



Towards Implementing Contrastive Explanations for Answer-Set Programs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Florian Mallinger, B.A., B.Sc.

Matrikelnummer 01046033

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

Mitwirkung: Dipl.-Ing. Tobias Geibinger, BSc

Wien, 27. November 2025

Florian Mallinger

Thomas Eiter



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Towards Implementing Contrastive Explanations for Answer-Set Programs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Florian Mallinger, B.A., B.Sc.

Registration Number 01046033

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

Assistance: Dipl.-Ing. Tobias Geibinger, BSc

Vienna, November 27, 2025

Florian Mallinger

Thomas Eiter

Erklärung zur Verfassung der Arbeit

Florian Mallinger, B.A., B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. November 2025

Florian Mallinger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First, I would like to thank my advisor Tobias Geibinger for his support throughout the writing process. His availability, guidance and feedback was crucial for this thesis. I truly appreciate the time he invested in our regular meetings.

I would also like to thank Thomas Eiter for his support and introducing me to the topic of answer-set programming in the first place. His lecture inspired me to seek a master's thesis in this field.

Last but not least, I want to thank my friends and family. Their unconditional support allowed me to keep studying through these years.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Antwortmengenprogrammierung (ASP) ist ein deklaratives logisches Programmierparadigma mit vielen Anwendungen in der Wissensrepräsentation und Künstlichen Intelligenz. Probleme wie, Terminplanung, Klassifikation, Planung oder Produktkonfiguration, werden in ASP als Regeln ausgedrückt. Deren Lösungen werden als Antwortmenge präsentiert, i.e. Mengen von Atomen. Eine besondere Schwierigkeit bei ASP besteht in der Fehlersuche und in der Nachvollziehbarkeit, also warum ein bestimmtes Atom in einer Antwortmenge enthalten ist oder nicht. Kontrastive Erklärungen versuchen zu erläutern, warum ein bestimmtes Ereignis im Gegensatz zu einem anderen aufgetreten ist. Eiter et al. [EGO23] haben kontrastive Erklärungen für ASP formalisiert. Diese Formalisierung bildet den theoretischen Rahmen für diese Masterarbeit. Das Hauptziel dieser Masterarbeit ist es, die praktische Umsetzbarkeit der Anwendung des theoretischen Ansatzes kontrastiver Erklärungen auf ASP zu bestimmen. Zu diesem Zweck wurde ein neues Programm entwickelt, bestehend aus einem naiven Grounder, einem Backend, das den von Eiter et al. gegebenen Definitionen folgt, und einer benutzerfreundlichen Schnittstelle. Das Programm wurde hinsichtlich seiner Leistung durch Tests an bekannten Problemen evaluiert. Die Benutzerfreundlichkeit wurde anhand von Fallstudien beurteilt, die die Benutzeroberfläche des Programms in zwei realistischen Szenarien demonstrierten. Die Ergebnisse der Experimente zeigen, dass der Ansatz bei kleinen bis mittelgroßen Problemen gut funktioniert.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Answer-set programming (ASP) is a declarative logic programming paradigm with many applications in knowledge representation and artificial intelligence. In ASP, problems such as scheduling, classification, planning, or product configuration are encoded as rules. Solutions to these problems are presented as answer sets. An important difficulty for ASP is debugging and understanding why a certain atom is or is not included in an answer set. Contrastive explanations try to explain why a certain event occurred in contrast to another. Eiter et al. [EGO23] formalized contrastive explanations for ASP. This formalization forms the theoretical framework for this Master's thesis. The main goal of this thesis is to determine the practical feasibility of applying the theoretical approach of contrastive explanations to ASP. To achieve this, a new tool was created, consisting of a naive grounder, a backend, which follows the definitions provided by Eiter et al., and a user-friendly interface. The tool was then evaluated regarding performance by testing it on well known problems. Furthermore, its user-friendliness was evaluated through case studies demonstrating the program's interface in two realistic scenarios. The results of the experiments show that the approach performs well for small to medium-sized problems.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	ix
Abstract	xi
Contents	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Structure	2
2 Theoretical Background	3
2.1 Answer-Set Programming	3
2.1.1 Syntax	3
2.1.2 Semantics	4
2.1.3 Solving Answer-Set Programs	5
2.1.4 Extensions	6
2.1.5 Complexity	8
2.1.6 Meta programming	9
2.2 Contrastive Explanations	12
2.3 Contrastive Explanations for Answer-set Programming	13
3 Conceptual Overview	17
3.1 Grounding	17
3.2 Counterfactual Account	17
3.3 Counterfactual Explanation and Contrastive Explanations	20
3.4 User Interface	22
4 Implementation	25
4.1 Internal Computation	25
4.1.1 Grounder	25
4.1.2 Counterfactual Account	29
4.1.3 Counterfactual Explanation	32
4.1.4 Contrastive Explanation	33
4.2 User Interface	34
	xiii

4.2.1	Workflow	34
	Adding the program	34
	Configuration	34
	Result	36
5	Evaluation	39
5.1	Performance	39
5.1.1	Testing Setup	39
5.1.2	n-Queens	40
5.1.3	Sudoku	43
5.1.4	Vertex Coloring	46
5.2	Case study	47
5.2.1	Crow-Magpie	48
5.2.2	3-Coloring	51
6	Related Work	55
6.1	Contrastive Explanations for Other Systems	55
6.2	Debugging and Explainability Tools for ASP	56
7	Conclusion and Future Work	57
	List of Figures	59
	List of Tables	61
	Bibliography	63

Introduction

In this chapter, we present the context, motivation and goal of this thesis. Then, we provide an overview of the structure of the thesis.

1.1 Motivation

Artificial intelligence (AI) is increasingly being integrated into various aspects of our daily lives. Due to the black-box nature of many AI systems, understanding their output can be challenging. Therefore, tools that explain how AI algorithms arrive at solutions are essential for comprehension and debugging.

Answer-Set Programming (ASP) is a declarative logic programming paradigm well-suited for knowledge representation and AI with diverse applications, including scheduling, classification, planning, and product configuration. In ASP, problems are encoded as rules, and an ASP solver either generates a feasible solution called an answer set or determines that no answer set exists. An answer set is a set of atoms that satisfies the given rules and fulfills in addition a minimality condition. Depending on the program's complexity, it can be difficult to understand why certain atoms are included or excluded from the answer set.

Contrastive explanations aim to clarify why an event occurred in contrast to another, in our case, why a specific atom is true instead of another atom. Instead of only providing a single explanation for an event P , humans often have an alternative Q in mind that they believe should have occurred instead [Mil19]. Thus, an explanation often takes the form of “Why P rather than Q ?” Here, P is the *fact* or *explanandum*, and Q is the *foil*. A contrastive explanation highlights the differences in the reasons for P and Q and ignores their overlaps as the shared reasons are usually clear to the explainee and do not aid in the explanation. For example, if a classifier predicts bird labels, but shows *crow* instead of *maggie* by mistake, the user does not need to be told it's a bird.

Eiter et al. [EGO23] formalized contrastive explanations for ASP using sets of rules to explain both the explanandum and foil. This formalization will form the theoretical framework for this thesis.

Current state-of-the-art ASP tools for debugging and explainability often only justify the explanandum without considering any foil. Therefore, implementing contrastive explanations for general answer-set programs is necessary to provide more intuitive and understandable explanations.

The main goal of this thesis is to determine the practical feasibility of applying the theoretical approach of contrastive explanations to ASP, as presented in Eiter et al. [EGO23]. We apply contrastive explanations in the context of ASP to facilitate the debugging, explainability, and trustworthiness of (potentially black box) ASP. To achieve this, a new tool was created, designed specifically to apply contrastive explanations to ASP. This tool leverages the typical inputs and outputs of an ASP program, which are the answer set program (i.e., the rules) itself and the answer set (i.e., the set of atoms).

With this tool, we aim to bridge the gap between complex ASP systems and user understanding by integrating human feedback into the explanation process. This will not only make ASP more accessible to users but also increase their trust in the system by providing clear, understandable reasons for the outcomes generated by an ASP program. For the first time, we explicitly include user-provided explanandum (already part of the answer set) and foil (not part of the answer set but what the user would expect) examples and aim to output comprehensive explanations for both the provided parts.

Our main contributions are :

- The creation of a naive grounder suitable for generating contrastive explanations.
- The development of a new tool for producing contrastive explanations.
- The design of a dedicated user interface for the developed tool.

1.2 Structure

The structure of this Master's thesis is as follows. Chapter 2 provides the necessary theoretical background, including an overview of ASP and contrastive explanations. Chapter 3 presents an overview of the strategy and concepts used in the tool. Chapter 4 introduces the implementation of the tool in greater detail. This comprises a naive grounder, the implementation of contrastive explanations for ASP, as well as the frontend design of the tool. In Chapter 5, the tool is evaluated in terms of performance and usability. Chapter 6 presents related work regarding both contrastive explanations for other systems and other debugging or explainability tools for ASP. Finally, Chapter 7 offers concluding remarks, summarizes the key contributions of the thesis, discusses limitations and problems of the tool and provides an outlook with recommendations for future research.

Theoretical Background

In this chapter, the theoretical background of Answer-set Programming and contrastive explanations is introduced. Section 2.1 presents the basic terminology and concepts of Answer-set Programming that are necessary for the aim of this thesis. In Section 2.2 the idea of contrastive explanations is explained. Finally, Section 2.3 reviews the application of contrastive explanations to Answer-set Programming as presented by Eiter et al. [EGO23].

2.1 Answer-Set Programming

2.1.1 Syntax

The syntax of *Answer-set Programming (ASP)* [EIK09, GL88] is based on an alphabet $\mathcal{A} = (\mathbb{R}, \mathbb{V}, \mathbb{C})$, where \mathbb{R} is a set of *predicate symbols*, \mathbb{V} is a set of *variables*, and \mathbb{C} is a set of *constants*. \mathbb{C} is also called the *domain*. Each predicate of \mathbb{R} is associated with an *arity* $n \geq 0$, denoted by p/n for a predicate p with arity n . The elements of \mathbb{V} and \mathbb{C} are referred to as *terms*.

Atoms are of the form $p(t_1, \dots, t_n)$, where $p \in \mathbb{R}$ and t_i for $1 \leq i \leq n$ are terms. An atom signifies a special relationship between its terms, indicating that a particular statement about them holds true. In Answer-set Programming atoms, can be negated, denoted via *not* $p(t_1, \dots, t_n)$. This type of negation is called *default negation* or *negation-as-failure*, meaning $p(t_1, \dots, t_n)$ cannot be derived. A *literal* is an atom or a negated atom. The set of all atoms occurring in the rules of a program P is denoted by \mathcal{A}_P .

A disjunctive answer-set program consists of rules of the form:

$$a_1 \vee \dots \vee a_l \leftarrow b_1, \dots, b_m, \text{ not } c_1, \dots, \text{ not } c_n. \quad (2.1)$$

where a_i, b_j, c_k are atoms ($1 \leq i \leq l, 1 \leq j \leq m$ and $1 \leq k \leq n$). In practical answer-set program code the \leftarrow is denoted as “:-” and the \vee as “|”. The rule above in program code is written as:

$$a_1 \mid \dots \mid a_l :- b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n.$$

The intuitive meaning of a rule is that if right-side of the arrow is true, then also the left-hand side must be true. A rule is called *normal* if $l = 1$, *disjunctive* if $l \geq 2$ and *positive* if $n = 0$. If $m = 0$ and $n = 0$ the rule is referred to as a *fact*. Since the right-hand side is trivially true, the left-hand side must be true in any case. If $l = 0$ the rule is a constraint, the left-hand side cannot be true, hence the right-hand side must not be true either.

The set $H(r) = \{a_1, \dots, a_l\}$ is referred to as the head of a rule r . The set $B(r) = \{b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n\}$ is the body of a rule r , with $B^+(r) = \{b_1, \dots, b_m\}$ being the positive body and $B^-(r) = \{c_1, \dots, c_n\}$ being the negative body.

The *Herbrand universe* HU_P of a program P is defined as the set of all constants occurring in P . If there are no constants in P , HU_P consists only of an arbitrary constant. The *Herbrand base* HB_P of a program P denotes the set of all variable-free atoms over all predicates of P and all terms of HU_P .

A (negated) atom is called *ground*, if none of its terms are variables. A ground rule only consists of ground atoms. An answer-set program is referred to as ground, if all its rules are ground.

A ground *substitution* is a function $\sigma : \mathbb{V} \rightarrow \mathbb{C}$. *Grounding* is the process of substituting all variables of a rule or program with ground terms. The grounding of a rule r with regard to a set of constants C is defined as $gr(r, C) = \{r\sigma \mid \sigma : V \rightarrow C\}$ where V is the set of all variables in r . The grounding of a program P is then given by $gr(P) = \bigcup_{r \in P} gr(r, HU_P)$.

A special case in the grounding process is arithmetic, which is applicable when terms represent numbers. ASP system such as *clingo* [GKKS19, Lif19] allow *arithmetic expressions* and *arithmetic relations*. An arithmetic expression is defined as a term comprising a mathematical operator, selected from the set $\{+, -, *, /\}$, along with two operands, each of which may be a numeric value or another arithmetic expression. The operator is written in *infix notation*. During the grounding phase, such expressions are systematically evaluated and reduced to their corresponding numerical results.

An arithmetic relations in this context refers to atoms that represent the relation between two terms, which may be numbers or arithmetic expressions. A relation is expressed by one of these symbols $\{<, \leq, >, \geq, =, \neq\}$. Just as arithmetic expressions, arithmetic relations are also evaluated in the grounding step. An arithmetic relation that evaluates to true can be replaced by the constant true. Conversely, if the relation does not hold, it is substituted with the constant false.

2.1.2 Semantics

The semantics of answer-set programs is based on *interpretations*. An interpretation is a subset of the Herbrand base HB_P , i.e. a set of ground atoms. The body of a rule r is *satisfied* by an interpretation I iff $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$, denoted by $I \models B(r)$.

A ground rule r is satisfied by an interpretation I iff $H(r) \cap I \neq \emptyset$ whenever I satisfies the body of r . An interpretation I is called a *model* of a program P if it satisfies all ground rules of P , denoted by $I \models P$.

Given a program P and an interpretation I , the *reduct* P^I according to Wang et al. [WEZL23] is defined as: $P^I := \{H(r) \cap I \leftarrow B^+(r) \mid r \in gr(P), I \models B(r)\}$. Informally, the reduct P^I of an interpretation I is defined as the set of rules of P in a shortened form such that $I \models B(r)$. The head of the rules of P^I is reduced to the intersection of the head and I . The negative body is removed from these rules.

An interpretation I is an answer set of a program P if it is a minimal model of P^I , i.e., there is no $J \subset I$ such that J is a model of P^I .

All atoms of an answer set I need to be *supported* by the ground program P . An atom a is supported by a program P , if there is a rule r , such that I satisfied the body of r and $I \cap H(r) = \{a\}$.

Each atom a of an answer set I is satisfied by the reduct P^I , if $a \in I$, denoted by $P^I \models a$.

2.1.3 Solving Answer-Set Programs

Problems can be solved with answer-set programming using the following procedure [EIK09]:

1. *Encode* the problem instance I as an answer-set program P , such that the models of the program represent the solutions of I .
2. *Compute* one or all models M of P via an ASP solver.
3. *Extract* a solution for I from M .

The compute step usually starts with grounding the encoded problem. *Naive* grounding generates ground rules with all possible combinations of constants. The resulting program may have exponentially many more rules than the original non-ground program.

Example 1. Consider the answer-set program P :

```
pred(a) . pred(b) .
result(X_1, X_2, X_3) :- pred(X_1), pred(X_2), pred(X_3) .
```

P contains two constants, namely a and b . The naive grounding of P will result in the program with exponential size, since all combinations have to be generated.

```
pred(a) . pred(b) .
result(a, a, a) :- pred(a), pred(a), pred(a) .
result(a, a, b) :- pred(a), pred(a), pred(b) .
result(a, b, a) :- pred(a), pred(b), pred(a) .
result(a, b, b) :- pred(a), pred(b), pred(b) .
result(b, a, a) :- pred(b), pred(a), pred(a) .
```

```

result(b, a, b) :- pred(b), pred(a), pred(b).
result(b, b, a) :- pred(b), pred(b), pred(a).
result(b, b, b) :- pred(b), pred(b), pred(b).

```

However, in many cases a lot of these generated ground atoms can not be supported by the program and thus cannot be part of an answer set.

Example 2. The naive grounding of the program $P_2 = \{pred(a,b). result(X,Y) \leftarrow pred(X,Y).\}$ is:

```

pred(a,b).
result(a,a) :- pred(a,a).
result(a,b) :- pred(a,b).
result(b,a) :- pred(b,a).
result(b,b) :- pred(b,b).

```

Out of the last four rules only $result(a,b) :- pred(a,b).$ is relevant, since all other rules contain bodies with atoms that can never be supported, i.e. there is no way for the program to derive $pred(a,a)$, $pred(b,a)$ or $pred(b,b)$. This is due to the fact that none of these atoms appear in a head of a rule of P_2 .

Hence, unsupported atoms and all rules containing them in the positive body can be safely omitted.

The grounded program is then used by the actual ASP solver to find one, several or possibly all models. Modern ASP solver use conflict-driven search procedures [KLPS16], similar to SAT solving. A crucial difference to SAT solving is that all atoms of an answer set need to be supported, i.e. not only true but provably true.

2.1.4 Extensions

Answer-set programming allows for various extensions.

The definition of terms can be extended to include *functions* [CFG⁺20], which requires the addition of a set \mathbb{F} as function names to the alphabet. Functions are expressions of the form $f(t_1, \dots, t_n)$, where $f \in \mathbb{F}$ and t_i are terms.

A *conditional literal* [GKK⁺19, HLY14] has the form $L_0 : L_1, \dots, L_n$, where L_i are literals. L_1, \dots, L_n are called *conditions*. A conditional literal unfolds to ground atoms of L_0 such that the conditions L_1, \dots, L_n hold. Depending on whether a conditional literal appears in the head or body of a rule, it has a different meaning. If it is in the head, it will unfold to a disjunction of the ground atoms. If it appears in the body, it will unfold to a conjunction.

A similar concept to conditional literals are *aggregate elements* [CFG⁺20]. An aggregate element is denoted as

$$t_1, \dots, t_m : l_1, \dots, l_n$$

with t_1, \dots, t_m being terms and l_1, \dots, l_n are literals. Informally, the set $\{t_1, \dots, t_m : l_1, \dots, l_n\}$ in an interpretation I is the set of all terms t_1, \dots, t_m that satisfy the conjunction l_1, \dots, l_n in I .

Aggregate elements are used in *aggregate atoms* in the form

$$\#agg_r E \prec u$$

where u is a literal and E is a collection of aggregate elements, which are separated by “;”. $\#agg_r$ is a *aggregate function name*, i.e. an element of the set $\{\#count, \#sum, \#max, \#min\}$. $\prec \in \{<, \leq, =, \neq, >, \geq\}$ is an *aggregate relation*. An aggregate atom may also be written as $u \prec_1 \#agg_r E \prec_2 v$. An *aggregate function* represents the specific function applied to the collection of aggregate elements. An aggregate atom is true w.r.t. an interpretation I if the relationships $u \prec_1 \#agg_r E \prec_2 v$ hold.

A special usage of aggregate atoms are the *weight constraints* [GKK⁺19, SNS02], which assign weights to atoms. A weight constraint has the form

$$\#sum\{w_{a_1} : a_1; \dots; w_{a_n} : a_n, w_{b_1} : \text{not } b_1; \dots; w_{b_m} : \text{not } b_m\} \prec u$$

where a_i, b_j are atoms and w_k are the associated weights. Furthermore, u is a term called the *guard*. A weight constraint is satisfied by an interpretation I , if:

$$\sum_{a_i \in I} w_{a_i} + \sum_{b_j \notin I} w_{b_j} \prec u$$

Another important extension are *choice rules* [CFG⁺20, Sim99] of the form

$$\{a_1 : L_1; \dots; a_m : L_m\} \prec u \leftarrow b_1, \dots, b_n$$

Each $a_i : L_j$ represents a *choice element* where a_i is an atom and L_j is a list of literals. u is a term and b_1, \dots, b_n are literals. \prec is an *aggregate relation*. A choice rule is satisfied by an interpretation I iff a subset of $\{a_1, \dots, a_m\}$ is in I whenever I satisfies the body b_1, \dots, b_n and also the corresponding L_j literals. Hence, the rule is also satisfied if no element of the head is true.

Weak constraints [CFG⁺20, LPF⁺06, BLR00] have the form

$$:\sim b_1, \dots, b_n [w@l, t_1, \dots, t_m]$$

where b_1, \dots, b_n are literals and t_1, \dots, t_m are terms. w and l denote the weight and level, respectively. The terms t_1, \dots, t_m are used to define how many times the violation of a weak constraint is counted. Distinct tuples of those terms are counted separately. The standard arrow \leftarrow is replaced by $:\sim$ to avoid confusion. An answer-set of a program with weak constraints minimizes the sum of weights of the violated weak constraints of the highest level and among them those which minimize the sum of weights of the violated weak constraints in the next lower level, and so forth.

Heuristic-driven solving [GKK⁺19] for *clingo* is a way to modify the heuristic of an ASP solver in order to generate specific answer sets. The standard heuristic of *clingo* is *Variable State Independent Decaying Sum (VSIDS)* [MMZ⁺01]. VSIDS can be extended by the heuristic *Domain* to include domain-specific heuristics. This can be used to find subset minimal or subset maximal answer sets.

An heuristic can be implemented via the directive: `heuristic A : B. [w@p,m] A` is the atom that the heuristic addresses. `B` a rule body. If `B` is not satisfied, the heuristic is not applied. `w`, `p` and `m` are terms that denote the weight, priority and modifier respectively. `p` determines the priority of the heuristics for the same atom. `m` can take one of the values in `{sign, level, true, false, init, factor}`.

The `sign` modifier is used to prefer answer sets that contain the atom `A`, if `w` is positive, and answer sets without the atom `A`, if `w` is negative. `level` assigns a level to an atom that determines the order in which the atoms are decided. The modifiers `true` and `false` are a shortcut to assign a level with `w` and the sign to an atom. Naturally, the sign is positive if the modifier `true` is used, negative otherwise. `init` allows adding a value to an atom's *VSIDS heuristic score*, which also influences the decision order of atoms. The modifier `factor` can be used to multiply the *VSIDS heuristic score* of an atom.

In order to generate subset minimal answer sets in regard to a specific predicate `pred/3` the heuristic directive `#heuristic pred(_,_,_). [1, false]` can be used. The `false` modifier forces the solver to find answer sets that contain the least amount of `pred/3` atoms. Additionally, the command line options `--heuristic=Domain`, `--dom-mod=5,16` and `--enum-mod=domRec` need to be included when using *clingo*. `--heuristic=Domain` activates the *Domain* heuristic. `--dom-mod=5,16` defines the modifier as `false` and applies the modifier to all atoms that are shown. The command line option `--enum-mod=domRec` is needed to enable subset enumeration via domain-based recording.

The *interval* [Lif19] notation provides a convenient way of writing multiple atoms or terms that only differ in numerical values compactly. An interval has the form `n1..n2`, with `n1` and `n2` being integers and `n1 < n2`. It represents the range from `n1` to `n2`. The rule `p(1..3).` will be expanded to the rules `p(1).` `p(2).` `p(3)..` However, if the interval appears inside of a choice atom, the interval is unfolded inside the choice atom. E.g. The rule `{p(1..3)} > 1.` is equal to the rule `{p(1); p(2); p(3)} > 1..`

2.1.5 Complexity

Complexity classes are used to classify problems according to how much computational time and memory a machine needs to solve them in the worst case. A *Turing machine* [Tur37] is a theoretical machine that is capable of solving any computable problem. A Turing machine is the standard machine model to show the complexity of a problem. There are many different versions of Turing machines. For the complexity analysis of this thesis, the most relevant ones are *deterministic* and *non-deterministic* Turing machines.

As the name suggests, deterministic Turing machines strictly follow a unique execution order, while non-deterministic Turing machines may choose one of many execution orders.

Complexity classes organize problems by the time a Turing machine needs to solve them relative to input size. The complexity class P consists of all decision problems that can be solved by a deterministic Turing machine in polynomial time. NP is the class of all decision problems that can be solved by a non-deterministic Turing machine in polynomial time.

In order to classify problems further *oracle machines* are needed. Oracle machines are abstract machines for a specific complexity class C . They can solve all problems of C instantly. The class P^C denotes the class of problems that can be solved by a deterministic Turing machine with access to an oracle for the class C in polynomial time.

With oracle machines the *polynomial hierarchy* can be defined. The polynomial hierarchy consists of the complexity classes Σ_k^P , Π_k^P and Δ_k^P for $k \geq 0$.

$$\begin{aligned}\Sigma_0^P &= \Pi_0^P = \Delta_0^P = P \\ \Sigma_{k+1}^P &= NP^{\Sigma_k^P} \\ \Pi_{k+1}^P &= co-NP^{\Sigma_k^P} \\ \Delta_{k+1}^P &= P^{\Sigma_k^P}\end{aligned}$$

A problem A is called C -hard for a given complexity class C , if all problems of C can be reduced to A . If a problem A is C -hard and is also a member of C , it is called C -complete.

Deciding whether a certain atom occurs in an answer set of a given normal program is coNP-complete [DEGV01]. For disjunctive answer set programs the complexity rises to Π_2^P -complete [EG95].

Deciding whether a given normal program has some answer set is NP-complete. In case of disjunctive answer set programs the problem is in Σ_2^P -complete.

For a more details regarding complexity theory and the mentioned concepts, we refer to the standard textbook on the topic by Papadimitriou [Pap94].

2.1.6 Meta programming

Meta programming is a concept in computer science to treat programs as input data for other programs. This allows the programmer to analyze and manipulate programs automatically. Depending on the programming language, translating the program into data can be done in many different ways. In case of ASP, *reification* [KRSW23] can be used. Reification is the process of turning an answer-set program into a set of facts. The following procedure is based on the ASP system *clingo* [GKKS19] and *clingo*'s ASP intermediate format *aspif*[GKK⁺16].

Example 3. Consider the following answer-set program $P = \{\{a\}. b \leftarrow a. c \leftarrow \text{not } a. d.\}$. Listing 2.1 shows the usage of *clingo*'s reification function on the program P in Python.

2. THEORETICAL BACKGROUND

```
1 from clingox.reify import reify_program
2
3 prg = "{a}. b :- a. c :- not a. d."
4 symbols = reify_program(prg)
5 for s in symbols:
6     print(str(s))
```

Listing 2.1: Python program that reifies program P with clingo

```
1 output(a,2)
2 output(b,3)
3 output(c,4)
4 output(d,0)
5
6 rule(choice(1),normal(0))
7 atom_tuple(1)
8 atom_tuple(1,2)
9 literal_tuple(0)
10
11 rule(disjunction(3),normal(2))
12 atom_tuple(3)
13 atom_tuple(3,4)
14 literal_tuple(2)
15 literal_tuple(2,2)
16
17 rule(disjunction(2),normal(1))
18 atom_tuple(2)
19 atom_tuple(2,3)
20 literal_tuple(1)
21 literal_tuple(1,-2)
22
23 rule(disjunction(0),normal(0))
24 atom_tuple(0)
25 atom_tuple(0,1)
26
27 literal_tuple(3)
28 literal_tuple(3,4)
29
30 literal_tuple(4)
31 literal_tuple(4,3)
```

Listing 2.2: Formated output of the Python program in 2.1

The Python program in Listing 2.1 for Example 3 produces the output shown in Listing 2.2

In order to distinguish between the elements, different predicates are used. All atoms are assigned IDs via the predicate *output/2*, as shown in Lines 1-4 of Listing 2.2. Atoms get the ID 0, if *clingo* can infer that the atom has to be true.

Each rule can be identified by its head and body via the predicate *rule/2*. The first argument of *rule/2* indicates whether the head of the rule is a disjunction or choice with the functions *disjunction/1* and *choice/1* respectively. The choice rule $\{a\}$. is represented by Line 6 in Listing 2.2.

The second argument of `rule/2` represents the body of the rule via the functions `normal/1` and `sum/1`. The former denotes that the body is a set of literals without weights. The latter indicates that it the literals of the body are weighted.

The parameter of the four functions `disjunction/1`, `choice/1`, `normal/1` and `sum/1`, are unique IDs for heads and bodies.

Each ID of the head functions `disjunction/1`, `choice/1` are associated with the predicate `atom_tuple/1` with the ID as the argument. Similarly, the IDs of the body functions `normal/1` and `sum/1` have a corresponding `literal_tuple/1` predicate.

In order to assign atoms to heads and literals to bodies the same predicate symbols are used, but with a different arity. Hence, the predicate `atom_tuple/2` connects a head with its atoms via the two parameters, i.e. the first parameter is the ID of the head and the second parameter is a reference to the atom. The latter is not the ID of atom, but an integer that is linked to the atom ID via the predicate `literal_tuple/2`. Lines 11 to 15 in Listing 2.2 contain the information for the rule `b :- a.. atom_tuple(3,4)` links the head with ID 3 to the atom reference 4. This atom reference is connected to the atom ID of `b`, which is 3, by `literal_tuple(3,4)` of Line 28.

`literal_tuple/2` is also used to assign literals to bodies. This works in a similar way as `atom_tuple/2`. Negated literals are expressed by a negative literal reference in `literal_tuple/2`. E.g. the negated literal `not a` of `c:- not a.` is shown as `-2. literal_tuple(1,-2)` in Line 21 connects the body with ID 1 with the literal reference 2 as a negated atom. The literal reference 2 is linked to the literal ID 2 via `literal_tuple(2,2)` in Line 15.

The fact `d.` is represented by the Line 23. Since a fact does not have a body, there is no `literal_tuple(0,_)`. Both `d.` and `{a}.` have no bodies and thus share the same body ID 0, shown in Line 9.

The reified program can then be used to analyze the program, adapt it and by adding *meta encodings* [KRSW23] compute answer sets or other kinds of models with differing semantics.

Listing 2.3 shows a meta encoding that can be combined with a reified program. Both programs together can be solved, which will result in answer sets of the original program.

```

1 conjunction(B) :- literal_tuple(B),
2     hold(L) : literal_tuple(B, L), L > 0;
3     not hold(L) : literal_tuple(B, -L), L > 0.
4
5 body(normal(B)) :- rule(_, normal(B)), conjunction(B).
6 body(sum(B,G))  :- rule(_, sum(B,G)),
7     #sum { W,L :     hold(L), weighted_literal_tuple(B, L,W), L > 0 ;
8           W,L : not hold(L), weighted_literal_tuple(B, -L,W), L > 0 } >= G.
9
10 hold(A) : atom_tuple(H,A) :- rule(disjunction(H),B), body(B).
11 { hold(A) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B).
12
```

```
13 #show.  
14 #show T : output(T,B), conjunction(B).
```

Listing 2.3: Meta program to generate answer sets from a reified program

The predicate `hold(A)` in the meta encoding means that the atom with ID A is true. It can be derived by the rules on Line 10 and 11. The rule in Line 10 is used for disjunctive rules. Given a rule with body B , if the atom `body(B)` is true, then one element of the rules head must also be true. The atoms of the head H can be addressed by the predicate `atom_tuple(H,A).hold(A) : atom_tuple(H,A)` of Line 10 is a conditional literal, which unfolds to a disjunction. Line 11 is the analogous rule for choice rules.

The argument for the atom `body/1` is either `normal(B)` or `sum(B,G)`, which refers to a rule body that is weighted or not, as shown in Lines 5 to 8. If the rule is not weighted, the atom `body(B)` is true if there is a corresponding unweighted rule such that the body is satisfied. This is expressed by the atom `conjunction/1` on Lines 1 to 3. `hold(L) : literal_tuple(B, L)` on Line 2 is a conditional literal that expands to a conjunction of all atoms `hold(L)` such that `literal_tuple(B,L)` is true, i.e. literal with ID L is part of the body with ID B . Line 3 shows the complementary condition with negated literals.

If the argument for the atom `body/1` is `sum(B,G)`, then the rule with ID B is weighted. Similar to `conjunction`, Lines 6 to 8 contain the rule, which determines whether `body(sum(B,G))` needs to be true. The argument G refers to the lower bound of the weighted rule. the literal `weighted_literal_tuple(B,L,W)` is used to encode the weight W of literal L in body B . The first part of the symbolic set of Lines 7 and 8 unfolds to the set of all weights of the literals in the rule body B , if the corresponding literal holds. The second part of the symbolic set expands to the set of all weights of negated literals in the rule body B , if the corresponding literal does not hold. The aggregate function `#sum` of this symbolic set sums up all the weights in the set. The sum is then compared to the lower bound G and evaluates to true if it is equal or larger than G .

The last to Lines 13 and 14 of Listing 2.3 define the output of the program, via the `#show` statement.

2.2 Contrastive Explanations

Contrastive explanations aim to clarify why an event occurred in contrast to another. When asking for an explanation for an event P , humans often have an alternative Q in mind that they believe should have occurred instead [Mil19]. Thus, an explanation often takes the form of “Why P rather than Q ?”. Here, P is the *fact* or *explanandum*, and Q is the *foil* [Lip90]. Even if the foil isn’t directly mentioned, people often infer it from the context. When explaining something, people naturally assume what the listener might not know or understand. For the listener to learn effectively, the explainer needs to

understand the explainee’s expectations, i.e. what might cause the foil. A *counterfactual* refers to a hypothetical scenario that could lead to the foil occurring.

Although similar, there are key differences between the two questions “Why P rather than Q ?” and “Why P and not- Q ?”. Temple [Tem88] argued for the equivalence of both questions, but was refuted by Lipton [Lip90]. In Temple’s view, answering both questions required an explanation for P and an explanation for not- Q . According to Lipton this is not the case, since answering “Why P rather than Q ?” does not entail fully explaining P . Instead, it is necessary to highlight the different reasons for P and Q .

Hence, the explanation for the explanandum changes according to the foil. The intersection of the reasons for the explanandum and foil are clear to the explainee and should be omitted. Only the symmetric difference of the reasons between the explanandum and foil is new information for the explainee. The explainee is only interested in what exactly caused the situation to end up at the explanandum that is different from what may cause the foil.

Example 4. If a classifier predicts bird labels, but shows *crow* instead of *magpie* by mistake, a user may ask “Why does it say *crow* rather than *magpie*?”. Clearly, the user understands that the object in question is a bird, because he or she chose *magpie* as the foil. Hence, giving all the reasons for why this is a crow, i.e. there are feathers, a beak, etc., would be superfluous. Instead, everything that a crow and a magpie have in common may be omitted. The only relevant information is the difference between the two types of birds, namely the color of the feathers. The correct answer to the user’s question thus would be: “The animal was classified as a crow instead of a magpie, since it is a crow if it is a bird and has dark wings, and there are dark wings, whereas it would have been a magpie if it is a bird and has white wings, but there were dark wings.” [EGO23]

2.3 Contrastive Explanations for Answer-set Programming

Eiter et al. [EGO23] formalized contrastive explanation for ASP using sets of rules to explain both the explanandum and foil, respectively.

The first step to formulate contrastive explanations for answer-set programming is to define an *explanation frame*.

Definition 2.3.1 (Explanation Frame). An explanation frame is a tuple (P, S, A) , where (i) P is a program, (ii) $S \subseteq P$ is a set of fixed rules, and (iii) $A \subseteq A_P$ is a set of assumable atoms.

An explanation frame defines the context of the explanation process. P is the program in question that contains the set S which denotes the common knowledge, e.g. definitions and things the explainee firmly believes. Hence, S must not be changed or removed. A is a set of facts containing a subset of A_P that are not currently part of the program, but it is feasible that they were.

According to the explaine, an answer-set of P now seems to be erroneous and contains one or more atoms instead of a different ones. This can be formalized in an *contrastive explanation problem*.

Definition 2.3.2 (Contrastive Explanation Problem). A contrastive explanation (CE) problem for an explanation frame $\mathcal{F} = (P, S, A)$ is a tuple $\langle I, E, F \rangle$, where $I \in AS(P)$, $E \in I$ is the explanandum and $F \cap I = \emptyset$ is the foil.

The explaine was expecting to see the foil F in the answer-set I , but instead the explanandum E was present. In order to find the contrastive explanation, a *counterfactual account* is needed.

Definition 2.3.3 (Counterfactual Account). Let $\mathcal{F} = (P, S, A)$ be an explanation frame and $\mathcal{P} = \langle I, E, F \rangle$ be a CE problem for \mathcal{F} . Then, a program $P' \subseteq P$, an interpretation I' , and set $A' \subseteq A$ of assumptions are a counterfactual account (CA) for \mathcal{P} if

- (i) $S \subseteq P'$,
- (ii) $F \cap A' = I \cap A' = \emptyset$,
- (iii) $I' \in AS(P' \cup A')$,
- (iv) $F \subseteq I'$ and $E \not\subseteq I'$, and
- (v) P' is \subseteq -maximal, i.e., there is no $CA P'', I'', A''$ for \mathcal{P} where $P'' \supset P'$.

A counterfactual account depicts an alternative scenario where the explanandum does not occur, but the foil does. This account involves a modified answer-set program, P' , which closely resembles the original program P . However, unlike P , when combined with a set of assumptions A' , P' leads to the foil instead of the explanandum being true in an answer-set I' .

Naturally, the new program P' must also contain the common knowledge S . It is required that the foil F is not included in A' , since the goal is to find an explanation for F . Additionally, A' must not contain assumptions that were already present in the original answer-set I , since A' is a set of counterfactual assumptions. P' needs to be \subseteq -maximal, so the divergence to the original is minimal. This will ensure that the contrastive explanation will be as concise as possible.

Explanations for both the explanandum and foil can be extracted from the explanation frame, contrastive explanation problem and the counterfactual account via an *counterfactual explanation*.

Definition 2.3.4 (Counterfactual Explanation). Let $\mathcal{F} = (P, S, A)$ be an explanation frame and $\mathcal{P} = \langle I, E, F \rangle$ be a CE problem for \mathcal{F} . A counterfactual explanation for \mathcal{P} is any tuple $\langle Q_1, Q_2, Q_\Delta \rangle$ such that

- (i) there is some $CA P', I', A'$ for $\langle I, E, F \rangle$ under \mathcal{F} ,
- (ii) $Q_1 \subseteq P$ and $Q_1^I \models a$ for every $a \in E$,
- (iii) $Q_2 \subseteq P' \cup A'$ and $Q_2^I \models a$ for every $a \in F$,
- (iv) $Q_\Delta = P \setminus P'$,
- (v) Q_1 is lexicographic \subseteq -minimal w.r.t. (P', Q_Δ) satisfying (ii), and
- (vi) Q_2 \subseteq -minimal satisfying (iii).

A counterfactual explanation isolates only the essential rules required to derive the explanandum, Q_1 , and the foil, Q_2 . Here, Q_1 represents the minimal subset of P that leads to the explanandum, while Q_2 is the minimal subset of P' and A' that leads to the foil.

Furthermore, a counterfactual explanation includes all rules that must be excluded to derive the foil in the counterfactual account, i.e. Q_Δ . While a counterfactual explanation demonstrates how both the explanandum and foil can be derived, it may still contain extraneous information for the explainee. *Contrastive explanation* refines this by removing information that the explainee already knows from the counterfactual explanation.

Definition 2.3.5 (Contrastive Explanation). Given an explanation frame $\mathcal{F} = (P, S, A)$ and a CE problem $\mathcal{P} = \langle I, E, F \rangle$ for \mathcal{F} , a tuple $\langle C_1, C_2, C_\Delta \rangle$ is a contrastive explanation (CE) for \mathcal{P} if there is a counterfactual explanation $\langle Q_1, Q_2, Q_\Delta \rangle$ for \mathcal{P} such that (i) $C_1 = Q_1 \setminus (Q_2 \cup S)$, (ii) $C_2 = Q_2 \setminus (Q_1 \cup S)$, and (iii) $C_\Delta = Q_\Delta \setminus S$.

Since the explainee expected the foil to be the case, all rules that Q_1 and Q_2 have in common contain no new information for the explainee. Also, including rules of the set S does not help the explainee. The contrastive explanation gives concise explanations for the explanandum and foil by leaving out unnecessary information according to Lipton's [Lip90] concept. In particular, the causal explanation for the explanandum C_1 consists of all rules of Q_1 , excluding those of Q_2 and S . The causal explanation for the foil works the same.

Conceptual Overview

In this chapter the basic strategy for generating contrastive explanations is shown. This includes the steps and concepts necessary for finding counterfactual accounts, as well as how both counterfactual and contrastive explanations can be extracted from them.

3.1 Grounding

Before counterfactual accounts can be found, the input program needs to be ground. However, state-of-the-art grounders are unsuitable for this task, since they remove or modify rules that do not affect the answer set for the program. Counterfactual accounts and thereby also contrastive explanations may still rely on these rules. By adding assumptions or excluding certain rules, previously irrelevant rules can become satisfiable and essential for deriving the foil. That is why a new naive grounder was designed, which generates all possible ground rules.

The grounding process can be divided into two phases. First, all constants are extracted from the set of rules. In the second phase, each rule is instantiated by systematically substituting its variables with every possible combination of the extracted constants.

After the grounding step, the resulting program can then be used to generate counterfactual accounts.

3.2 Counterfactual Account

Figure 3.1 depicts the flowchart of the procedure to generate counterfactual accounts.

A counterfactual account represents an alternative scenario where the explanandum, i.e. the atom of the answer set which we seek to justify, does not occur, but the foil does. The foil is the atom which was expected instead of the explanandum. In order to find

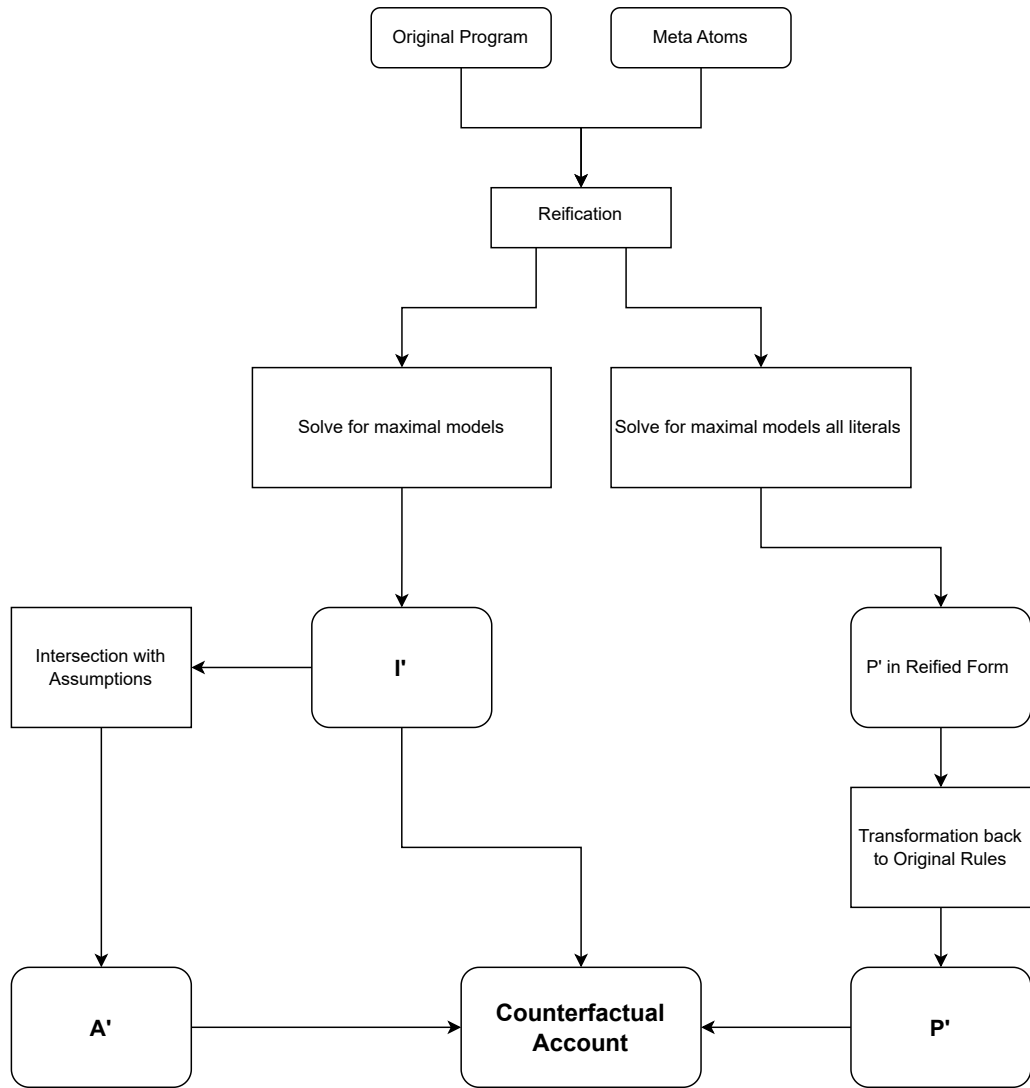


Figure 3.1: Flowchart of the procedure to generate Counterfactual Accounts

such an account it is necessary to remove certain rules that force the explanandum to be part of the answer set. Furthermore, the foil needs to be part of the answer set of the program that represents the new scenario. This may require additional facts, i.e. assumptions, to be the case. The counterfactual account can be found with the help of reification and meta programming.

Similar to the grounding step, the reification has to be performed by a custom made procedure. State-of-the-art reification could remove or modify rules that are needed in

their original form in order to generate counterfactual accounts, which are informative for contrastive explanations.

As a first step, the grounded program together with the *meta atoms* are reified. Meta atoms are those atoms that encode the explanandum, foil and assumptions which are specified by the user. The reified program, as a meta representation of the original program, can be used to omit certain rules that prohibit the derivation of the foil or rules that enforce the explanandum to be part of the answer set. This is done by representing rules that are not fixed as choice rules. Moreover, the integration of auxiliary rules tailored to the meta atoms makes it possible to enforce specific constraints at the meta-level, i.e. the foil must be present in the resulting answer set, while the explanandum is explicitly excluded. Assumptions are included via choice rules. They will only be included in the answer set, if the derivation of the foil requires them.

Solving this reified program requires additional rules and a special heuristic.

The reified program, together with auxiliary rules for the meta atoms and the meta program, shown in Listing 2.3, has to be solved twice. In both cases heuristic-driven solving for subset-maximal answer sets is applied in order to obtain all possible counterfactual accounts. The first solving step yields the answer set I' , which specifies the atoms that hold in the counterfactual account. Here, the meta program remains unchanged. In the second run, it is also necessary to determine which rules of the original program contributed to the creation of I' and which ones were omitted. This can be done by modifying the meta program. The result of this is P' , but still in reified form. After transforming these rules back to the original form, we receive P' .

The set of assumptions A' represents the subset of the original assumptions A that was needed to derive the foil. It can simply be extracted from I' by intersecting it with A .

Example 5. Considering the Example 4, the input for the program needs to be:

- $P = \{\text{crow} :- \text{bird}, \text{darkwings}.$
 $\text{bird} :- \text{feathers}, \text{beak}, \text{shape}.$
 $\text{magpie} :- \text{bird}, \text{whitewings}.$
 $\text{beak. shape. feathers. darkwings.}\}$
- $S = \{\text{crow} :- \text{bird}, \text{darkwings}.$
 $\text{bird} :- \text{feathers}, \text{beak}, \text{shape}.$
 $\text{magpie} :- \text{bird}, \text{whitewings}.$
 $\text{beak. shape. feathers.}\}$
- $A = \{\text{whitewings}\}$
- $I = \{\text{crow}, \text{bird}, \text{feathers}, \text{beak}, \text{shape}, \text{darkwings}\}$
- $E = \{\text{crow}\}$
- $F = \{\text{magpie}\}$

From this explanation frame (P, S, A) and contrastive explanation problem (I, E, F) one counterfactual account (P', I', A') can be generated. P' consists of all rules of P except for `darkwings.`, i.e. `bird :- feathers, beak, shape. feathers. beak. shape. crow :- bird, darkwings. magpie :- bird, whitewings.` The inclusion of `darkwings` would result in the derivation of the explanandum `crow`. Hence, P' is \subseteq -maximal.

A' contains the atom `whitewings`. The only way to derive the foil `magpie` is through the rule `magpie :- bird, whitewings.` Since the facts `feathers.`, `shape.`, `beak.` together with the rule `bird :- feathers, beak, shape.` are part of P' , the atom `bird` is also part of the answer set I' . The atom `whitewings` does not appear in P . Therefore, it needs to be included as an assumption of the set A to derive `magpie`.

The model I' consists of the following atoms: `beak, shape, bird, whitewings, magpie, feathers.` I' includes the foil `magpie` but not the explanandum `crow`.

Once the counterfactual accounts have been found, the crucial rules for deriving the explanandum and foil have to be filtered out, as counterfactual explanations.

3.3 Counterfactual Explanation and Contrastive Explanations

A counterfactual explanation identifies only the essential rules needed to derive the explanandum Q_1 and the foil Q_2 . Specifically, Q_1 corresponds to the minimal subset of P that entails the explanandum, whereas Q_2 corresponds to the minimal subset of P' together with A' that entails the foil. In addition to the two sets Q_1 and Q_2 , the counterfactual explanation also includes a third component, denoted by Q_Δ . Specifically, Q_Δ is defined as the collection of all rules that must be removed from the original program P in order to render the derivation of the foil possible. In other words, Q_Δ consists precisely of those rules that are present in P but absent in P' .

This can be achieved by applying heuristic-driven solving again. In contrast to the maximal subset heuristic employed for constructing counterfactual accounts, we adopt a subset-minimal heuristic in this context. In order to filter out only those rules that are relevant for deriving the explanandum, or foil respectively, additional auxiliary atoms are added to each rule. For each of these atoms, a corresponding choice rule is appended to the program. The atoms of all rules needed for the derivation of the explanandum or foil can then be set to true, while all others are set to false. The minimal subset of rules needed to derive the explanandum, or foil respectively, can then be extracted from the answer set of this program. These rules correspond to the auxiliary atoms present in the answer set.

This procedure is first applied to the program P , which yields the set Q_1 , i.e. the rules to derive the explanandum. In a second step the same procedure is applied to the program P' together with the set of assumptions A' . From this we obtain the set Q_2 ,

which represent the rules required to derive the foil. Additionally, the counterfactual explanation also contains the set Q_Δ , which represents all the rules that needed to be removed from P to be able to derive the foil. Thus, the set Q_Δ contains all those rules of P that are not present in P' .

Example 6. The counterfactual explanation $\langle Q_1, Q_2, Q_\Delta \rangle$ for the counterfactual account of Example 5 is as follows:

Q_1 consists only of those rules of P that are needed to derive `crow`, i.e.:

```
feathers.
beak.
shape.
bird :- feathers, beak, shape.
crow :- bird, darkwings.
darkwings.
```

Notably, the rule `maggie :- bird, whitewings.` has been removed, as it has no impact on the derivation of `crow`.

Q_2 contains all the rules of P' and A' required to derive the foil `maggie`, i.e.:

```
feathers.
beak.
shape.
bird :- feathers, beak, shape.
maggie :- bird, whitewings.
whitewings.
```

Here, similar to Q_1 , only the rule `crow :- bird, darkwings.` was removed from P' in Q_2 .

Q_Δ consists only of the rule `darkwings.`, because it needed to be removed from P to derive the foil.

A contrastive explanation removes superfluous information of a counterfactual explanation. The set C_1 narrows Q_1 to rules that aid the explainee. Set S , representing common knowledge, can be removed as the explainee is certain that it is true. Rules in both Q_1 and Q_2 are also excluded, as the explainee assumes their intersection.

The set C_2 is defined as Q_2 without the rules in S or Q_1 . C_Δ consists of Q_Δ 's rules excluding those in S .

Example 7. Continuing the Example of 5 and 6, the contrastive explanation $\langle C_1, C_2, C_\Delta \rangle$ is $\{\{\text{darkwings.}\}, \{\text{whitewings.}\}, \{\text{darkwings.}\}\}$. The only relevant information that explains the derivation of the explanandum `crow` is the rule `darkwings.`. Hence, C_1 is simply the set $\{\text{darkwings.}\}$.

Likewise, the addition of the rule `whitewings.` would allow for the derivation of the foil `maggie` in the counterfactual case. C_2 only contains the the rule `whitewings..` Since Q_Δ does not contain any rule that is an element of the set S , C_Δ simply remains `darkwings..`

3.4 User Interface

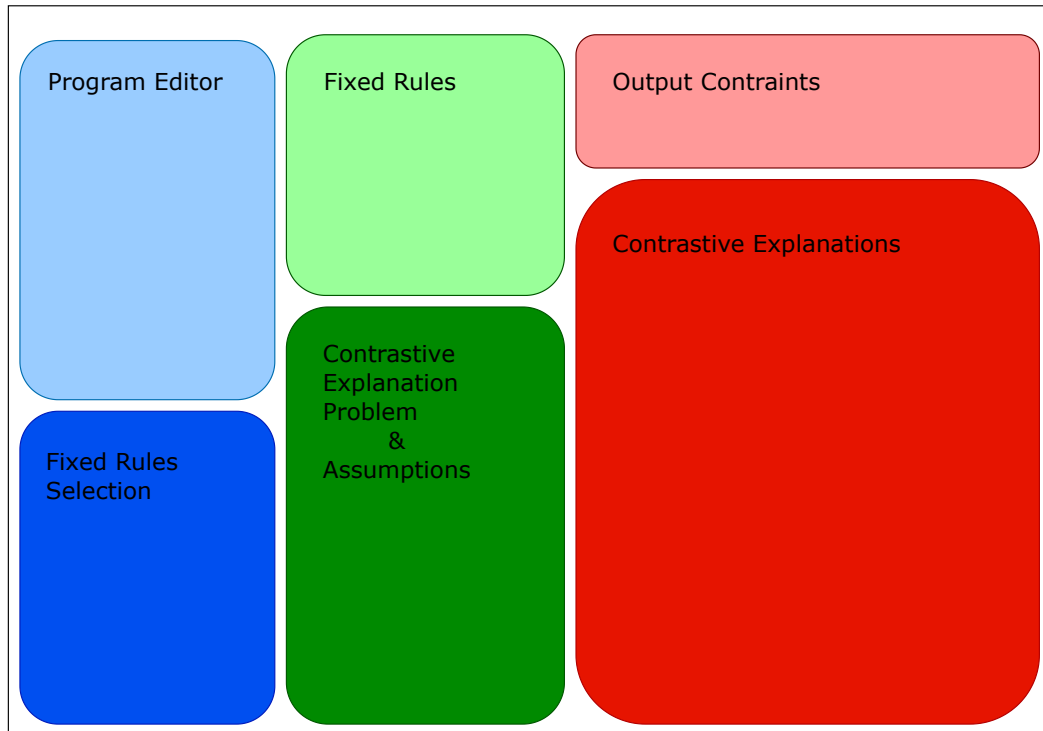


Figure 3.2: User Interface Mockup

The user interface provides an easy, user-friendly way to generate contrastive explanations. A mockup is presented in Figure 3.2. It is divided into three distinct sections, arranged in a columnar layout. The first, located on the left side of the interface, is shaded in blue and dedicated to show the original program P and its subset of fixed rules S . This column is further split into two parts. The upper one, highlighted in light blue, can be used to enter and edit the rules P . The lower one, shaded in dark blue, allows the user to select fixed rules as the set S .

The middle column, represented in green, is used to set the configuration. The top part of this column, colored in light green, can be used to add the set S of fixed rules, which is synchronized with the bottom part of the leftmost column. In the lower part of the middle column, shaded in dark green, the contrastive explanation problem together with

the set of assumptions A can be entered. The contrastive explanation problem comprises the explanandum E , foil F and the interpretation I in question.

The rightmost section, colored in red in Figure 3.2, deals with the generation and presentation of contrastive explanations. After the program and the configuration has been added, the contrastive explanations are shown in the dark red section of this column. Prior to execution, the upper light red section enables the users to define constraints on the output, i.e. the number of counterfactual accounts and the number of explanations to be generated.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Implementation

This chapter describes the implementation of the Python ¹ program for contrastive explanations for answer-set programs. The source code is licensed under the MIT open-source license and is publicly available online ².

First the customized grounder is introduced, since state-of-the-art grounders are not suitable to find contrastive explanations. Then, the algorithms to generate counterfactual accounts, counterfactual explanations and finally contrastive explanations are shown.

4.1 Internal Computation

In order to apply contrastive explanations to answer-set programs, the user needs to provide the following information:

First of all, the explanation frame has to be given. This consists of the original program, the common knowledge and assumptions. The common knowledge are those rules of the original program that the user wants to remain unchanged. The assumptions are additional facts that could be included, if they are needed to derive the foil.

Secondly, the user needs to input the contrastive explanation problem. This comprises an answer set of the original program, the explanandum and the foil. This answer set is surprising to the user, since it contains the explanandum and not the foil, as expected.

4.1.1 Grounder

The first step to generate contrastive explanations is to ground the original program P . Although the answer-set solver *clingo* [GKKS19] and its corresponding Python module

¹<https://www.python.org/>

²https://github.com/mallinger/Contrastive_ASP

provide a sophisticated grounder, grounding with it is not suitable to find contrastive explanations. This is due to the fact that the grounder may also remove or simplify rules, which are crucial for contrastive explanations. Generating a counterfactual account may rely on rules that have no impact on the answer sets of the original program. Those rules would be omitted by a modern grounder, since they only make solving harder. Hence, it is not adequate to use optimized grounders for finding contrastive explanations.

The grounding for this task has to be naive, i.e. no rule can be removed or modified other than replacing the variables with terms. Examples 1 and 2 depict such instances of a naive grounding. In Example 2, a modern grounder would simply remove the rule `result(b,b) :- pred(b,b) .`, since `pred(b,b)` cannot be part of an answer set. However, if we define `result(b,b)` to be the foil and include `pred(b,b)` as an assumption, the deleted rule is crucial for the derivation of the foil.

Our naive grounder does not support aggregates or conditionals in choice atoms. If the input program contains one of these, `NotImplementedError` will be raised. Similarly, the range notation is only supported for concrete numerical values and not for variables. The grounder also raises `NotImplementedError` if the interval notation is used with variables.

Before grounding takes place, the interval notations are expanded.

The actual grounding procedure consists of two steps. The first is to find all the constants in the program. This is shown in Listing 4.1. The program is given in string form as the variable `program_string`. The constants of a rule in string form can be identified via the regular expression `(?!not\b)\b[a-z0-9]\w*\b(?:\()`, which is saved as the variable `REGEX_CONSTANTS`. This expression matches substrings that begin with a lowercase letter or number, i.e. `[a-z0-9]\w*`. The word “not”, `(?!not\b)`, is excluded, since it is reserved as the keyword for negation as failure. Furthermore, the substring must not end with an opening parenthesis `(?:\()`, since this would signify a predicate or function. This version of the grounder does not support functions. The constants are collected in a list in Line 2 of Listing 4.1. The input `program_string` is split into individual rules in Line 3. Lines 4 to 10 extract the constants of a fact, i.e. a rule without a body and no arrow. If the rule is disjunctive, the head atoms have to be separated. Then, the function `findall` of the Python module `re` extracts the constants of each head atom using the expression in `REGEX_CONSTANTS`. The constants are added to the `constants` list. This step is only applied, if the head atom contains a “(”, because otherwise it cannot contain constants.

If the rule is not a fact, it is split up into the head and the body part. Then, in the same way as before, the constants of the atoms of the head and of the literals of the body are gathered and added to the `constants` list. Finally, duplicates are removed by applying `list(set(constants))`.

```
1 REGEX_CONSTANTS = r'(?!not)\b[a-z0-9]\w*\b(?:\('
2 constants = []
3 for rule in program_string[:-1].split(". "):
```

```

4     if ":" not in rule:
5         head_atoms = rule[:-1].split("|")
6         for head_atom in head_atoms:
7             if "(" in head_atom:
8                 constants.extend(re.findall(
9                     REGEX_CONSTANTS, head_atom))
10            continue
11
12    head, body = rule.split(":-")
13    head_atoms = head.split("|")
14    for head_atom in head_atoms:
15        if "(" in head_atom:
16            constants.extend(re.findall(
17                REGEX_CONSTANTS, head_atom))
18    for literal in literals_from_body(body):
19        if "(" in literal:
20            constants.extend(re.findall(
21                REGEX_CONSTANTS, literal))
22 constants = list(set(constants))

```

Listing 4.1: Python code that collects all constants from all the rules of an answer-set program.

Then in the second step, all rules have to be generated that result from replacing each variable in it with all possible constants. This procedure is depicted in Listing 4.2. First, an empty `Program` object `grounded_rules` is created to store grounded rules. Then the input `program_string` is split into individual rules. For each rule the regular expression `[^a-z]([A-Z]\w*)` is used to find all the variables. Variables can easily be identified, because they are the only words that may start with an uppercase letter.

If there are no variables in a rule, then the rule is already ground and can be added to `grounded_rules` as such. Otherwise, all possible combinations of constants for the set of variables have to be generated. This can easily be achieved by using the function `product(constants, repeat=len(variables))` of the Python module `itertools`. The `product` function generates the Cartesian product of the input list `constants`. The result will be a set of all possible tuples of constants. By setting the argument `repeat`, the amount of constants in each tuple is equal to the amount of variables in the respective rule. For each of these tuples a new grounded rule is created and added to `grounded_rules`. This is achieved by successively replacing each variable with the corresponding constant of the tuple. The function `sub` of the Python module `re` performs the substitution of the variable. In order not to match a word that contains the variable only as a substring, the regular expression `(?<![\w]){v}(?![\w])` is used. Here, the part `{v}` is replaced by the variable, while the surrounding `(?<![\w])`, respective `(?![\w])`, make sure that there is no letter before or after the variable.

```

1 REGEX_VARIABLES = r'[^a-z]([A-Z]\w*)'
2 REGEX_SUBSTITUTION = lambda v: rf'(?<![\w]){v}(?![\w])'
3 grounded_rules = Program("")
4 for rule in re.split(r"\.s+", program_string[:-1]):
5     variables = set(re.findall(REGEX_VARIABLES, rule))

```

```
6     if not variables:
7         grounded_rules.add_rule(f"{rule}.")
8     continue
9     for c in list(product(constants, repeat=len(variables))):
10        grounded_rule = rule
11        for i, v in enumerate(variables):
12            grounded_rule = re.sub(
13                REGEX_SUBSTITUTION, c[i], grounded_rule)
14        grounded_rules.add_rule(f"{grounded_rule}.")
```

Listing 4.2: Python code that generates all possible ground rules.

Once the program is grounded, all arithmetic expressions and relations are simplified. The simplification process begins with the evaluation of arithmetic relations. An empty `Program` object is created in order to store the resulting rules. Due to the immutability of elements within a set in Python, each rule must be duplicated to allow for modifications. The variable `valid` is initially set to `True` and is only set to `False`, if a relation in the body does not hold. Each literal of the body is examined, whether it contains a relation symbol `<`, `<=`, `>`, `>=`, `=`, `!=` or `not`. If such an operator is present, the literal is identified either as an arithmetic relation or as part of a choice atom. In the case of a choice atom, the literal is removed and reinserted into the rule after its guard has been evaluated, provided the guard represents an arithmetic expression. This evaluation is carried out using Python's built-in `eval` function. If the relation symbol is not part of a choice atom, both sides of the relation have to be compared. The grounder supports not only arithmetic comparisons but also equality and inequality checks between symbolic names and tuples. In these cases no evaluation is necessary and the two terms can be compared directly. If the two sides are arithmetic expressions, they are evaluated and then compared. If the relation is valid, the literal can be safely removed from the rule, since it has no effect on the satisfiability of the rule. However, if the relation does not hold, the entire rule is rendered unsatisfiable and is subsequently discarded. In this case, the variable `valid` is set to `False` and further evaluation of the remaining literals in the rule body is skipped. Following the simplification of the body, the head of the rule is examined. Relational operators are permitted in the head only within choice atoms. Otherwise a `SyntaxError` is raised. If the variable `valid` is `True`, the rule will be added to the `Program` object.

Following the simplification of arithmetic relations, any remaining arithmetic expressions are evaluated. In contrast to arithmetic relations, this step does not result in the removal of entire atoms, literals, or rules. Since expressions involved in relational comparisons have already been processed, any remaining arithmetic expressions are now confined to the terms within predicates. Similar to the procedure of simplifying arithmetic expressions, a new empty `Program` object and a copy of each rule are created. Each rule is then examined in turn: every atom in the head and every literal in the body is analyzed to determine whether it contains an arithmetic operator from the set $\{+, -, *, /\}$. When such an operator is found, it is necessary to distinguish between expressions occurring within choice atoms and those within standard predicates. If the operator appears inside

a choice atom, each predicate inside that choice atom is individually examined. Once the predicate that contains an arithmetic expression is identified, each term of the predicate has to be checked. Any term that is determined to be an arithmetic expression is then evaluated and simplified using Python's built-in `eval` function.

4.1.2 Counterfactual Account

In order to include the explanandum E and foil F as such in an answer-set program, it is necessary to mark them with the predicates `explanandum/1` and `foil/1`.

In a similar way, the predicate `assumption/1` is used to denote each assumption. We refer to the atoms with predicates `explanandum/1`, `foil/1` and `assumption/1` as *meta atoms*.

Afterwards, the whole program together with assumptions, foil and explanandum, each with their respective predicates, is reified.

The reification procedure used here is a simplification of the algorithm described in Section 2.1.6. *Clingo's* function `reify_program` cannot be applied, since it may perform simplifications, similar to its grounder, which would remove or manipulate rules which are essential for the explanation. Instead of just assigning an ID to atoms and literals, also rules get one. The arguments of `choice`, `disjunction`, `normal` and `sum` here refer to the specific rule. For simplicity, IDs of atoms are strictly greater than the IDs of rules. `atom_tuple` and `literal_tuple` are used in the same way as in Section 2.1.6. In the reification step, all rules are represented by facts consisting only of atoms with the predicate `rule/2`. Deriving the foil may require the exclusion of some rules that are not part of the common knowledge. All meta atoms and rules that are in S are fixed and have to be part of the programs that derive the foil. Those rules are directly represented by the `rule/2` atoms. All other rules may be excluded. In order to depict these optional rules as such, these facts are turned into choice rules. This way, rules that block the derivation of the foil can simply be excluded.

Example 8. Example 5 showed the input and the counterfactual account of the bird prediction example. The corresponding reified program is provided in a sorted format in Listing 4.3, with some rules omitted for brevity. In the first lines of Listing 4.3, each atom is assigned an ID via the output predicate. The IDs start with index 11 as the program consists of ten rules. Each rule of the original program is represented by the `rule/2` predicate. A comment, starting with “%” was added in front of each rule for clarity. Note that the choice rule `{darkwings}` is also expressed by a choice `{rule(disjunction(4), normal(4))}` on Line 35.

```

1 output(assumption(whitewings), 11).
2 output(beak, 12).
3 output(feathers, 13).
4 output(shape, 14).
5 output(bird, 15).
6 output(darkwings, 16).
7 output(crow, 17).

```

4. IMPLEMENTATION

```
8 output(explanandum(crow), 18).
9 output(foil(magpie), 19).
10 output(whitewings, 20).
11 output(magpie, 21).
12
13 % assumption(whitewings).
14 rule(disjunction(0), normal(0)).
15 atom_tuple(0, 11).
16
17 % beak.
18 rule(disjunction(1), normal(1)).
19 atom_tuple(1, 12).
20
21 % bird :- beak, feathers, shape.
22 rule(disjunction(2), normal(2)).
23 atom_tuple(2, 15).
24 literal_tuple(2, 12).
25 literal_tuple(2, 13).
26 literal_tuple(2, 14).
27
28 % crow :- bird, darkwings.
29 rule(disjunction(3), normal(3)).
30 atom_tuple(3, 17).
31 literal_tuple(3, 15).
32 literal_tuple(3, 16).
33
34 % {darkwings}.
35 {rule(disjunction(4), normal(4))}.
36 atom_tuple(4, 16).
37
38 % explanandum(crow).
39 rule(disjunction(5), normal(5)).
40 atom_tuple(5, 18).
41
42 % feathers.
43 rule(disjunction(6), normal(6)).
44 atom_tuple(6, 13).
45
46 % foil(magpie).
47 rule(disjunction(7), normal(7)).
48 atom_tuple(7, 19).
49
50 % magpie :- bird, whitewings.
51 rule(disjunction(8), normal(8)).
52 atom_tuple(8, 21).
53 literal_tuple(8, 15).
54 literal_tuple(8, 20).
55
56 % shape.
57 rule(disjunction(9), normal(9)).
58 atom_tuple(9, 14).
```

Listing 4.3: Reified rules of the bird prediction example

```

1 :- hold(E), output(explanandum(A),_), output(A,N), literal_tuple(N,E).
2 :- not hold(F), output(foil(A),_), output(A,N), literal_tuple(N,F).
3 {hold(S)} :- output(assumption(A),_), output(A,N), literal_tuple(N,S).

```

Listing 4.4: Rules to include meta atoms for counterfactual accounts

The reified program can then be used to generate counterfactual accounts. In order to do so it needs to be combined with the rules for the meta atoms in Listing 4.4 and the meta encoding in Listing 2.3. Each subset-maximal answer set of these rules is an instance of I' of the definition of a counterfactual account. Subset-maximal answer sets can be generated via *heuristic-driven solving*. By including the directive `#heuristic rule(_, _). [1, true]` in the program, the solver is guided to generate subset-maximal answer sets.

The set A' of assumptions that are needed to derive the foil, can be directly extracted from I' .

P' can be computed by modifying the meta encoding of Listing 2.3. In the original version, the last two lines with the `#show` statement specify that the solver will just return the atoms of the original program that are true in the answer set. By removing these two lines, all reified atoms will be returned. It is then possible to reconstruct the original rules from the reified atoms. All of the optional rules that needed to be removed, will not appear in the answer set. The set P' is the set of rules of P that are still present in the answer set.

The tuple (P', I', A') constitutes a counterfactual account according to Definition 2.3.3.

The first condition $S \subseteq P'$ is satisfied, since all the rules of S are included in the reified program. The rules of S were added as non-choice rules and thus are required to be in every answer set.

The second condition $F \cap A' = I \cap A' = \emptyset$ depends on the input given by the user, but is also verified at the beginning of the procedure.

The third condition $I' \in AS(P' \cup A')$ is satisfied by the construction of the answer set program that generates I' . This answer set program consists of all rules P in reified form, the meta encoding and the rules for the meta atoms. The latter enforces that the explanandum is not part of I' , but that the foil is. In order to avoid deriving the explanandum in I' , some rules may need to be excluded. This is possible, since optional rules are expressed as choice rules in reified form. The remaining rules are P' . So as to derive the foil in I' , some assumptions of A may need to be included. This can be achieved with the last line of Listing 4.4. A' are all the assumptions that are included in I' . Hence, I' is an answer set of the program consisting only of P' and A' .

The fourth condition $F \subseteq I'$ and $E \not\subseteq I'$ is enforced by the first two lines of Listing 4.4, which is part of the answer-set program that generates I' . The explanandum A is identified by the atom `output(explanandum(A, _))`. The same variable A appears in `output(A, N)`, which links it to its ID N . N is further linked to the reference ID E via

the atom `literal_tuple(N, E)`. Thus, `atom_hold(E)` is true, if the explanandum A is true in the answer set I' . Since the rule is a constraint, it is impossible that the explanandum appears in the answer set I' . The second rule of Listing 4.4 links the foil to the `hold` atom in the same way. The difference here is that `hold` is negated, which enforces that the foil is part of the answer set I' .

The last condition requiring that P' is \subseteq -maximal, is satisfied, since the answer set is computed with a \subseteq -maximal heuristic. The `#heuristic` directive was added to the program as `#heuristic rule(_,_) [1, true]`. This forces the solver to generate all answer sets that are \subseteq -maximal with respect to the `rule` atom.

4.1.3 Counterfactual Explanation

The first step to compute counterfactual explanations for a given explanation frame and a contrastive explanation problem is to generate the corresponding counterfactual accounts.

The counterfactual explanation tuples $\langle Q_1, Q_2, Q_\Delta \rangle$ can then be extracted from the counterfactual accounts (P', I', A') . Q_1 is a minimal subset of P that derives the explanandum and uses as few rules from P' as possible. Similarly, Q_2 is a minimal subset of P' and A' that derives the foil. Q_Δ is the set of rules that needed to be removed from P in order to be able to derive the foil.

In order to compute the minimal subset Q_1 of P , a new answer set program is created. This program contains the set P . Then a new literal is added to each rule. If this literal evaluates to false, the rule becomes unsatisfiable and can be discarded. For all rules in P' , the literal `active_P_prime/1` is appended, while for all other rules, the literal `active/1` is used. Both literals take a rule ID as their argument. For every `active(i)` literal, the choice rule `{active(i)}` is included, and similarly, the choice rule `{active_P_prime(i)}` is included for each `active_P_prime(i)` literal. This way each `active` and `active_P_prime(i)` literal can be set to false, if the corresponding rule is not needed.

The rule `:- not e.` is added to the program for the explanandum e . This rule forces the solver to find an answer set that includes the explanandum.

To compute minimal subset, the following rules are appended to the program:

```
#heuristic active_P_prime(_). [2, false]
#heuristic active(_). [1, false]
```

These rules are `#heuristic` directives. Similar to the usage of *heuristic-driven solving* for counterfactual accounts, the heuristic here is applied to the atoms `active/1` and `active_P_prime/1`. The main difference here is the usage of the modifier `false`. This instructs the solver to generate subset minimal answer sets in regard of the two atoms. The directive for the atom `active_P_prime/1` has a higher level than the atom `active/1`, which causes the solver to first minimize the amount of `active_P_prime/1` atoms in the answer set.

The answer set for this program represents a way to derive the explanandum with as few of P' and P rules as possible.

Q_2 can be computed likewise. Instead of using the rules of P , the program consists of the rules of P' and A' . The rule `:- not e.` is replaced by the rule `:- not f.` for the foil f .

Q_Δ is simply defined as the rules P without the rules of P' .

Example 9. In order to generate Q_1 of the counterfactual explanation presented in the Example 6 the following program is created:

```

1 darkwings :- active(0).
2 feathers :- active_P_prime(1).
3 magpie :- bird, whitewings, active_P_prime(2).
4 crow :- bird, darkwings, active_P_prime(3).
5 shape :- active_P_prime(4).
6 beak :- active_P_prime(5).
7 bird :- feathers, beak, shape, active_P_prime(6).
8 {active(0)}.
9 {active_P_prime(1)}.
10 {active_P_prime(2)}.
11 {active_P_prime(3)}.
12 {active_P_prime(4)}.
13 {active_P_prime(5)}.
14 {active_P_prime(6)}.
15 :- not crow.
16 #heuristic active_P_prime(_). [2, false]
17 #heuristic active(_). [1, false]

```

Listing 4.5: Rules needed to generate Q_1 of the counterfactual explanation.

The first seven lines in the program of Listing 4.5 show the rules of P with the specific literals, `active(_)` and `active_P_prime(_)`, added to the bodies of the rules. Since `darkwings.` is the only rule not part of P' , it is the only rule with the literal `active(_)` added to it. The corresponding choice rules are appended in the subsequent seven lines. The constraint `:- not crow.` is added to the program for the respective explanandum. This forces the answer set of this program to include the atom `crow`, which can only be achieved by using the rules of P . The last two lines of Listing 4.5 enforce the use of as few rules as possible. The solver prioritizes minimizing the amount rules of P' first. Finally, the set Q_1 can then be extracted from the answer set of this program by tracing the atoms `active(_)` and `active_P_prime(_)` back to the corresponding rules.

4.1.4 Contrastive Explanation

Contrastive explanations $\langle C_1, C_2, C_\Delta \rangle$ are a simplified version of counterfactual explanations $\langle Q_1, Q_2, Q_\Delta \rangle$. The set C_1 reduces the set Q_1 to only include those rules that help the explainee. The set S can be removed, because it represents the common knowledge. The explainee is already aware that this must be the case. All rules of Q_2 that are also part of Q_1 are excluded, since the explainee expects the intersection of both to be case.

The set C_2 is defined similarly, as it consists of the set Q_2 without the rules of S or Q_1 . C_Δ consists of the rule of Q_Δ without the rules of S .

Example 10. Continuing the Example of 9, Example 7 presented the contrastive explanation $\{\{\text{darkwings.}\}, \{\text{whitewings.}\}, \{\text{darkwings.}\}\}$.

The intersection of Q_1 and Q_2 is `bird :- feathers, beak, shape. feathers. beak. shape..` These rules can be omitted in the contrastive explanation. Furthermore, all rules of the set S can be excluded from Q_1 and Q_2 . These are the rules `magpie :- bird, whitewings.` and `crow :- bird, darkwings.` respectively. By removing these rules from Q_1 we get C_1 , which is the set $\{\text{darkwings.}\}$. The same procedure is applied to Q_2 to get C_2 as $\{\text{whitewings.}\}$.

4.2 User Interface

This Section shows the usage of the program for contrastive explanations for answer-set programs. The user interface was created using the *PySide6*³ framework. *PySide6* is the official Python module from the *Qt for Python project*⁴. *Qt* is a cross-platform framework for developing graphic user interfaces. Figure 4.1 shows the user interface of the program.

4.2.1 Workflow

Adding the program

First, the program has to be included in the leftmost section of the interface. This can be achieved by manually writing the rules in the text box or by clicking on the **Add Program** button. In the latter case, a dialog menu will pop up that allows the user to select a text file. This file needs to contain a valid answer set program. Once the dialog menu has been closed by clicking on the **open** button, every rule is appended to the text box. Furthermore, for each rule a new layout is added below the text box. The rules in the horizontal layouts are sorted, with facts being at the beginning and constraints being at the end. These horizontal layouts consist of a button labeled “S” and the rule. The button allows to select or deselect the corresponding rule as a fixed rule, which must not be excluded in the counterfactual account. The rules to the right of the button will appear in bold font, if they are fixed.

Configuration

Secondly, the configuration needs to be added. This includes the set of fixed rules, the assumption, the explanandum, the foil and an answer set of the program. The middle section of the interface contains the necessary text fields to enter the configuration. Similarly to the **Add Program**, there is also an **Add Configuration** button, which

³<https://pypi.org/project/PySide6>

⁴<https://www.qt.io/qt-for-python>

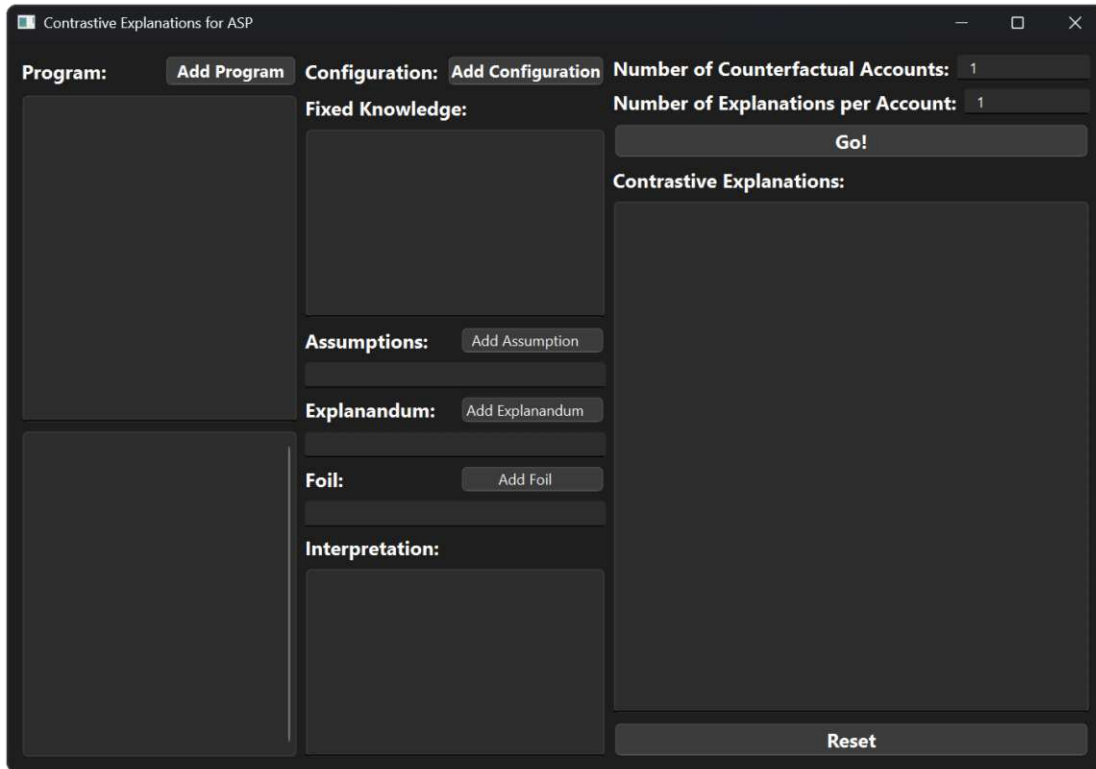


Figure 4.1: The empty user interface

offers the possibility to select a `json` file. This file needs to contain five name-value pairs. The names need to be S , A , I , E , F . The value for S has to be the answer set program in string format. The values for the other names need to be lists of strings representing the atoms. An example of such a configuration file is depicted in Listing 4.6.

```

1 {
2   "S": "crow :- bird, darkwings. magpie :- bird, whitewings. bird :-
   feathers, beak, shape. shape. beak. feathers.",
3   "A": ["whitewings"],
4   "I": ["crow", "bird", "feathers", "beak", "shape", "darkwings"],
5   "E": ["crow"],
6   "F": ["magpie"]
7 }
```

Listing 4.6: Example of a configuration file.

After choosing a valid configuration file, the respected text fields will be filled automatically. This will override any existing selection of fixes rules of the horizontal layouts in the leftmost program section. The text field of the fixed knowledge segment is synchronized with these horizontal layouts. This means that adding a rule to the text field will also mark the rule in the layout. Selecting, respectively deselecting, the rules to the left will add, respectively remove, the rules in the text field.

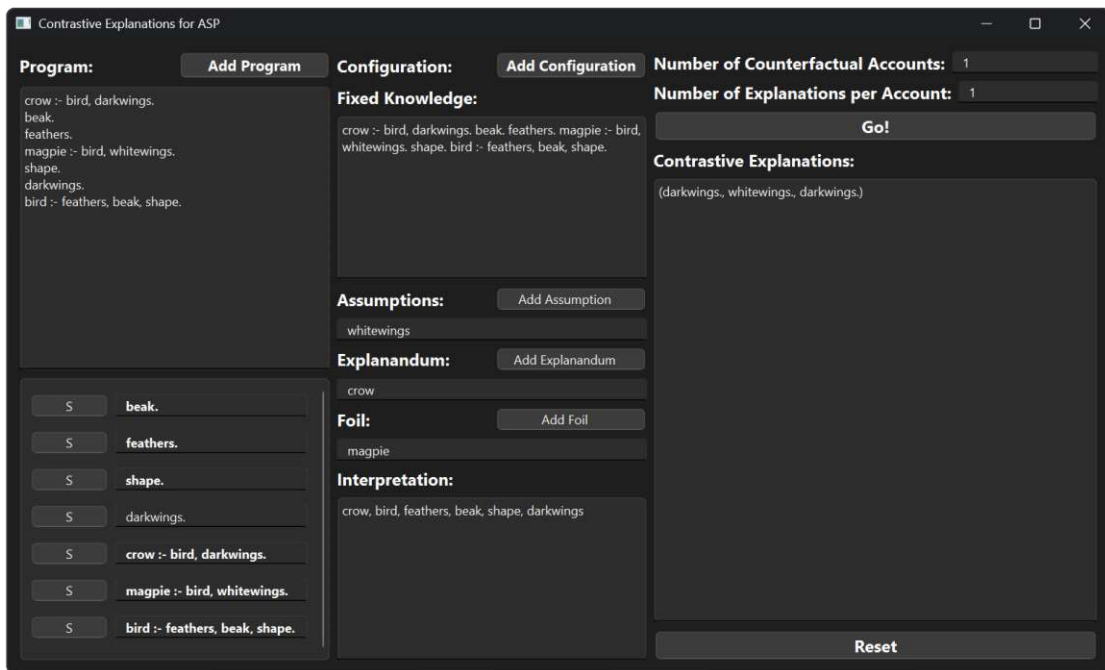


Figure 4.2: The user interface showing a contrastive explanation

Assumptions, the explanandum and the foil can be entered manually in the corresponding text fields or added manually by using the `Add Assumption`, `Add Explanandum` and `Add Foil` buttons. Clicking on the buttons will open a menu consisting of the atoms of the program of the leftmost section. The selected atom will be appended to the according text field. There is an auto-complete feature, once the user starts typing inside the text fields. This feature reduces the likelihood of typographical errors and facilitates a smoother user experience. The interpretation has to be entered manually.

Result

The final step is to click on the **Go!** button at the top of the rightmost section of the interface. This will trigger the grounding of the input program. In case of syntactical errors, a popup will appear informing the user of the problem. Mismatched parentheses and commas are pointed out precisely. If the input program contains aggregates or conditionals within choice atoms, the grounder also raises `NotImplementedError`, which is shown to the user. Once grounding succeeds, all other input values are checked. Neither the explanandum nor the foil must be empty, in order to generate a contrastive explanation. If one of their text fields is empty, the corresponding label will be colored in red, reminding the user to provide the information. A counterfactual account can only be created, if the foil is part of some rule of the input program. In the same way, the explanandum also needs to appear in the input program. If this is not the case, an error message is displayed.

If all inputs are valid, the fixed knowledge rules are grounded. It is not possible to simply ground the set of rules given in the fixed knowledge text field, since it does not necessarily contain all the relevant constants. Hence, the list of constants needed for grounding is taken from the input program.

Just above the **Go!** button, there are two input fields for the number of counterfactual accounts and counterfactual explanations per account, respectively. These inputs determine the amount of generated contrastive explanations. Clicking on the **Go!** button will start the generation of the contrastive explanations. These will then be displayed in the text field below the label **Contrastive Explanations:**. If there is no contrastive explanation for the given explanation frame and contrastive explanation problem, an error message will pop up.

There is a **Reset** button at the bottom of the rightmost section, which clears all input fields.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Evaluation

This chapter consists of two sections and evaluates the efficiency, usability and user-friendliness of the program. The Section 5.1 deals with performance by measuring the runtime of the prototype on different explanation tasks. The Section 5.2 is concerned with user experience, examining how intuitive and accessible the tool is.

5.1 Performance

In order to assess the program's performance, three different problems have been chosen for which a contrastive explanation has to be created. These three problems are *n*-queens, *Sudoku* and *coloring*. In all cases, the input program is grounded and the grounding time is not included in the resulting runtime. Furthermore, the program only generates one contrastive explanation.

5.1.1 Testing Setup

All experiments were conducted on a machine equipped with an AMD Ryzen 5 3600X 6-Core Processor running at 3.8 GHz, supported by 32 GB of RAM. The system operated under Windows 11. The software environment consisted of Python 3.12.3 and clingo version 5.8. All source code, datasets, and software configurations used in this thesis are openly available via Zenodo (DOI: 10.5281/zenodo.17621875).

The *n*-queens problem was evaluated for instances ranging from 4 to 31 queens, while the Sudoku problem was tested using grid sizes of 4×4 , 9×9 , 16×16 and 25×25 . For both problem domains, each test case was executed five times to ensure consistency and reliability of results. The runtime performance for each set of runs is illustrated through corresponding benchmark box plots. Additionally, the average runtime across the five executions is presented in the associated benchmark tables.

For the *coloring* problem, a reference graph was selected from the fourth Answer Set Programming Competition[ACC⁺13]. Subgraphs were subsequently derived by progressively removing nodes along with their associated edges. Unlike the approach used for the *n*-queens and Sudoku problems, each subgraph was tested three times, with each test executed on a different node, rather than repeating the same test multiple times on a single graph instance.

5.1.2 n-Queens

The *n*-queens problem consists of a chess board with *n* rows and columns. The task is to place *n* queens on the board, such that no two queens attack each other. This means that in each row, column and diagonal exactly one queen has to be placed. Figure 5.1 depicts a solution for the 8-queens problem.

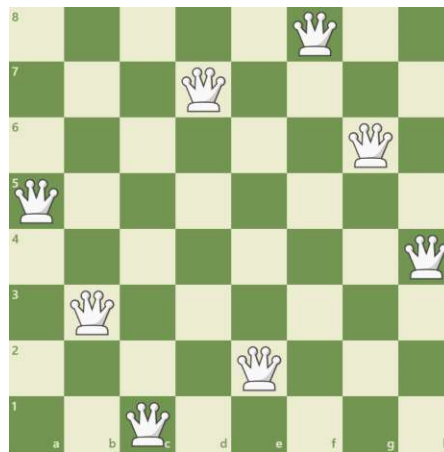


Figure 5.1: A solution for the 8-queens problem¹

One possible ASP encoding is shown in Listing 5.1. The predicate `queen(X, Y)` is used to indicate that a queen is placed in row *X* and column *Y*. The first line forces the solver to place exactly *n* queens in total. Lines 2 and 3 are constraints that forbid the placement of two queens in the same row (Line 2) or column (Line 3). The last two lines of Listing 5.1 deal with the diagonal restrictions.

```

1 { queen(1..n, 1..n) } == n.
2 :- queen(R, C), queen(R, C2), C != C2.
3 :- queen(R, C), queen(R2, C), R != R2.
4 :- queen(R, C), queen(R2, C2), (R, C) != (R2, C2), R - C == R2 - C2.
5 :- queen(R, C), queen(R2, C2), (R, C) != (R2, C2), R + C == R2 + C2.

```

Listing 5.1: Answer set program for the *n*-queens problem

In order to test the performance for finding one contrastive explanation for the *n*-queens problem, a queen is placed in row 1 and column 2 and a second queen in row 2 and

¹Image created via www.chess.com/analysis?tab=analysis

column 4. The question for the contrastive explanation then is, why can the second queen not be placed in row 1 and column 3. Of course, the answer is that the first queen already occupies the first row. In the concrete formulation of the explanation frame, the set S of fixed rules is defined as the rule in Listing 5.1. The set P includes the set S plus additionally the fact $queen(1, 2)$. Since no assumptions are needed here, the set A is empty. In case of the contrastive explanation problem, the explanandum is $queen(2, 4)$ and the foil is $queen(1, 3)$. The only contrastive explanation for this is the tuple $\langle \emptyset, \emptyset, \{queen(1, 2)\} \rangle$. Both, the explanandum and foil, can be derived by the first rule of Listing 5.1, which is part of the set S . The only rule that needs to be removed from the original program, so the foil can be derived, is $queen(1, 2)$..

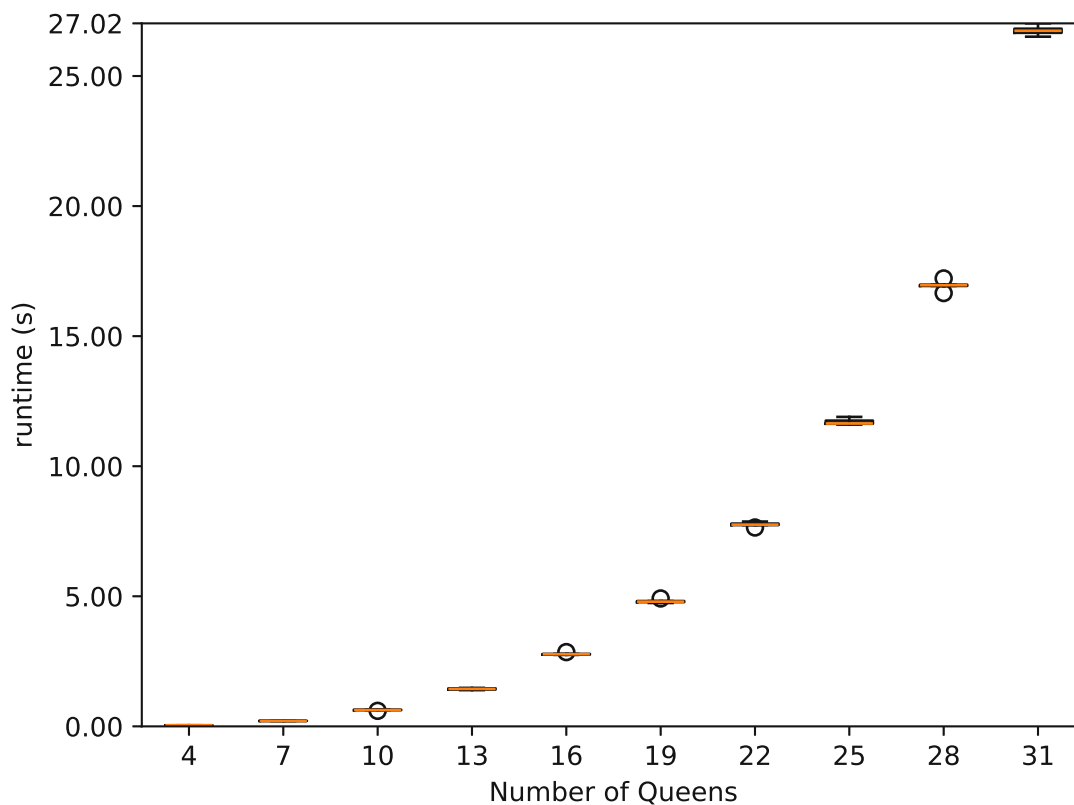


Figure 5.2: n-Queens Problem benchmark box plot

Figure 5.2 presents the box plot benchmark results for the n -queens problem, covering instances ranging from 4 to 31 queens in increments of 3. Each instance was executed five times, with minimal variation observed in runtime across the repeated runs for a given instance. As expected, the runtime increases consistently as the number of queens grows. Table 5.1 provides a more detailed overview of the results, listing all instances from 4 to 31 queens. In addition to runtime data, the table includes the number of grounded rules generated for each instance and the average runtime. Similar to the runtime, the number

Table 5.1: n-Queens Problem benchmark table

Number of Queens	Amount of Grounded Rules	avg. Runtime (s)
4	78	0.0414
5	162	0.0778
6	292	0.134
7	478	0.2104
8	730	0.3198
9	1058	0.4334
10	1472	0.6258
11	1982	0.8418
12	2598	1.0966
13	3330	1.4384
14	4188	1.8098
15	5182	2.3041
16	6322	2.7839
17	7618	3.3601
18	9080	4.0242
19	10718	4.8097
20	12542	5.8928
21	14562	6.7453
22	16788	7.761
23	19230	9.0115
24	21898	10.5089
25	24802	11.705
26	27952	13.4801
27	31358	15.0169
28	35030	16.9489
29	38978	18.7999
30	43212	21.1009
31	47742	26.7514

of grounded rules exhibits a steady increase with the size of the instance.

5.1.3 Sudoku

Sudoku is a logic puzzle, in which numbers have to be filled into a grid. As a restriction, each number must appear only once in each row and column. Additionally, the grid is divided into subgrids, which also must contain each number exactly once. The most popular form of Sudoku uses a 9×9 grid with 3×3 subgrids. However, it is also possible to create larger grids. The next larger Sudoku uses a 16×16 grid with 4×4 subgrids, and so on.

```

1   row(1..9).
2   column(1..9).
3   {sudoku(X, Y, 1..9)} = 1 :- row(X), column(Y).
4   :- sudoku(X, Y, N), sudoku(A, Y, N), X != A.
5   :- sudoku(X, Y, N), sudoku(X, B, N), Y != B.
6   subgrid(X, Y, A, B) :- row(X), row(A), column(Y), column(B), (X-1)/3 == (
   A-1)/3, (Y-1)/3 == (B-1)/3.
7   :- sudoku(X, Y, V), sudoku(A, B, V), subgrid(X, Y, A, B), X != A, Y != B.
```

Listing 5.2: Answer set program for sudoku

Listing 5.2 shows an ASP encoding for the classical 9×9 Sudoku. The predicate `sudoku(X, Y, N)` is used to denote that the number N is assigned to the cell in row X and column Y . The first two Lines define the grid with the predicates `row`, `column`. Line 3 forces the answer set to assign exactly one number between 1 and 9 to each cell. A cell is determined by the coordinates X and Y via the predicates `row` and `column` in the rule body. The head of the rule is a choice rule. Its choice elements are the atoms `sudoku(X, Y, 1)`, `sudoku(X, Y, 2)` ... `sudoku(X, Y, 9)`, generated by the range notation `1..9`. The relation “=” together with the guard 1 forces the answer set to contain exactly one of those atoms. Lines 4 and 5 prohibit the same number to appear in the same row and column, respectively. Line 6 defines the subgrid relation between two cells. Two cells belong to the same subgrid, if their rows fall into the same section as well as their columns. Since there are 3 row sections in a Sudoku with 9 rows, the rows 0-2, 3-5, 6-8 fall into the same section. The section can be calculated by subtracting 1 from the row and dividing by 3. Hence, the predicate `subgrid(X, Y, A, B)` is true in an answer set, if the cell in row X and column Y belongs to the same subgrid as the cell in row A and column B . The last Line prohibits two numbers from being in the same subgrid, unless they are the same cell.

```

1   sudoku_prg = f"""
2       row(1..{n}).
3       column(1..{n}).
4       :- sudoku(X, Y, N), sudoku(A, Y, N), X != A.
5       :- sudoku(X, Y, N), sudoku(X, B, N), Y != B.
6   """
7   sudoku_atoms = [f"sudoku(X, Y, {N})" for N in range(1, n + 1)]
8   sudoku_prg += " | ".join(sudoku_atoms) + " :- row(X), column(Y)."
9
```

```

10     subgrid_constraints = ""
11     for x in range(1, n + 1):
12         for y in range(1, n + 1):
13             for a in range(1, n + 1):
14                 for b in range(1, n + 1):
15                     if int((x-1)/sqrt(n)) == int((a-1)/sqrt(n)) and int((y-1)
/sqrt(n)) == int((b-1)/sqrt(n)) and x != a and y != b:
16                         for v in range(1, n + 1):
17                             subgrid_constraints += f" :- sudoku({x},{y},{v}),
sudoku({a},{b},{v})."
18     sudoku_prg += subgrid_constraints

```

Listing 5.3: Python implementation to generate the answer set program for Sudoku.

In order to dynamically generate an answer set program for larger Sudokus, to simplify and to make it more efficient, the program in Listing 5.2 has been modified in Listing 5.3. The answer set program is stored in the variable `sudoku_prg`. The first four lines also appear in the unmodified version. These define the grid and prohibit two numbers from being in the same row and column. One modification for performance reasons was applied in Line 7 in Listing 5.3. Instead of a choice rule, a disjunctive rule was used. First, a list of `sudoku(X, Y, N)` strings with `N` being replaced by all the numbers between 1 and `n + 1` is created. The list is then joined together into a string with the separator "|". Afterwards, `:- row(X), column(Y).` is appended to the resulting string, which represents the whole disjunctive rule. This rule is then appended to `sudoku_prg`. Instead of creating the atoms `subgrid(X, Y, A, B)` and using them in the constraint, the relevant subgrid constraints are directly generated in the modified program in Listing 5.3, in Lines 10 to 17. The four for-loops generate all possible row and column combinations for two cells. The if condition in Line 15 checks whether the cell in row `x` and column `y` belongs to the same subgrid in row `a` and column `b`. If this is the case and the two cells are not in the same row and column, i.e. $x \neq a$ and $y \neq b$, then new constraints are added to the program. This is done in Lines 16 and 17. For each number, a new constraint is added, such that the two cells in the same subgrid do not have the same number.

Similar to the n -queens problem, a simple contradiction is formulated to create a contrastive explanation. The number 1 is placed in the cell in row 1 and column 1. Then the number 2 is assigned to the cell right next to it in row 1 and column 2. The contrastive explanation question is, why the number 2 in column 2 cannot be the number 1. The answer is that this would violate the restriction of having the same number twice in one row. Hence, the number 1 in row 1 and column 1 would need to be removed, in order to be able to place the number 1 in row 1 and column 2. Considering the explanation frame, the set S of fixed rules is given by the encoding of Listing 5.3. The set P contains the set S as well as the fact `sudoku(1, 1, 1)`. The set A is the empty set, since there are no assumptions. The contrastive explanation problem is defined by the interpretation `sudoku(1, 1, 1) . sudoku(1, 2, 2) .`, explanandum `sudoku(1, 2, 2) .` and the foil `sudoku(1, 2, 1) .`

Table 5.2: Sudoku benchmark table

Length of Sudoku Row	Amount of Grounded Rules	avg. Runtime (s)
4	253	0.1273
9	7399	3.4934
16	80177	41.5273
25	500701	306.2521

The only contrastive explanation for this is that the fact `sudoku(1,1,1)` needs to be removed from the program, since the foil and explanandum can be derived by the set S .

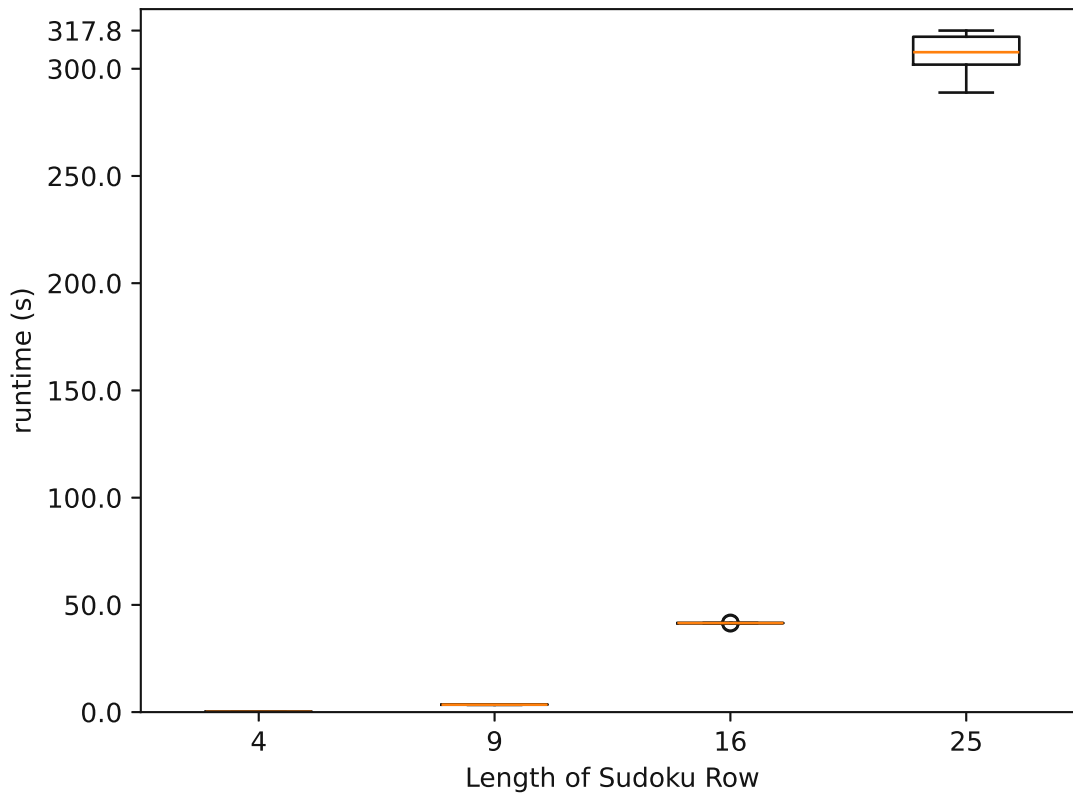


Figure 5.3: Sudoku benchmark box plot

Figure 5.3 shows the box plot benchmark for Sudoku. The evaluated grid sizes include 4×4 , 9×9 , 16×16 and 25×25 . Consistent with the methodology used for the n -queens problem, each instance was executed five times. The run times for each instance exhibited minimal variance, with the exception of the 25×25 grid, which showed a slightly higher variability. Table 5.2 provides the average runtime and the number of grounded rules for

each grid size. As the grid size increases, the complexity of the corresponding Sudoku instance rises sharply, which is clearly reflected in both the runtime and the number of grounded rules.

5.1.4 Vertex Coloring

In the *vertex coloring* problem a color has to be assigned to each vertex of a graph, such that no two adjacent vertices have the same color. If a graph has a vertex coloring with at most k colors, it is *k-colorable*. Such a coloring is called a *k-coloring*.

```

1   red(X) | blue(X) | green(X) | yellow(X) | cyan(X) :- node(X) .
2   :- red(X), red(Y), link(X, Y) .
3   :- blue(X), blue(Y), link(X, Y) .
4   :- green(X), green(Y), link(X, Y) .
5   :- yellow(X), yellow(Y), link(X, Y) .
6   :- cyan(X), cyan(Y), link(X, Y) .

```

Listing 5.4: Answer set program for the 5-coloring problem

Listing 5.4 shows an encoding of the 5-coloring problem with the colors: red, blue, green, yellow and cyan. The first line assigns a color to each node, via the respective predicate. Lines 2 to 6 forbid that two adjacent nodes have the same color. Adjacency between two nodes is expressed by the predicate `link/2`.

To evaluate the performance of the program on this problem, a graph and a valid 5-coloring is given. The goal is then to find out, how a specific node can be colored in a different color. This is always possible, because two colors of a valid k-coloring can be swapped without changing the k-colorable property.

The set S of fixed rules is given by the encoding of Listing 5.4 and the encoding of the specific graph, i.e. the facts `node(X)` and `link(X, Y)`. The set P contains the set S and a valid 5-coloring. Again, no assumptions are needed and the set A is empty. The interpretation is the 5-coloring. The explanandum is the atom denoting the color of the i^{th} node for the i^{th} test iteration, e.g. `b(1)` for the first iteration. Depending on the explanandum, the foil is either `g(1)`, if the explanandum is `b(1)`, otherwise the foil is `b(1)`. There are three iterations, one for each of the first three nodes.

For non-trivial cases, there are always many different contrastive explanations. Since the foil can always be derived via the first rule of Listing 5.4, the second element of each contrastive explanation tuple is the empty set. Similarly, the explanandum can be derived in the same way. However, since there also must be a fact justifying the explanandum in the program P as part of the coloring, this rule may also be chosen as the first element in the contrastive explanation. The last element of a contrastive explanation denotes the subset of rules of P that need to be removed in order to be able to derive the foil. This always includes the fact that justifies the explanandum. Besides this fact, the set may not consist of any other rule or it may be comprise all the rules of P , if the coloring of the counterfactual account is entirely different from the original coloring.

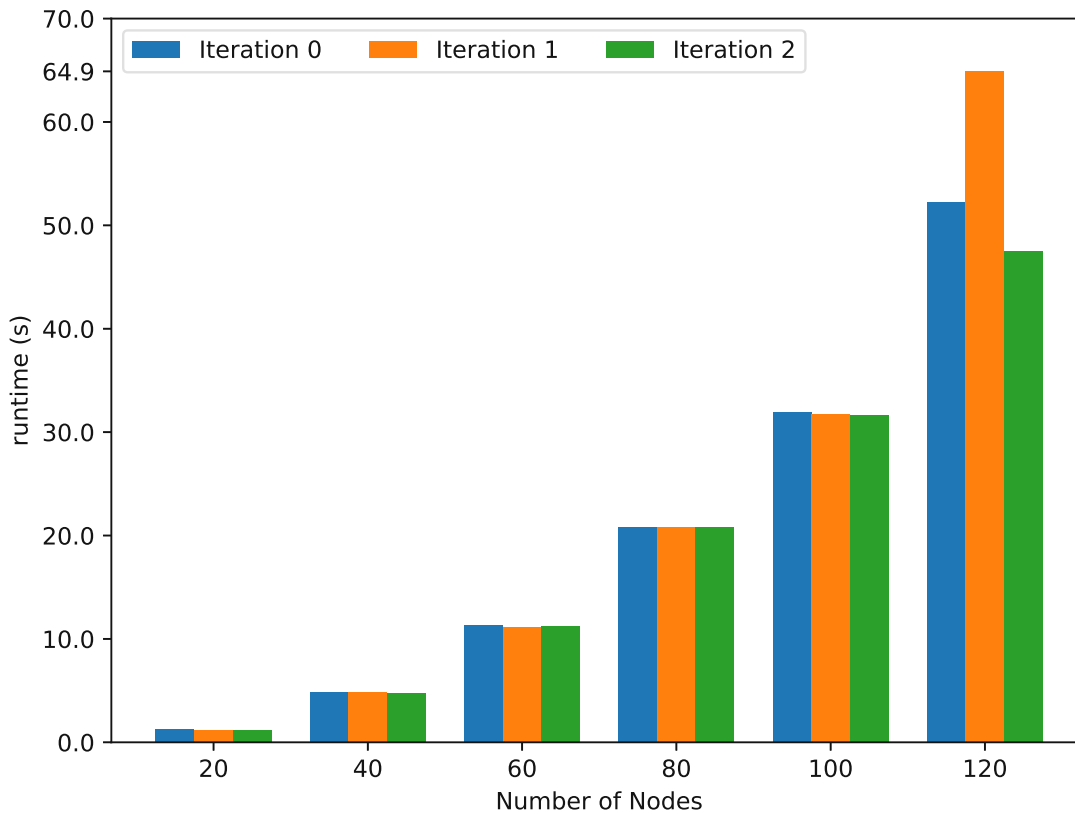


Figure 5.4: 5-Coloring Problem benchmark

The run times for smaller graphs remain consistent across all three iterations, as demonstrated in Table 5.3 and Figure 5.4, exhibiting a gradual increase with growing graph size. However, this trend shifts notably for graphs containing 120 nodes or more, where performance begins to diverge. In particular, contrastive explanation generation in iterations 0 and 2 is significantly faster than in iteration 1. For the graph with 120 nodes, iteration 1 exhibits approximately 20 percent longer runtime compared to iteration 0. This discrepancy becomes even more pronounced with the 125-node graph, where the runtime for iteration 1 exceeds that of iteration 0 by a factor greater than 10, see Table 5.3. Across all iterations, the run times for the 125-node graph deviate markedly from the previously observed steady increase, indicating a sharp rise in computational cost at this scale.

5.2 Case study

In this section the application of the program on two use cases are shown. By situating the interface within a realistic scenario, this section aims to bridge the gap between the theoretical design and practical application, providing insights into the interface's

Table 5.3: 5-Coloring Problem benchmark table

Number of Nodes	Amount of Grounded Rules	Run times (s)
10	540	0.3074 / 0.3236 / 0.2956
15	1184	0.6742 / 0.6783 / 0.6471
20	2088	1.2751 / 1.1538 / 1.1512
25	3254	1.8706 / 1.8505 / 1.8798
30	4670	2.6968 / 2.7144 / 2.6857
35	6350	3.6692 / 3.6251 / 3.6085
40	8292	4.8375 / 4.8493 / 4.7614
45	10462	6.2421 / 6.0913 / 6.0684
50	12892	7.69 / 7.632 / 7.6335
55	15586	9.4316 / 9.449 / 9.4509
60	18524	11.3548 / 11.0914 / 11.2525
65	21720	13.3838 / 13.1898 / 13.1758
70	25190	15.5951 / 15.9308 / 15.8332
75	28938	18.4073 / 18.2254 / 18.1198
80	32902	20.8273 / 20.8443 / 20.7631
85	37112	23.7433 / 23.4412 / 23.3652
90	41596	26.1667 / 26.7791 / 25.5144
95	46330	28.7276 / 28.6052 / 28.7419
100	51326	31.9635 / 31.6977 / 31.6762
105	56570	35.2839 / 37.1 / 36.6446
110	62072	40.1821 / 40.2226 / 40.328
115	67834	44.172 / 44.8251 / 44.2854
120	73836	52.263 / 64.8931 / 47.4851
125	80092	167.4499 / 1799.1933 / 278.8156

usability and overall user experience.

5.2.1 Crow-Magpie

The first use case is our running example of 5, 6 and 7. As a first step, the user inputs the given program in the top left text box or uses the **Add Program** button. The text box is highlighted in blue in Figure 5.5. As soon as the first rule `bird :- feathers, beak, shape.` has been inserted into the text box, a new row will be added to the field right below the text box, outlined in green in the same Figure. The user is certain that this rule is correct and must not be removed. This can be achieved by clicking on the “S” right next to the rule. This will highlight the rule and also add it to the text field in the top middle section of the interface, which is marked in red in Figure 5.5.

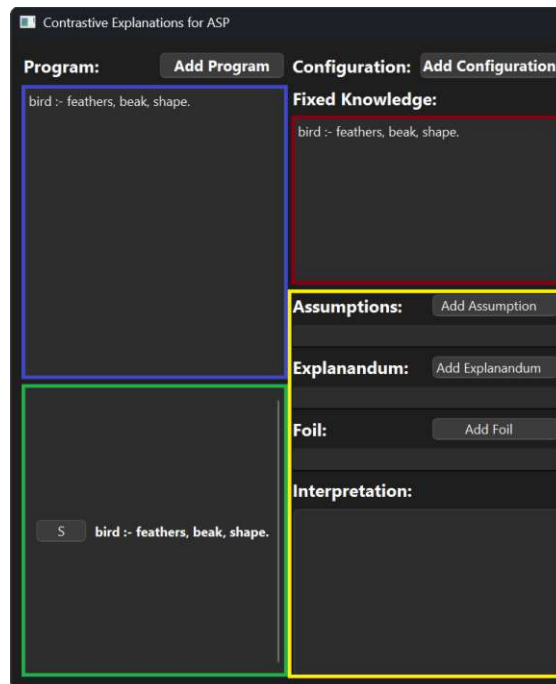


Figure 5.5: User enters the first rule and marks it as fixed.

As the title suggests, these rules represent the fixed knowledge. The user continues to add the remaining rules and marks them as fixed, except for `darkwings`..

In the subsequent phase of the workflow, the user is required to explicitly define the assumptions, the explanandum, the foil, and the corresponding interpretation. This task is carried out in the central portion of the user interface, which is highlighted in yellow in Figure 5.5.

For the specification of the assumptions, explanandum, and foil, the system offers two primary modes of input. First, the user may utilize the dedicated buttons labeled **Add Assumption**, **Add Explanandum** or **Add Foil**. Upon activation, each of these buttons presents a drop down menu that lists all available predicates of the program. The user can then select the appropriate predicate from the list, for instance, by clicking **Add Assumption** and choosing the predicate `whitewings`. Alternatively, predicates may be entered manually into the respective input fields. In this case, the interface provides real-time support through an auto-complete feature, which assists the user by suggesting valid predicate names based on partial input. Since the user did not expect the answer set to include `crow` they enter it into the respective text field. Conversely, the user assumed the answer set would contain `maggie` and enters it below. Afterwards, the user adds the interpretation in the designated field located in the lower central area of the interface.

Before proceeding to generate explanations, the user may configure the number of

counterfactual accounts and the number of counterfactual explanations per account to be generated. By default, both values are set to one, but they may be increased if the user seeks a more exhaustive exploration of alternative scenarios. This can be done in the upper right area of the interface, which is highlighted in orange in Figure 5.6. After clicking on the **Go!** button on the right-hand side of the interface, the contrastive explanation (`darkwings. whitewings. darkwings.`) will be displayed in the section marked in light blue in Figure 5.6. This will help the user understand that the bird was classified as a crow, because there were dark wings. It would have been a magpie if there were white wings. In order to change the outcome, the fact `darkwings.` needs to be removed from the original program.

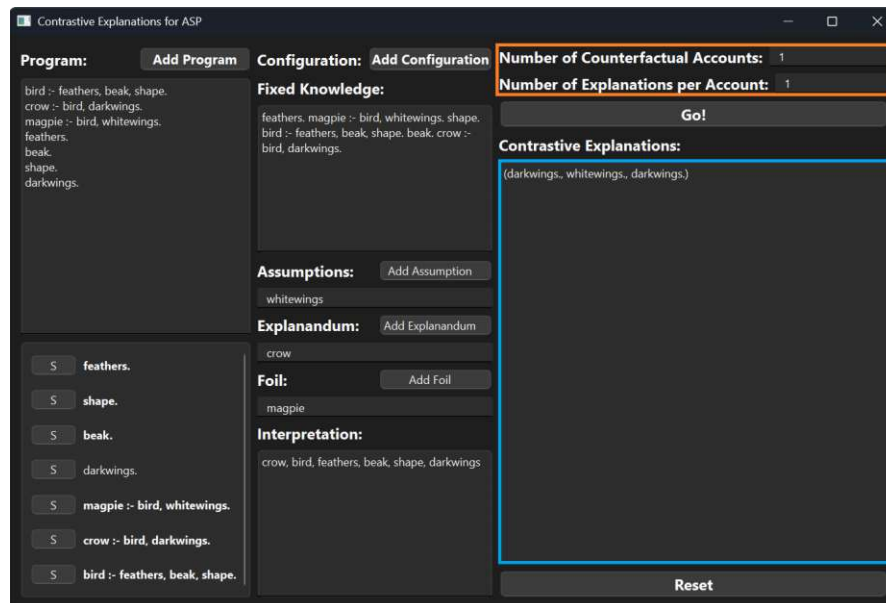


Figure 5.6: The interface is filled out and depicts a contrastive explanation.

Suppose that the user is uncertain, whether the rule `crow :- bird, darkwings.` is fixed or not. To explore this, the user deselects the rule from the rule set shown on the left side of the interface and increases the number of counterfactual accounts to two. After re-executing the explanation generation by clicking **Go!**, the system presents two distinct contrastive explanations. The first is `(darkwings. crow :- bird, darkwings., whitewings., darkwings.)`. The second contrastive explanation is the same regarding the first two elements: `(darkwings. crow :- bird, darkwings., whitewings., crow :- bird, darkwings.)`. The difference between the two contrastive explanations is the way the derivation of the atom `crow` can be blocked. In the first case, the fact `darkwings.` is removed, indicated by the third element of the tuple. This way, the rule `crow :- bird, darkwings.` can never be satisfied and `crow` cannot be derived. In the second case, the rule `crow :- bird, darkwings.` itself is removed. Hence, even if the atom `darkwings` is present in the answer set, `crow` will not be. The rule

`crow :- bird, darkwings.` appears now in both cases among the rules to derive `crow`, since it is no longer fixed and it is not required to derive `magpie`.

Contrastive explanations can support the user in two distinct ways. First, consider the scenario in which the original program is incorrect, and the image actually depicts a magpie rather than a crow. In this case, the bird does not have dark wings but white ones. Therefore, the fact `darkwings.` should not be present in the program—a discrepancy that the contrastive explanation highlights. This allows the user to investigate why `darkwings..` was included instead of the correct fact `whitewings..`, thereby facilitating the identification and correction of the underlying error.

In a second scenario, assume that the image indeed shows a crow, but the user lacks sufficient knowledge to recognize it and mistakenly identifies the bird as a magpie. While the user input and contrastive explanations remain the same as in the first scenario, the role of the explanation changes: it now aids the user in understanding why the classification as a crow is correct. Specifically, the presence of dark wings, emphasized by the contrastive explanation, serves as a key distinguishing feature, helping the user to correctly classify the bird.

5.2.2 3-Coloring

The second use case explores the 3-coloring problem, which was initially introduced in Section 5.1.4 as the 5-coloring problem. The formulation for the 3-coloring problem is presented in Listing 5.5, and it mirrors the encoding of the 5-coloring problem shown in Listing 5.4, with the only difference being the removal of two color options.

```

1   red(X) | blue(X) | green(X) :- node(X).
2   :- red(X), red(Y), link(X, Y).
3   :- blue(X), blue(Y), link(X, Y).
4   :- green(X), green(Y), link(X, Y).
```

Listing 5.5: Answer set program for the 3-coloring problem

The graph for our example is simple. It consists of three nodes, `one`, `two` and `three` and two links. The first link connects node `one` and node `three`, while the other link connects node `two` and node `three`. In the initial configuration, node `one` is assigned the color green and node `two` the color red. Due to the constraints imposed by the two links, the only valid coloring for node `three` is blue.

The user inputs the rules of Listing 5.5 into the text box at the top left of the interface, highlighted in blue in Figure 5.7. Next, the user adds the graph specific facts, i.e. `node(one).`, `node(two).`, `node(three).`, `green(one).`, `red(two).`, `link(one, three).` and `link(two, three).` in the same area. The program is non-ground, as it contains variables in the rules of the problem formulation.

These non-ground rules are essential components of the problem formulation. Hence, the user marks them as fixed in the area outlined in green or manually inputs them in the text field marked in red in the same Figure. The set of nodes is also fixed and cannot be

altered. However, unlike the 5-coloring example in Section 5.1.4, the graph's links are not fixed and may be modified to generate contrastive explanations.

In this scenario, the user seeks to understand why node `three` is colored blue instead of green. To do this, they specify the explanandum as `blue(three)`. This can be done either by using the **Add Explanandum** button or through the auto-completion feature, both of which support all ground predicates defined in the program. The same process applies when defining the foil. This area is highlighted in yellow in Figure 5.7.

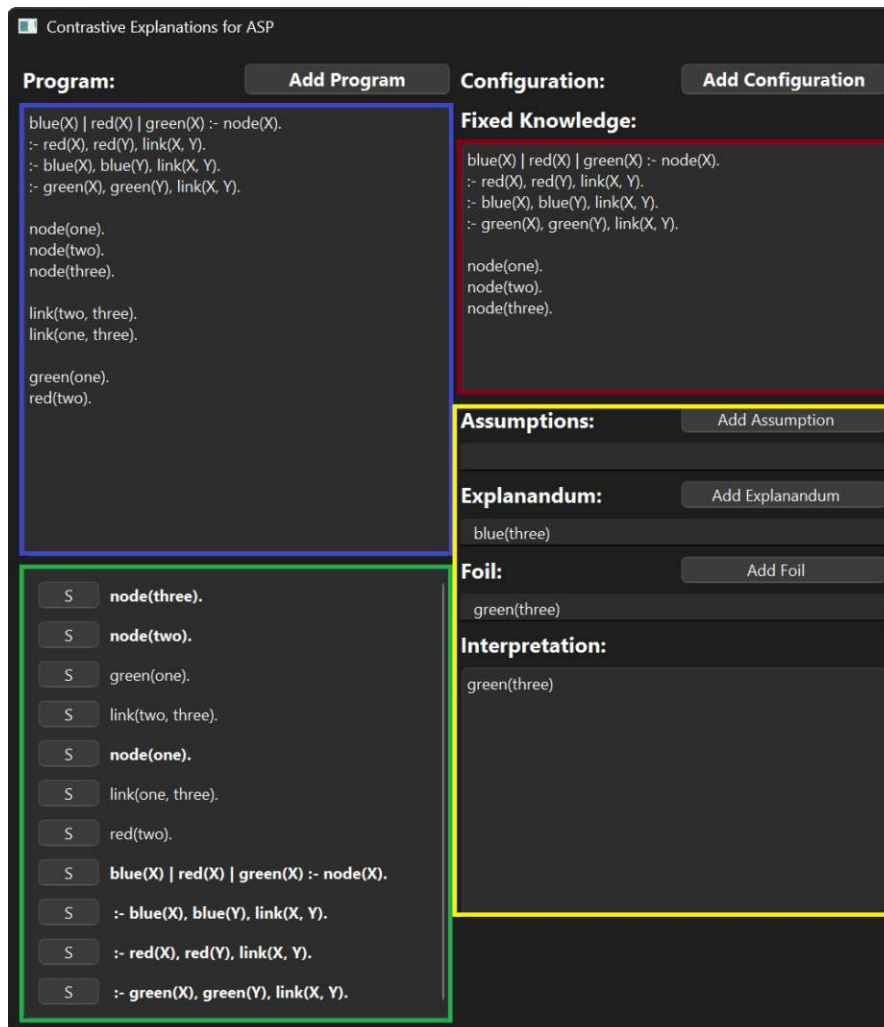


Figure 5.7: The interface for the 3-Coloring problem filled out containing non-ground rules

Since the user is not sure about how many contrastive explanations there are, they select ten counterfactual accounts. After clicking the **Go!** button, six different contrastive explanations appear in the section below. These are in sorted form:

```
(, , green(one).)
(, , link(one, three).)
(, , link(one, three). red(two).)
(, , green(one). red(two).)
(, , green(one). red(two). link(two, three).)
(, , link(one, three). link(two, three). red(two).)
```

The first two elements of each contrastive explanation are empty, because the disjunctive rule $\text{red}(X) \mid \text{blue}(X) \mid \text{green}(X) \text{ :- node}(X) .$ can derive both the explanandum and the foil. The rule has to be fixed, as it is part of the problem encoding. Thus, the third component of each explanation carries the relevant information. It outlines the minimal modifications to the program necessary to derive the foil instead of the explanandum. The contrastive explanations do not entail how each node has to be colored. This is also due to the rule $\text{red}(X) \mid \text{blue}(X) \mid \text{green}(X) \text{ :- node}(X) .$. Instead only facts about the current coloring, i.e. $\text{green}(\text{one}) .$, which need to be changed for a specific contrastive explanation, are shown. For example, in the first contrastive explanation, removing the fact $\text{green}(\text{one}) .$ is sufficient to allow the derivation of $\text{green}(\text{three}) .$ The precise new color of node one is irrelevant, provided it is no longer green.

The last contrastive explanation provides a more complex scenario. By removing the link between node one and node three, the first node can remain colored in green. In this variant green was chosen as the color for the second node, indicated by the fact $\text{red}(\text{two}) .$ in the third element of the contrastive explanation. Since two green nodes cannot be adjacent, the link between node two and node three has to be removed.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Related Work

This section surveys prior work relevant to the development of contrastive explanations for ASP, with a particular focus on two main areas. First, we examine existing methods for generating contrastive explanations in other reasoning frameworks. Second, we review existing debugging and explanation tools developed specifically for ASP.

6.1 Contrastive Explanations for Other Systems

Contrastive explanations have been applied to different systems and concepts. The first explicit formalization was proposed by Kean [Kea98]. His approach focused on propositional knowledge bases with classical semantics. Due to their monotonic nature, they are rather different from ASP. First, two logical sentences that entail the foil and explanandum are identified. Both sentences are reduced to the minimal set that still satisfies the entailment. A contrastive explanation after Kean is defined as the symmetric difference between these two minimal sets.

Miller [Mil21] gave a formalism of contrastive explanation for *structural causal models* [HP05, Hal15] by Halpern and Pearl. This concept is also not as expressive as ASP. Unlike the definition by Eiter et al. [EGO23], the internals of the models are not part of an explanation.

Ignatiev et al. [INAMS21] applied contrastive explanations to machine learning. However, they did not specify a foil. A contrastive explanation in this context was a minimal set of features that needed to change for the algorithm to produce a different prediction.

Contrastive explanations were also used for planning [KKM⁺21]. In this approach, instead of pointing out elements that need to change, a new plan was generated, which included the foil.

Abductive logic programming [KKT92] is a form of logic programming that computes explanations for certain observations. It does not include the definition of a foil.

Model-based diagnosis by Console et al. [CT91] uses abductive reasoning to find defective components in a system. Unlike contrastive explanations, the diagnosis creates a counterfactual setting with certain components removed to repair the system. The result does not include information about how the system is fixed, but only identifies the elements that need to change.

6.2 Debugging and Explainability Tools for ASP

Considering ASP, many systems have similar goals as contrastive explanation, cf. a survey by Fandinno and Schulz [FS19]. Explainability and debugging are concepts that try to help a programmer understand or fix an answer-set program. The key difference between contrastive explanation and previous work on explainability is that the latter provides justifications without specifying a foil. Debugging is used to find errors in the program, i.e., why there is no answer set or a wrong one. However, contrastive explanation is not necessarily only aimed at programmers or erroneous programs but also helps non-programmers understand why a correct program produced a certain answer set.

Mencía and Marques-Silva [MMS20] presented a way to apply algorithms for finding strong inconsistencies in monotonic logics to non-monotonic logics like ASP. In monotonic logics, if a sentence can be deduced from a set of sentences, it can also be deduced from a superset of these sentences. A strong inconsistency refers to a subset of a program of which all supersets are inconsistent.

Model reconciliation [SNVY21, NVSY20] is similar to contrastive explanation as it also tries to adapt a program so it entails a certain formula. However, the setup is quite different. In model reconciliation, the input is two programs and a formula, which is entailed by the first program but not the second. The goal is to modify the second program to entail the formula. Instead of a fact and a foil, only a formula is given.

A different approach for contrastive explanations for ASP was developed by Eiter et al. [EGHO23]. They implemented a contrastive explanation framework with ASP for *visual question answering*. However, this implementation was customized for this specific purpose, so it cannot be used for general cases.

Conclusion and Future Work

In this thesis, we investigated the practical feasibility of applying the theoretical approach of contrastive explanations to ASP. To this end, we created a naive grounder, which is suitable for contrastive explanations. Furthermore, we developed a new tool, which generates contrastive explanations, as well as a dedicated user interface.

The findings presented in Chapter 5 demonstrate that generating contrastive explanations using our developed tool is indeed practical and effective for small to medium-sized logic programs. This confirms the viability of our approach within constrained computational environments or with moderately complex inputs. However, as the size and complexity of the input programs increase, the performance of the system begins to degrade significantly due to the grounding bottleneck. This bottleneck becomes more pronounced with larger programs, leading to increased computational effort and longer processing times. The optimization of removing or altering rules that cannot contribute to the resulting answer set is a state-of-the-art procedure of modern ASP solvers. In the case of contrastive explanations, also those rules that do not affect the answer set for the explanandum, may still be needed for deriving the foil. This means that naive grounding without optimizations has to be performed.

In addition to the underlying program, a key contribution of this work is the design and implementation of a new user interface tailored specifically for contrastive explanations. This interface was developed to enhance usability and interpretability, allowing users to interact with the explanation system more intuitively. By presenting explanations in a clear and structured manner, the interface supports better user understanding of the differences between the explanandum and the foil, thereby improving the overall explanatory power of the system.

While the presented tool for generating contrastive explanations in Answer-set Programming (ASP) offers promising results, there remain several directions for future development and research. The tool could be improved in terms of functionality and

7. CONCLUSION AND FUTURE WORK

performance, which would broaden its applicability to more realistic and complex ASP programs.

A significant limitation of the current implementation is its restricted support for certain constructs commonly used in modern ASP. In practical applications, ASP programs often utilize language extensions such as conditional literals in choice atoms, aggregates, optimization statements, functions, and weight constraints among others. At present, our tool does not fully support these features, which constrains its usability in more expressive or industry-grade ASP encodings. Future work should focus on extending the explanation mechanism to accommodate these constructs.

Another key area for improvement is the system's performance scalability, which is currently limited by the well-known grounding bottleneck. As demonstrated in Chapter 5, the tool performs adequately for small and medium-sized programs, but runtime and memory usage escalate rapidly as program size increases. One promising avenue for mitigating this bottleneck lies in static analysis of the input program prior to grounding. By identifying parts of the program that are irrelevant to the derivation of the explanandum or the foil, the tool could avoid grounding and evaluating rules that have no semantic impact on the contrastive explanation.

List of Figures

3.1	Flowchart of the procedure to generate Counterfactual Accounts	18
3.2	User Interface Mockup	22
4.1	The empty user interface	35
4.2	The user interface showing a contrastive explanation	36
5.1	A solution for the 8-queens problem ¹	40
5.2	n-Queens Problem benchmark box plot	41
5.3	Sudoku benchmark box plot	45
5.4	5-Coloring Problem benchmark	47
5.5	User enters the first rule and marks it as fixed.	49
5.6	The interface is filled out and depicts a contrastive explanation.	50
5.7	The interface for the 3-Coloring problem filled out containing non-ground rules	52



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

5.1	n-Queens Problem benchmark table	42
5.2	Sudoku benchmark table	45
5.3	5-Coloring Problem benchmark table	48



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [ACC⁺13] Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. The fourth answer set programming competition: Preliminary report. In Pedro Cabalar and Tran Cao Son, editors, *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148, pages 42–53, Coruña, Spain, 2013. Springer.
- [BLR00] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.
- [CFG⁺20] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020.
- [CT91] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.
- [DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [EG95] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3):289–323, 1995.
- [EGHO23] Thomas Eiter, Tobias Geibinger, Nelson Higuera, and Johannes Oetsch. A logic-based approach to contrastive explainability for neurosymbolic visual question answering. In Edith Elkind, editor, *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023)*, pages 3668–3676, Macao, China, 2023. International Joint Conferences on Artificial Intelligence Organization.

- [EGO23] Thomas Eiter, Tobias Geibinger, and Johannes Oetsch. Contrastive explanations for answer-set programs. In Sarah Gaggl, Maria Vanina Martinez, and Magdalena Ortiz, editors, *Proceedings of the 18th European Conference on Logics in Artificial Intelligence (JELIA 2023)*, volume 14281 of *Lecture Notes in Computer Science*, pages 73–89, Dresden, Germany, 2023. Springer.
- [EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009 (RW 2009)*, volume 5689, pages 40–110, Brixen-Bressanone, Italy, 2009. Springer.
- [FS19] Jorge Fandinno and Claudia Schulz. Answering the “why” in answer set programming – A survey of explanation approaches. *Theory and Practice of Logic Programming*, 19(2):114–203, 2019.
- [GKK⁺16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5 (extended version), 2016. Available at <http://www.cs.uni-potsdam.de/wv/publications/>, [Online, accessed: July 24, 2024].
- [GKK⁺19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. Potassco user guide, version 2.2.0 (2019), 2019. Available at <https://github.com/potassco/guide/releases/tag/v2.2.0>, [Online, accessed: July 24, 2024].
- [GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.
- [GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of the 5th International Logic Programming Conference and Symposium*, pages 1070–1080, Cambridge, MA, 1988. MIT Press.
- [Hal15] Joseph Y. Halpern. A modification of the halpern-pearl definition of causality. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 3022–3033, Buenos Aires, Argentina, 2015. AAAI Press.
- [HLY14] Amelia Harrison, Vladimir Lifschitz, and Fangkai Yang. The semantics of gringo and infinitary propositional formulas. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014)*, pages 32–41, Vienna, Austria, 2014. AAAI Press.

- [HP05] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach. Part II: Explanations. *The British Journal for the Philosophy of Science*, 56(4):889–911, 2005.
- [INAMS21] Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and Joao Marques-Silva. From contrastive to abductive explanations and back again. In Matteo Baldoni and Stefania Bandini, editors, *AIxIA 2020 – Advances in Artificial Intelligence (AIxIA 2020)*, volume 12414, pages 335–355. Springer, 2021.
- [Kea98] Alex Kean. A characterization of contrastive explanations computation. In Hing-Yan Lee and Hiroshi Motoda, editors, *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI 1998)*, volume 1531, pages 599–610, Singapore, 1998. Springer.
- [KKM⁺21] Benjamin Krarup, Senka Krivic, Daniele Magazzeni, Derek Long, Michael Cashmore, and David Smith. Contrastive explanations of plans through model restrictions. *Journal of Artificial Intelligence Research*, 72:533–612, 2021.
- [KKT92] Antonis Kakas, Robert Kowalski, and Francesca Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1992.
- [KLPS16] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.
- [KRSW23] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. How to build your own ASP-based system?! *Theory and Practice of Logic Programming*, 23(1):299–361, 2023.
- [Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.
- [Lip90] Peter Lipton. Contrastive explanation. *Royal Institute of Philosophy Supplement*, 27:247–266, 1990.
- [LPF⁺06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
- [Mil19] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.
- [Mil21] Tim Miller. Contrastive explanation: A structural-model approach. *The Knowledge Engineering Review*, 36:e14, 2021.

- [MMS20] Carlos Mencía and Joao Marques-Silva. Reasoning about strong inconsistency in ASP. In Luca Pulina and Martina Seidl, editors, *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing — SAT 2020*, volume 12178, pages 332–342, Alghero, Italy, 2020. Springer.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC 2001)*, pages 530–535, New York, NY, USA, 2001. Association for Computing Machinery (ACM).
- [NVSY20] Van Nguyen, Stylianos Loukas Vasileiou, Tran Cao Son, and William Yeoh. Explainable planning using answer set programming. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020)*, pages 662–666, Rhodes, Greece, 2020. International Joint Conferences on Artificial Intelligence Organization.
- [Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.
- [Sim99] Patrik Simons. Extending the stable model semantics with more expressive rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1999)*, volume 1730 of *Lecture Notes in Computer Science*, pages 305–316, El Paso, Texas, 1999. Springer.
- [SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1):181–234, 2002.
- [SNVY21] Tran Cao Son, Van Nguyen, Stylianos Loukas Vasileiou, and William Yeoh. Model reconciliation in logic programs. In Wolfgang Faber, Gerhard Friedrich, Martin Gebser, and Michael Morak, editors, *Proceedings of the 17th European Conference on Logics in Artificial Intelligence (JELIA 2021)*, volume 12678, pages 393–406. Springer, 2021.
- [Tem88] Dennis Temple. Discussion: The contrast theory of why-questions. *Philosophy of Science*, 55(1):141–151, 1988.
- [Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.
- [WEZL23] Yisong Wang, Thomas Eiter, Yuanlin Zhang, and Fangzhen Lin. Witnesses for answer sets of logic programs. *ACM Transactions on Computational Logic*, 24(2):15:1–15:46, 2023.