Then we will explain Justifications for Classical Entailment by going through the different sub-algorithms used for computing justifications.

2 BACKGROUND

2.1 Propositional Logic

Propositional Logic is the branch of formal logic concerned with joining and/or modifying statements to create more complex statements [6]. Propositional logic contains indivisible statements such as, "philosophers are wise," or "humans are mortal," referred to as *atoms*. Atoms are assigned a truth value of *true* or *false*, but not both, and represent what we know about the world [3]. These propositional atoms can be combined using *logical connectives* to create more complex statements.

Formal logic comprises of various logical systems including this paper's focus ***Classical Reasoning***. Formal logic utilises a structured approach to reasoning, characterised by syntax, semantics, and deduction. The *syntactic* framework comprises symbols and rules governing legal statements, while *semantics* assign meaning to these symbols within different contexts. Deduction in formal logic ensures that valid conclusions emerge from sound premises through the application of logical rules [11].

2.1.1 Syntax

In the language of Propositional Logic, atoms are represented as lowercase letters. A statement such as, "humans are mortal," could then be represented as h. Atoms are joined or modified using logical signs or *connectives* to create more complex statements called *formulas*. The connectives in the language are *conjunction*, *disjunction*, *negation*, *implication*, and *equivalence*. They are used in place of the truth-functional operators, *"and"*, *"or"*, *"not"*, *"if … then"*, and *"if and only if"*, respectively. The negation connective is unary, meaning it takes only one operand, while the rest of the connectives are binary, meaning that they take on two operands.

See the table below for connectives and their symbols.

Name	Meaning	Symbol
disjunction	or	$\lor$
conjunction	and	$\land$
negation	not	$\neg$
implication	if then	$\rightarrow$
equivalence	if and only if	$\leftrightarrow$

Figure 1: The Boolean operators used in propositional logic

The set of atoms is denoted as $\mathcal{P}$. The formula $p \rightarrow q$ is read "p implies q" and means if p is *true* then q is *true*. The set of all formulas can be denoted as $\mathcal{L}$. We can define formulas inductively as follows:

Definition 2.1. (Propositional Formulas) [14].

- For every atom $p \in \mathcal{P}, p \in \mathcal{L}$
- For a formula $\alpha \in \mathcal{L}, \neg\alpha \in \mathcal{L}$
- If the formulas $\alpha, \beta \in \mathcal{L}$, then $\alpha \wedge \beta, \alpha \vee \beta, \alpha \rightarrow \beta, \alpha \leftrightarrow \beta \in \mathcal{L}$

2.1.2 Semantics

An *interpretation* $\mathcal{I}$ is a function used to assign a truth value to an atom. A *truth table* is a finite table used to show each possible combination of truth values of atoms. Each line in a truth table corresponds to a different interpretation [6]. A proposition having a truth value *true*, is called *Satisfaction*. Formally:

Definition 2.2. (Interpretation). An interpretation is a total function $\mathcal{I} : \mathcal{P} \mapsto \{T, F\}$ that assigns one of the truth values *true* or *false* to every atom $p \in \mathcal{P}$.

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \rightarrow \beta$	$\alpha \leftrightarrow \beta$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	F	F	F	T	T

Figure 2: A truth table for the Boolean operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$

Definition 2.3. (Satisfiability). An Interpretation $\mathcal{I}$ satisfies some $p \in \mathcal{P}$ if $I(p) = T$ and it is denoted by $I \Vdash p$. An interpretation does not satisfy some $p \in \mathcal{P}$ if $I(p) = F$ and it is denoted by $I \nVdash p$. To derive the truth value of a propositional formula under any interpretation $\mathcal{I}$, we look at the truth values of the individual atoms that make up the formula and use the semantics and precedence order of the connectives used to evaluate whether the formula is *true* or *false*. The evaluation for any formulas $\alpha, \beta \in \mathcal{L}$ can be done as follows:

- $I \Vdash \neg\alpha$ if and only if $I \nVdash \alpha$
- $I \Vdash \alpha \wedge \beta$ if and only if $I \Vdash \alpha$ and $I \Vdash \beta$
- $I \Vdash \alpha \vee \beta$ if and only if $I \Vdash \alpha$ or $I \Vdash \beta$ or both
- $I \Vdash \alpha \rightarrow \beta$ if and only if $I \nVdash \alpha$ or $I \Vdash \beta$ or both
- $I \Vdash \alpha \leftrightarrow \beta$ if and only if $I \Vdash \alpha \rightarrow \beta$ and $I \Vdash \beta \rightarrow \alpha$

If an interpretation satisfies a formula, then it is a model of that formula.

Definition 2.4. (Model). An interpretation $\mathcal{I}$ is a *model* of α if for any $\alpha \in \mathcal{L}$ and an interpretation I, I satisfies α.

2.2 Classical Reasoning

One of the main aspects of formal logic is *deduction*, where conclusions are drawn from given premises using established rules of inference. *Classical Entailment* provides a way to formally express how conclusions are drawn from premises:

Definition 2.5. (Classical Entailment). Let $\mathcal{T}$ be a set of formulas and α a formula. $\mathcal{T}$ entails α, written $\mathcal{T} \models \alpha$, if and only if the models of $\mathcal{T}$ are a subset of the models of α.

A finite set of propositional formulas is referred to as a *knowledge base.*

Definition 2.6. (Knowledge Base). Let $\mathcal{K}$ be a set of formulas. If $\mathcal{K}$ is finite, it is referred to as a knowledge base.

2.2.1 Reasoning with SAT Solvers

The concepts of entailment and satisfiability can be extended to knowledge bases. A knowledge base that has a model is satisfiable.

Definition 2.7. (Knowledge Base Satisfiability) [14]. Let a knowledge base $\mathcal{K} = \{\alpha_1, \alpha_2, \alpha_3, ...\alpha_n\}$. If there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \Vdash \alpha_i$ for all i where $1 \leq i \geq n$, then $\mathcal{K}$ is satisfiable. If for all interpretations $\mathcal{I}$ there is an i such that $\mathcal{I} \nVdash \alpha$, then $\mathcal{K}$ is *unsatisfiable.*

Let us consider a knowledge base $\mathcal{K}$ along with a query α. To check whether $\mathcal{K} \models \alpha$, we add $\neg\alpha$ to $\mathcal{K}$. If and only if $\mathcal{K} \cup \{\neg\alpha\}$ is unsatisfiable does $\mathcal{K} \models \alpha$ [3]. Let us consider an example to illustrate this logic. Let a knowledge base $\mathcal{K} = \{human \rightarrow mortal, Aristotle \rightarrow human, mortal \rightarrow die\}$ and a query = $human \rightarrow die$. Therefore, we add $\neg$ ($human \rightarrow die$) to $\mathcal{K}$. It is easy to see that the set $\mathcal{K}' = \{human \rightarrow mortal, Aristotle \rightarrow human, mortal \rightarrow die, \neg (human \rightarrow die)\}$ is unsatisfiable. We can use deduction to see that the formulas $human \rightarrow mortal$ and $mortal \rightarrow die$ lead to a conclusion that $human \rightarrow die$, however, $\neg$ ($human \rightarrow die$) is a direct contradiction to that contradiction which renders $\mathcal{K}'$ unsatisfiable.

The above example illustrates that checking whether a knowledge base $\mathcal{K}$ entails a query α can be reduced to checking whether $\mathcal{K} \cup \{\neg\alpha\}$ is satisfiable.

We have used a SAT Solver, SAT4J, as our satisfiability checker for this project[1]. This tool allows us to automate the reasoning process and, therefore, check if a query is entailed by a knowledge base or not.

2.3 Explanation Services

In the field of formal logic systems, the motivation behind introducing explanation services comes from the increasing reliance on automated reasoning systems across various domains, such as healthcare, transport, and security [13]. As these systems perform complex logical operations, the challenge lies in rendering their outcomes understandable and verifiable by humans. Explanation services address this challenge by providing a clear breakdown of the logical steps taken by the automated reasoning system. This helps users, both experts and non-experts, to gain insights into the logical processes undertaken.

Horridge provides three common use cases that make explanation services valuable [5]. The first is that a user might simply want to understand why an entailment holds. The second is that a knowledge base engineer might have an inconsistent knowledge base, and by viewing explanations for an entailment, they can diagnose the cause of the inconsistency. The third is that a user sees a knowledge base for the first time and tries to understand its complexity by counting the number of entailments or the average number of explanations for an entailment.

3 PROJECT AIMS

The main aims of this project were to:

- Provide the theory underlying classical reasoning.
- Provide the theory and motivation for explanation services, specifically justifications.
- Provide and explain the sub-algorithms that make up the Classical Justifications Algorithm.
- Develop a tool that implements the Classical Justifications algorithm.
- Combine the work presented for classical justifications with the work presented by Hamayobe [4] for defeasible justifications.

4 JUSTIFICATIONS FOR CLASSICAL ENTAILMENT

4.1 Overview

We focus on a specific type of explanation called a *justification*. A justification is a subset of the knowledge base that entails a given query and has a property that if any element is removed from the subset, it will no longer entail the query.

Definition 4.1. (Classical Justification). *Let $\mathcal{K}$ be a knowledge base and have a query α such that $\mathcal{K} \models \alpha$; the set of formulas $\mathcal{J}$ is a justification for $\mathcal{K} \models \alpha$ if $\mathcal{J} \subseteq \mathcal{K}$, $\mathcal{J} \models \alpha$ and for all $\mathcal{J}' \subset \mathcal{J}$, it holds that $\mathcal{J}' \not\models \alpha$* [14].

For example, a knowledge base $\mathcal{K}$ = {*human* $\longrightarrow$ *mortal*, *Aristotle* $\longrightarrow$ *human*, *mortal* $\longrightarrow$ *die*}. Given a query *human* $\longrightarrow$ *die*, it is clear that $\mathcal{K} \models$ *human* $\longrightarrow$ *die*. The justification for the entailment is the set $\mathcal{J}$ = {*human* $\longrightarrow$ *mortal*, *mortal* $\longrightarrow$ *die*}.

In the instance where a query α is not entailed by the knowledge base, the justification given is simply the knowledge base itself. The intuition behind this is that the reason why the query does not hold is because nothing in the knowledge base renders it true.

Definition 4.2. (Classical Justification for Non-entailment). *Let $\mathcal{K}$ be a knowledge base and have a query α such that $\mathcal{K} \not\models \alpha$. The justification for $\mathcal{K} \not\models \alpha$ is $\mathcal{K}$* [14].

For example, a knowledge base $\mathcal{K}$ = {*human* $\longrightarrow$ *mortal*, *Aristotle* $\longrightarrow$ *human*, *mortal* $\longrightarrow$ *die*}. Given a query *mortal* $\longrightarrow$ *human*, it is clear that $\mathcal{K} \not\models$ *mortal* $\longrightarrow$ *human*. The justification for the entailment is then $\mathcal{K}$.

4.2 Classical Justification Algorithm

4.2.1 Overview

Algorithms for computing justifications are usually categorised using two axes of classification [5]. The first axis is the *single-all-axis* which classifies whether an algorithm computes only one or all the justifications for an entailment. The second axis is the *reasoner-coupling-axis* which classifies whether an algorithm is a *black-box* algorithm or a *glass-box* algorithm. The *black-box* and *glass-box* algorithms compute justifications and are distinct based on their relationship with the reasoner. A *black-box* algorithm computes justifications independent of the reasoner, while a *glass-box* algorithm is tightly interwoven with the reasoner.

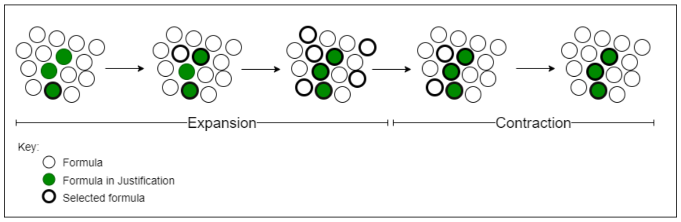


Figure 3: Depiction of the Expand-Contract strategy.

The idea behind the Classical Justification Algorithm is that given a query that is entailed by the knowledge base, the algorithm first expands a subset of the knowledge base until it entails the query. This will be done by the ExpandFormulas sub-algorithm. After this, the algorithm removes formulas from this subset that do not play a role in the entailment. This will be done by the ContractFormulas sub-algorithm. Figure 3 depicts the Expand-Contract strategy.

4.2.2 Expand Formulas

The goal of this algorithm is to find a subset of the knowledge base that entails the query. Instead of linearly adding formulas to create this subset, Horridge proposes finding formulas whose *signature* intersects with the *signature* of the query[5].

Definition 4.3. (Signature). *The signature of a formula is all the propositional atoms that appear in that formula.*

These formulas are the initial elements of subset S.

Let us consider a knowledge base $\mathcal{K}$ = {*philosopher* $\longrightarrow$ *thinker*, *Aristotle* $\longrightarrow$ *philosopher*, *thinker* $\longrightarrow$ *ponder_profound_questions*, *ponder_profound_questions* $\longrightarrow$ *wise*, *Aristotle* $\longrightarrow$ *wise*, *wise* $\longrightarrow$ *respected*}. We can simplify this by using a single letter to represent each propositional atom. p for philosopher, t for thinker, A for Aristotle, q for ponder_profound_questions, w for wise and r for respected. Therefore $\mathcal{K} = \{p \longrightarrow t, A \longrightarrow p, t \longrightarrow q, q \longrightarrow w, A \longrightarrow w, w \longrightarrow r\}$.

If we have a query $A \models r$, we can define a set Σ as the signature of the query. This means Σ := {A, r}. Our initial subset S would then be $\{A \longrightarrow p, A \longrightarrow w, w \longrightarrow r\}$.

We define *IntersectingFormulas*$(\Sigma, \mathcal{K})$ as a method that finds all the propositional formulas in $\mathcal{K}$ that have signatures that intersect with Σ.

The algorithm first checks if the knowledge base entails the query by running it through a SAT solver. If the result is non-entailment then an empty set is returned as there is no entailment to justify. If $\mathcal{K} \models \alpha \longrightarrow \beta$ then S and S' are initialised to empty sets while Σ is set to the signature of the query.

Example: (ExpandFormulas)

Let us consider $\mathcal{K} = \{p \longrightarrow t, A \longrightarrow p, t \longrightarrow q, q \longrightarrow w, A \longrightarrow w, w \longrightarrow r\}$. Our query is $A \longrightarrow q$.

(1) The knowledge base does entail this query therefore Σ is initialised to {A, q}.

(2) In the while loop line 9 assigns $\{A \longrightarrow p, t \longrightarrow q, q \longrightarrow w, A \longrightarrow w\}$ to S. S does not entail the query so it needs to be expanded.

(3) In the second iteration of the loop line 9 expands S to $\{A \longrightarrow p, t \longrightarrow q, q \longrightarrow w, A \longrightarrow w, p \longrightarrow t, w \longrightarrow r\}$. The set S now does entail the query, meaning that the algorithm can stop.

Algorithm 1: ExpandFormulas

```
Input: Knowledge base $\mathcal{K}$ and a query $\alpha \rightarrow \beta$
Output: Set $S$
1  if $\mathcal{K} \not\models \alpha \rightarrow \beta$ then
2      return $\emptyset$
3  else
4      $S := \emptyset$
5      $S' := \emptyset$
6      $\Sigma := signature(\alpha \rightarrow \beta)$
7  while $S' \neq S$ do
8      $S' = S$
9      $S = S \cup IntersectingFormulas(\Sigma, \mathcal{K})$
10     if $S \models \alpha \rightarrow \beta$ then
11         return $S$
12     $\Sigma = signature(S)$
13 return $S$
```

It is important to note the role of the set S'. It keeps track of whether or not the set S expanded in the previous loop iteration. If there was no expansion then it means that the set S can no longer be expanded beyond its current size, and therefore the algorithm needs to end to avoid an infinite loop.

4.2.3 Contract Formulas

We note that although the output of the ExpandFormulas example does entail the query, it is not a minimal set. This introduces the ContractFormulas algorithm which aims to remove the formulas from the set S that do not contribute to the entailment.

We use a divide and conquer technique to remove formulas from the set S as described by Horridge [5]. The set S is split into two halves, S_1 and S_2. If S_1 entails the query, then S_2 is not considered and, conversely, S_1 is not considered if S_2 entails the query. If neither one of S_1 and S_2 entail the query then S_1 is arbitrarily chosen to itself be split into half and one the halves is combined with S_2 and then we check whether this combination entails the query. This process of halving and combining continues until an entailment occurs.

Our algorithm presents the recursive implementation of the divide and conquer strategy. We construct a binary tree where each child node represents one half of the parent node.

Algorithm 2: ContractFormulas

```
Input: Set $S$ and a query $\alpha \rightarrow \beta$ such that $S \models \alpha \rightarrow \beta$
Output: Set $S' \subseteq S$ such that such that $S' \models \alpha \rightarrow \beta$
1  return $ContractRecursive(\emptyset, \mathcal{K}, \alpha \rightarrow \beta)$
```

Example: (ContractFormulas)

Let us consider $\mathcal{K} = \{p \rightarrow t, A \rightarrow p, w \rightarrow r, A \rightarrow w, q \rightarrow w, t \rightarrow q\}$. Our query is $p \rightarrow q$.

(1) The ContractFormulas algorithm calls the ContractRecursive algorithm with the arguments $S_{combination} = \emptyset, S_{complete} = \mathcal{K}$ and our query. The rest of the explanation will detail what goes on in the ContractRecursive algorithm.

Algorithm 3: ContractRecursive

```
Input: Set $S_{combination}$, Set $S_{complete}$ and a query $\alpha \rightarrow \beta$
       such that $S_{complete} \models \alpha \rightarrow \beta$
Output: Set $S' \subseteq S_{complete}$ such that such that $S' \models \alpha \rightarrow \beta$
1  if $|S_{complete}| == 1$ then
2      return $S_{complete}$
3  $S_1 :=$ Halve($S_{complete}$, 1)
4  $S_2 :=$ Halve($S_{complete}$, 2)
5  if $S_{combination} \cup S_1 \models \alpha \rightarrow \beta$ then
6      return $ContractRecursive(S_{combination}, S_1, \alpha \rightarrow \beta)$
7  if $S_{combination} \cup S_2 \models \alpha \rightarrow \beta$ then
8      return $ContractRecursive(S_{combination}, S_2, \alpha \rightarrow \beta)$
9  $S'_1 := ContractRecursive(S_{combination} \cup S_2, S_1, \alpha \rightarrow \beta)$
10 $S'_2 := ContractRecursive(S_{combination} \cup S'_1, S_2, \alpha \rightarrow \beta)$
11 return $S'_1 \cup S'_2$
```

(2) In the first recursion, we halve $S_{complete}$ to get $S_1 = \{p \rightarrow t, A \rightarrow p, w \rightarrow r\}$ and $S_2 = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$. $S_{combination} \cup S_1$ does not entail $p \models q$ and neither does $S_{combination} \cup S_2$. Thus, we move to line 9 which calls the recursive algorithm in order to assign a value to S'_1.

(3) In the next recursion the algorithm is given the arguments $S_{combination} = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$, $S_{complete} = \{p \rightarrow t, A \rightarrow p, w \rightarrow r\}$ and our query. We halve $S_{complete}$ to get $S_1 = \{p \rightarrow t, A \rightarrow p\}$ and $S_2 = \{w \rightarrow r\}$. $S_{combination} \cup S_1$ does entail $p \models q$. The algorithm therefore returns the output of the next recursion as shown in line 6.

(4) In the next recursion the algorithm is given the arguments $S_{combination} = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$, $S_{complete} = \{p \rightarrow t, A \rightarrow p\}$ and our query. We halve $S_{complete}$ to get $S_1 = \{p \rightarrow t\}$ and $S_2 = \{A \rightarrow p\}$. $S_{combination} \cup S_1$ does entail $p \models q$. The algorithm therefore returns the output of the next recursion as shown in line 6.

(5) In the next recursion the algorithm is given the arguments $S_{combination} = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$, $S_{complete} = \{p \rightarrow t\}$ and our query. Since $S_{complete}$ only has one element, we return it as per line 2. This value propagates to the first recursive call made and thus $S'_1 = \{p \rightarrow t\}$.

(6) We can now move to line 10 of the algorithm which uses the value of S'_1 to obtain S'_2. We are now back to the first call to the recursive algorithm, meaning our arguments are as follows:

- $S_{combination} = \emptyset$
- $S_{complete} = \{p \rightarrow t, A \rightarrow p, w \rightarrow r, A \rightarrow w, q \rightarrow w, t \rightarrow q\}$
- $S_1 = \{p \rightarrow t, A \rightarrow p, w \rightarrow r\}$
- $S_2 = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$
- $S'_1 = \{p \rightarrow t\}$

We call the recursive algorithm to get the value of S'_2.

(7) In the next recursion the algorithm is given the arguments $S_{combination} = \{p \rightarrow t\}$, $S_{complete} = \{A \rightarrow w, q \rightarrow w, t \rightarrow q\}$ and our query. We halve $S_{complete}$ to get $S_1 = \{A \rightarrow w, q \rightarrow w\}$ and $S_2 = \{t \rightarrow q\}$. $S_{combination} \cup S_1$ does not entail $p \models q$ but $S_{combination} \cup S_2$ does entail $p \models q$. The algorithm therefore returns the output of the next recursion as shown in line 8.

(8) In the next recursion the algorithm is given the arguments $S_{combination} = \{p \rightarrow t\}$, $S_{complete} = \{t \rightarrow q\}$ and our query. Since $S_{complete}$ only has one element, we return it as per line 2. This value propagates to the recursive call made for S'_2 and thus $S'_2 = \{t \rightarrow q\}$.

(9) Finally, the algorithm returns $S'_1 \cup S'_2$ as the set $S' = \{p \rightarrow t, t \rightarrow q\}$.

4.2.4 Compute Single Justification

One reason why computing a single justification for an entailment is important is that non-experts may merely want an explanation of why an entailment holds. This means that they are satisfied with seeing only one justification. An engineer can also get enough information from just a single justification to debug the system.

To compute a single justification you simply need to use the ExpandFormulas sub-algorithm to find a set $S \subseteq \mathcal{K}$ that entails the query and then use the ContractFormulas sub-algorithm to find the minimal subset of S that entails the query.

```
Algorithm 4: ComputeSingleJustification
  Input: Knowledge base $\mathcal{K}$ and a query $\alpha \rightarrow \beta$ such that
         $\mathcal{K} \models \alpha \rightarrow \beta$
  Output: Justification $\mathcal{J}$
1 if $\alpha \rightarrow \beta \in \mathcal{K}$ then
2     return $\alpha \rightarrow \beta$
3 $S$ := ExpandFormulas ($\mathcal{K}$, $\alpha \rightarrow \beta$)
4 if $S == \emptyset$ then
5     return $\emptyset$
6 $\mathcal{J}$ = ContractFormulas($S$, $\alpha \rightarrow \beta$)
7 return $\mathcal{J}$
```

4.2.5 Compute All Justifications

We compute all justifications of an entailment by using Reiter's *Hitting Set Tree* algorithm [9]. This algorithm has foundations in the field of *Model Based Diagnosis*. When a system is faulty, the objective is to identify a diagnosis, which is a set of system components whose erroneous operation provides an explanation for the faulty system behaviour [10]. The idea is that faults in a system are caused by a bad interaction between distinct sets of system components. There are usually multiple, or even an exponential amount of diagnoses for a faulty system, therefore Reiter's algorithm presents a technique that finds the minimal sets of explanations for a fault. The algorithm constructs diagnoses as *hitting sets of conflicts* [9]. A conflict set is a set of system components that cannot all be fault-free given how the system is expected to behave [10]. A *minimal conflict set* is a conflict set that has no subset which is a conflict set. In the context of a set of system conflict sets, a *hitting set* refers to a set that has at least one element from every individual conflict set. A *minimal hitting set* is a hitting set that has no subset which is a hitting set. Formally, a diagnosis is referred to as a minimal hitting set [5].

Given the concepts above, we can note the parallels between Model Based Diagnosis and Computing Justifications. A knowledge base corresponds to a system, an entailment corresponds to a fault, and a set of justifications corresponds to a minimal hitting set.

We will now give a brief overview of how the hitting set tree algorithm works. Given a knowledge base $\mathcal{K}$ and a query α such that $\mathcal{K} \models \alpha$, a hitting set tree consists of nodes that are labelled with the justifications for $\mathcal{K} \models \alpha$ and edges that are labelled with formulas found in $\mathcal{K}$. Every non-leaf node n is connected to a child node n' through an edge which is labelled with a formula β such that β is in the label of n but not in the label of n'. If the label of n' is an empty set, then it is a leaf node. For any node n'' , the set of formulas that label the path from n'' to the root node does not intersect with the justification that labels n''.

```
Algorithm 5: ComputeAllJustifications
   Input: Knowledge base $\mathcal{K}$ and a query $\alpha \rightarrow \beta$ such that
          $\mathcal{K} \models \alpha \rightarrow \beta$
   Output: Set of Justifications $\mathcal{J}$
 1 Just$_{set}$ :=$\emptyset$
 2 Root_Just := ComputeSingleJustification($\mathcal{K}$, $\alpha \rightarrow \beta$)
 3 Root_Node := CreateNode($\mathcal{K}$, Root_Just)
 4 Enqueue(Root_Node, $Q$)
 5 AddRoot($HS_{tree}$)
 6 while $Q \neq \emptyset$ do
 7     Node = Dequeue(Q)
 8     Node_Just = GetJust(Node)
 9     Node$\mathcal{K}$B = Get$\mathcal{K}$(Node)
10     for each formula $\delta \in$ Node_Just do
11         Smaller$\mathcal{K}$ = Node$\mathcal{K}$B \ $\delta$
12         Smaller$\mathcal{K}_{Just}$ =
             ComputeSingleJustification(Smaller$\mathcal{K}$, $\alpha \rightarrow \beta$)
13         Child_Node = CreateNode(Smaller$\mathcal{K}$,
             Smaller$\mathcal{K}_{Just}$)
14         $HS_{tree}$ = $HS_{tree}$ $\cup$ Child_Node
15         if Child_Node $\neq \emptyset$ then
16             Enqueue(Child_Node)
17             Just$_{set}$ = Just$_{set}$ $\cup$ Smaller$\mathcal{K}_{Just}$
18 return Just$_{set}$
```

Example: (ComputeAllJustifications)

Let us consider $\mathcal{K} = \{p \rightarrow q, q \rightarrow w, p \rightarrow w, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$. Our query is $p \rightarrow w$.

(1) We begin by finding the root justification. This is done by calling *ComputeSingleJustification*($\mathcal{K}$, $p \rightarrow w$). Since $p \rightarrow w \in \mathcal{K}$, Root_Just = $p \rightarrow w$. We then initialise the root node by calling *CreateNode*($\mathcal{K}$, Root_Just) which takes in a justification and its associated knowledge base. We add this node to our queue Q. The root node of the tree is then $p \rightarrow w$.

(2) We enter our while loop and retrieve the root node from the queue. For line 7 and 8 we have $Node\_Just = p \rightarrow w$ and $Node\mathcal{K}B = \mathcal{K}$. Our for loop executes only once as we only have one formula in our justification. We now aim to find another justification for our query and we do this by removing $\delta = p \rightarrow w$ from $\mathcal{K}$ in line 11. $Smaller\mathcal{K} = \{p \rightarrow q, q \rightarrow w, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$. By calling *ComputeSingleJustification*() on this smaller knowledge base, we get $Smaller\mathcal{K}_{Just} = \{p \rightarrow q, q \rightarrow w\}$. We create a new node with this justification and the reduced knowledge base in line 12. We add the new node to the tree in line 13. Line 15 adds the new node to the queue.

(3) In the next iteration of the while loop, we retrieve the node we just added from the queue. $Node\_Just = \{p \rightarrow q, q \rightarrow w\}$ and $Node\mathcal{K}B = \{p \rightarrow q, q \rightarrow w, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$. Our for loop will execute twice. For the iteration where $\delta = p \rightarrow q$, $Smaller\mathcal{K} = \{q \rightarrow w, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$. There is a justification in this smaller knowledge base, therefore $Smaller\mathcal{K}_{Just} = \{p \rightarrow r, r \rightarrow w\}$. A new node is created which is added to the tree and queue.

(4) For the iteration where $\delta = q \rightarrow w$, $Smaller\mathcal{K} = \{p \rightarrow q, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$ and $Smaller\mathcal{K}_{Just} = \{p \rightarrow r, r \rightarrow w\}$. A new node is created and added to the tree and queue. Although this new node and the previous node have the same label, they are considered different.

(5) In the next iteration of the while loop, we retrieve the node we added in the first iteration of the previous for loop from the queue. $Node\_Just = \{p \rightarrow r, r \rightarrow w\}$. In our first for loop iteration, $\delta = p \rightarrow r$ and $Smaller\mathcal{K} = \{q \rightarrow w, q \rightarrow r, r \rightarrow w\}$. It is clear that there is no justification for our query in this smaller knowledge base, therefore we add an empty node to the tree. The same result is observed when $\delta = r \rightarrow w$, therefore another empty node is added to the tree.

(6) In the next iteration of the while loop, we retrieve the last non-empty node we added from the queue. $Node\_Just = \{p \rightarrow r, r \rightarrow w\}$ and $Node\mathcal{K}B = \{p \rightarrow q, q \rightarrow r, r \rightarrow w, p \rightarrow r\}$. In our first for loop iteration, $\delta = p \rightarrow r$ and $Smaller\mathcal{K} = \{p \rightarrow q, q \rightarrow r, r \rightarrow w\}$. There is a justification in this smaller knowledge base, therefore $Smaller\mathcal{K}_{Just} = \{p \rightarrow q, q \rightarrow r, r \rightarrow w\}$. A new node is created which is added to the tree and queue.

(7) For the next for loop iteration where $\delta = r \rightarrow w$, we have no justifications. For the remaining nodes in the queue, we find that there are no justifications.

5 DEFEASIBLE REASONING

5.1 The Need for Non-monotonic Reasoning

Classical reasoning has a property referred to as *monotonicity* which means that adding new information cannot cause the retraction of previously drawn conclusions. Monotonic reasoning restricts the expressivity of classical logics such as Propositional Logic because these logics cannot handle exceptions. The classic example used to illustrate this concept is that of reasoning that a bird can fly [12]. Birds typically can fly, but there are exceptions such as penguins. Let us consider a knowledge base $\mathcal{K} = \{birds \rightarrow fly, penguins \rightarrow birds\}$. It is clear that $\mathcal{K} \models penguins \rightarrow fly$. If we then add the statement ($penguins \rightarrow \neg\, fly$) to $\mathcal{K}$, we cannot retract our conclusion. This is a significant shortfall of monotonic reasoning because there are exceptions to almost all statements and rules about the world. To cover this shortfall, experts formalised *non-monotonic* reasoning, which gives us the ability to retract previously drawn conclusions upon gaining new knowledge [7, 8]. *Defeasible reasoning* is an implementation of non-monotonic reasoning and is the basis of the paper written by Hamayobe [4].

Hamayobe's work is based on the *KLM Framework*, which is used to perform defeasible reasoning. The framework extends Propositional Logic by adding a binary connective to represent *defeasible implication*. The symbol for defeasible implication is $\mid\!\sim$. The formula $p \mid\!\sim q$ is read "p typically implies q". A set of formulas that contain a defeasible implication is referred to as a *defeasible knowledge base*. The classical statement $p \rightarrow q$ is the *materialisation* of $p \mid\!\sim q$.

5.2 Classical Justification Algorithm for Defeasible Reasoning

The Classical Justification Algorithm is used in the research project done by Hamayobe. His work is based on explanations, but for defeasible reasoning, which is non-monotonic. The process undertaken to find justifications for defeasible knowledge bases involves firstly methodically discarding statements from the knowledge base that cause a contradiction that renders a query false. If the remaining statements entail the query, then any defeasible statements are converted to classical statements. The Classical Justification Algorithm is then used as a sub-algorithm to find the justifications for the given knowledge base and query. Before the justifications are returned, any statement that was a defeasible statement and was converted to classical is converted back to defeasible in a process called *dematerialisation.*

The Classical Justification Algorithm needs to be used to compute justifications for defeasible knowledge bases. The integration of this algorithm with the work done by Hamayobe is what has allowed us to be able to compute justifications not just for classical knowledge bases but also for defeasible knowledge bases.

6 IMPLEMENTATION OF CLASSICAL JUSTIFICATION ALGORITHM

6.1 Overview

We now present the practical work done for this project. We implemented the Classical Justification Algorithm as a tool called ClassicalJustificationTool. In the following subsections we will highlight the Software Architecture (Section 6.2), External Packages used (Section 6.3), the Input and Output Parameters with excerpts (Section 6.4), the Algorithm Implementation (Section 6.5) and finally the Test Design, Execution and Evaluation (Section 6.6).

6.2 Software Architecture

The **ClassicalJustificationTool** was implemented using the Java Programming Language. We implemented our software using the

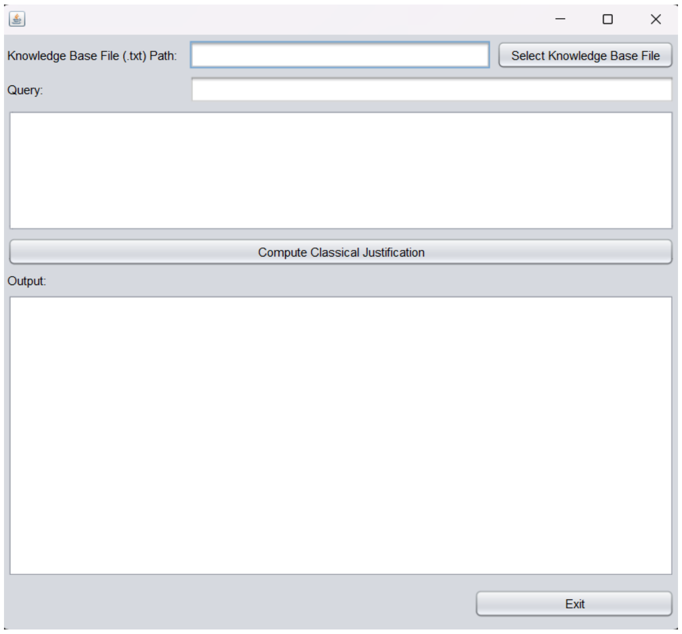
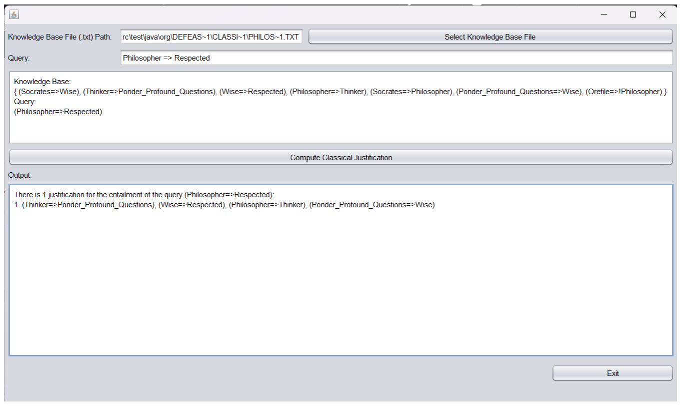

Knowledge Base File (.txt) Path:
Select Knowledge Base File
Query:
Compute Classical Justification
Output:
Exit

Figure 4: Graphical User Interface

Table 1: Symbol Representation

Symbols in the literature	Symbols in the tool
$\neg$	!
$\vee$	||
$\wedge$	&&
$\rightarrow$	=>
$\leftrightarrow$	<=>



```
Knowledge Base File (.txt) Path: rc\test\java\org\DEFEAS~1\CLASSI~1\PHILOS~1.TXT
Query: Philosopher => Respected

Knowledge Base:
{ (Socrates=>Wise), (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Socrates=>Philosopher), (Ponder_Profound_Questions=>Wise), (Orefile=>!Philosopher) }
Query:
(Philosopher=>Respected)

Output:
There is 1 justification for the entailment of the query (Philosopher=>Respected):
1. (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Ponder_Profound_Questions=>Wise)
```

Figure 5: Example input and output of our Compute All Justifications tool

Model-View-Controller design pattern. The Model component consists of the propositional formulas that form the basis of our operation, objects that we created to return values as well as our Hitting Set Tree which is used in the *ComputeAllJustifications* algorithm. The View component consists of a graphical user interface that allows users to interact with the tool. Finally, the Controller component consists of the implementation of the algorithms that we have theoretically presented in this paper.

The external packages used are written in Java therefore this was the main motivator behind the language choice. Using Java allowed us to leverage Object Oriented Programming in constructing our Hitting Set Tree and creating objects that represent our propositional formulas.

The **ClassicalJustificationTool** GUI is shown in Figure 4. The user is expected to interact with the tool in the following manner:

- The user selects their knowledge base by clicking the *Select Knowledge Base File* button.
- The user is then presented with a file chooser window where they then select a .txt file with the knowledge base of their choice.
- The knowledge base is then printed out on the text area.
- The user can then enter their query in the provided text field.
- The user can then click the *Compute Classical Justification* button to run the Classical Justifications Algorithm.
- The justification(s) for the given query is then printed out in the text area at the bottom of the screen.
- The user can terminate the program by clicking the *Exit* button.

6.3 External Packages

The **ClassicalJustificationTool** utilises two external packages. The first is the Tweety Project, a library from which we got the models and functions for the logic in our project - Propositional Logic [2]. The symbols that are used to represent logic operations in the literature are difficult to use practically, therefore we had to replace them accordingly as shown in Table 1.

The second external package is the SAT4J SAT solver, which is the black-box reasoner used in our project to compute classical entailments [1].

6.4 Input and Output Parameters

In this subsection we showcase the input and output needed by the **ClassicalJustificationTool**. The tool takes in two inputs: a classical knowledge base and a query. We let $\mathcal{K} = \{philosopher \rightarrow thinker, Aristotle \rightarrow philosopher, thinker \rightarrow ponder\_profound\_questions, ponder\_profound\_questions \rightarrow wise, Socrates \rightarrow wise, wise \rightarrow respected\}$. The query is entered into a textbox as a string. We let our query string be “philosopher => respected”. The result of these two inputs is then displayed in the text area to the user. Figure 5 shows the GUI display of the described scenario.

6.5 Algorithm Implementation

We implemented the functionality of computing a single justifications by following Horridge’s algorithms for expanding and contracting formulas (algorithms 1 to 4) [5]. To implement the functionality of computing all justifications, we needed to create an object that represents the Hitting Set Tree structure. We used Java’s Object Oriented Programming basis to create this binary tree object.

Each node stores a classical justification for the given query and a knowledge base from which that justification was extracted. The root node represents the first justification found using the *ComputeSingleJustification* algorithm on the full knowledge base. The child nodes have a knowledge base with one less propositional formula than their parent. We can think of the edge between a parent and each child as representing the formula that was removed to get the child node's knowledge base. We implemented the Classical Justifications Algorithm by using open-source code provided by Wang [14].

6.6 Test Design, Execution and Evaluation

The **ClassicalJustificationTool** was tested using a series of test knowledge bases and queries. Among those used were the examples included in Algorithm 1: *ExpandFormulas*, Algorithm 2 and 3: *ContractFormulas* and Algorithm 5: *ComputeAllJustifications*. The results of the tests were accurate and in accordance to the theory. We noted that there is a potential "blow up" that can occur when using the *ExpandFormulas* algorithm. This is to say that in order to find a justification, all the formulas in the knowledge base can end up being considered. This would significantly decrease the speed of execution of the **ClassicalJustificationTool** for large knowledge bases, therefore we limited our knowledge base size to an upper limit of ten.

These tests were designed to assess the correctness of each algorithm's implementation under various scenarios, providing a robust validation process for our work.

6.6.1 Unit Testing Approach

Due to the modular design of the **ClassicalJustificationTool** we were able to make use of unit tests to test each of the algorithms. To ensure the correctness of our implemented algorithms, comprehensive testing was carried out by means of Java unit tests. Each algorithm was encapsulated within its own test suite, allowing us to focus on specific functionalities and evaluate their outcomes independently. By employing unit tests, we aimed to detect and rectify any potential errors at an early stage of development.

6.6.2 Test Case Diversity

To rigorously evaluate the algorithms, we carefully designed a diverse set of test cases that covered different input variations and edge cases. This approach allowed us to examine the behavior of our algorithms across a wide range of scenarios, ensuring that they could handle various input sizes, constraints, and special conditions. By addressing both common and exceptional cases, we increased the confidence in the correctness and robustness of our implementations. For all cases where there was no justification, our algorithms were correct. For the cases where there were multiple justifications, our algorithms were able to compute all the justifications proving our Hitting Set Tree implementation was done correctly. For cases where invalid symbols were in our knowledge base or query, an appropriate error message was returned to the user. Our Appendix section highlights these test cases along with the respective GUI outputs in Figures 6, 7 and 8.

6.6.3 Validation Metrics

The unit tests were designed to verify correctness and not necessarily to quantify algorithm performance. We also enlisted the expertise of our supervisor to test the correctness and practicality of our tool and its interface. Our supervisor approved the correctness of our tool as well as its practicality in the field of *KRR*. The results of the test cases were correct and in accordance with the theory. We noted, however, that there is a potential "blow up" that can occur when using the *ExpandFormulas* algorithm. This is to say that in order to find a justification, all the formulas in the knowledge base can end up being considered. This would significantly decrease the speed of execution of the **ClassicalJustificationTool** for large knowledge bases, therefore we limited our knowledge base size to an upper limit of ten.

7 CONCLUSIONS

Explanation services help users understand how reasoning systems reach a conclusion. This is useful as it makes the black-box reasoning process transparent and thus allows for knowledge base comprehension and even debugging by experts. In this paper we have presented the Classical Justification Algorithm which explains why an entailment holds for a given knowledge base query. The algorithm utilises an expand-contract strategy proposed by Horridge in order to find the minimal sets of propositional formulas that entail a query.

For the theoretical part of this paper we presented an algorithm that computes justifications for the classical logic, propositional logic. This algorithm comprises of the sub-algorithm *ComputeSingleJustification* which uses the *ExpandFormulas* and *ContractFormulas* sub-algorithm to return one justification. We use the *ComputeAllJustifications* sub-algorithm to return all justifications of an entailment through the use of Reiter's Hitting Set Tree.

For the practical part of this paper, we implemented the theorised algorithms in Java. We used two external packages, The Tweety Project and the SAT4J SAT Solver which we used as our reasoner. The tool created follows an MVC architecture and allows user interaction through a Graphical User Interface (GUI). Furthermore, we were able to combine our work with that done by Hamayobe to allow for the computing of defeasible justifications.

We tested our tool with various test cases and expert testing by our supervisor. We uncovered a substantial time-explosion when using large knowledge bases to find justifications. This prompted us to limit the size of our test knowledge bases to $n = 10$.

8 FUTURE WORK

The output that is presented to the user has the syntax of propositional logic. This means that the potential users of this work are limited to those who are familiar to prompositional logic and other forms of formal logic. A future adaptation of this work could seek to present justifications using natural language and thus eliminating all symbols that are unique to formal logic.

Future work could also include listing justifications in a logical way mirroring how a human would reach a conclusion given a set of premises. This would involve identifying which deductive inference rules were used to reach a conclusion and then determining the dependencies between the premises accordingly, perhaps using graphs.

Future work could also involve optimising the process of computing justifications. This could include an investigation into whether pre-sorting formulas in the knowledge base could aid in streamlining the expand and contract process.

9 ACKNOWLEDGEMENTS

I would like to say a huge thank you to my supervisor, Professor Tommie Meyer, for introducing me to the world of Knowledge Representation and Reasoning and for guiding me through this project. Thank you to my project partner Chipo Hamayobe for the incredible support throughout this project.

References

[1] [n. d.]. *SAT4J SAT SOLVER.* http://www.sat4j.org/ Last Accessed: August 26, 2023.
[2] [n. d.]. *The Tweety Project.* https://tweetyproject.org/ Last Accessed: August 26, 2023.
[3] Mordechai Ben-Ari. 2012. *Propositional Logic: Formulas, Models, Tableaux.* Springer London, London, 7–47. https://doi.org/10.1007/978-1-4471-4129-7_2
[4] Chipo Hamayobe. 2023. Defeasible Entailment and Explanations.
[5] Matthew Horridge. 2011. *Justification based explanation in ontologies.* Ph. D. Dissertation. University of Manchester UK.
[6] Kevin C. Klement. 2004. Propositional Logic. In *Internet Encyclopedia of Philosophy.*
[7] John McCarthy. 1980. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence* 13, 1-2 (1980), 27–39.
[8] Drew McDermott and Jon Doyle. 1980. Non-monotonic logic I. *Artificial intelligence* 13, 1-2 (1980), 41–72.
[9] Raymond Reiter. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1 (1987), 57–95. https://doi.org/10.1016/0004-3702(87)90062-2
[10] Patrick Rodler and Manuel Herold. 2018. StaticHS: A Variant of Reiter's Hitting Set Tree for Efficient Sequential Diagnosis. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 9. 72–80.
[11] Peter Smith. 2003. *An introduction to formal logic.* Cambridge University Press.
[12] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. 2008. *Handbook of knowledge representation.* Elsevier.
[13] Warren J von Eschenbach. 2021. Transparency and the black box problem: Why we do not trust AI. *Philosophy & Technology* 34, 4 (2021), 1607–1622.
[14] S Wang. 2022. *Defeasible Justification for the KLM Framework.* Master's thesis.

A SUPPLEMENTARY INFORMATION

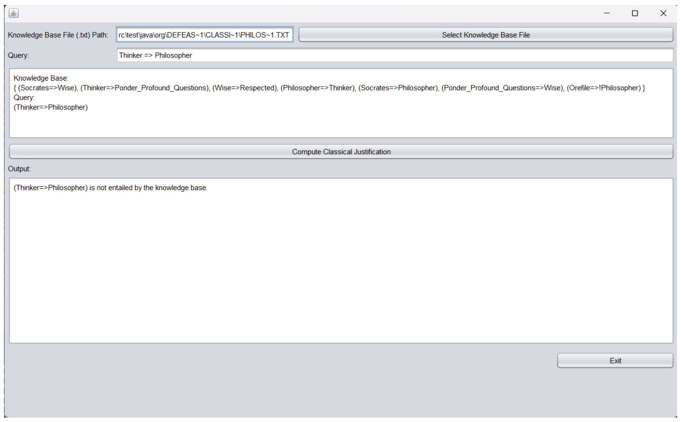
A.1 Test Cases

We present three representative test cases. The first shows a scenario where the query is not entailed by the knowledge base, the second shows a scenario where there is only one justification for the entailment, and the third shows where there are multiple justifications for the entailment.

For all our test cases, we let $\mathcal{K}$ = {*philosopher* $\longrightarrow$ *thinker*, *Aristotle* $\longrightarrow$ *philosopher*, *thinker* $\longrightarrow$ *ponder_profound_questions*, *ponder_profound_questions* $\longrightarrow$ *wise*, *Socrates* $\longrightarrow$ *wise*, *wise* $\longrightarrow$ *respected*, *Orefile* $\longrightarrow$ *!philosopher*}. The query is entered into a textbox as a string.

A.1.1 No entailment

The first test case shows a scenario where there is no entailment. The query entered is "thinker => philosopher". This conclusion cannot be inferred from our knowledge base, so as a result there can be no justifications. Figure 6 shows the output displayed to the user upon trying to compute a justification based on the given query and knowledge base.

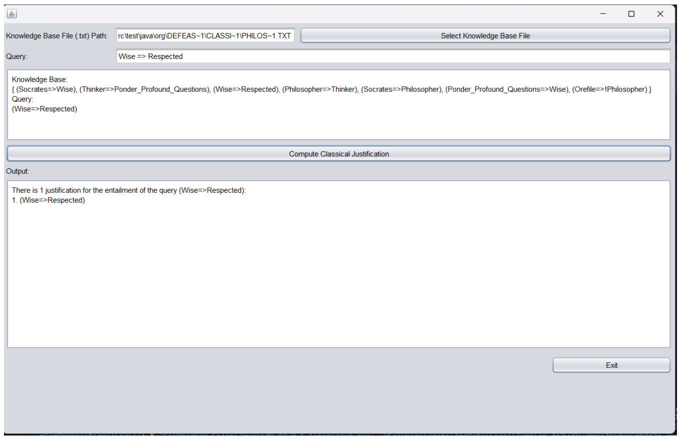


```
Knowledge Base File (.txt) Path: rc\test\java\org\DEFEAS~1\CLASSI~1\PHILOS~1.TXT
Query: Thinker => Philosopher

Knowledge Base:
{ (Socrates=>Wise), (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Socrates=>Philosopher), (Ponder_Profound_Questions=>Wise), (Orefile=>!Philosopher) }
Query:
(Thinker=>Philosopher)

Output:
(Thinker=>Philosopher) is not entailed by the knowledge base.
```

Figure 6: Example output of no entailment

A.1.2 One justifications

The second test case shows a scenario where there is an entailment and there is only one justification for it. The query entered is "wise => respected". This conclusion can be inferred from our knowledge base and this is because this query is a statement within our knowledge base. Figure 7 shows the output displayed to the user upon trying to compute a justification based on the given query and knowledge base.

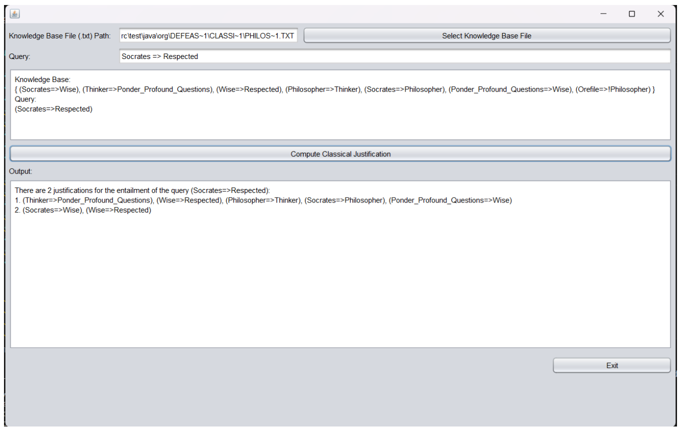


```
Knowledge Base File (.txt) Path: rc\test\java\org\DEFEAS~1\CLASSI~1\PHILOS~1.TXT
Query: Wise => Respected

Knowledge Base:
{ (Socrates=>Wise), (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Socrates=>Philosopher), (Ponder_Profound_Questions=>Wise), (Orefile=>!Philosopher) }
Query:
(Wise=>Respected)

Output:
There is 1 justification for the entailment of the query (Wise=>Respected):
1. (Wise=>Respected)
```

Figure 7: Example output of one justification

A.1.3 Multiple justifications

The third test case shows a scenario where there is an entailment and there is more than one justifications for it. The query entered is "Socrates => respected". This conclusion can be inferred from our knowledge base and there are two justifications for why this is so. Figure 8 shows the output displayed to the user upon trying to compute a justification based on the given query and knowledge base.



```
Knowledge Base File (.txt) Path: rc\test\java\org\DEFEAS~1\CLASSI~1\PHILOS~1.TXT
Query: Socrates => Respected

Knowledge Base:
{ (Socrates=>Wise), (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Socrates=>Philosopher), (Ponder_Profound_Questions=>Wise), (Orefile=>!Philosopher) }
Query:
(Socrates=>Respected)

Output:
There are 2 justifications for the entailment of the query (Socrates=>Respected):
1. (Thinker=>Ponder_Profound_Questions), (Wise=>Respected), (Philosopher=>Thinker), (Socrates=>Philosopher), (Ponder_Profound_Questions=>Wise)
2. (Socrates=>Wise), (Wise=>Respected)
```

Figure 8: Example output of multiple justifications