(EMCL)
European Master's Program in Computational Logic
Faculty of Computer Science
Technische Universität Dresden

Navigation Approaches for Answer Sets

Thesis Submission for a
Master of Science in
Computer Science

Asmaa Afeefi
Defense on
December, 2015

Supervisors:
Prof. Dr. rer. nat. Sebastian Rudolph
Dr. Sarah Gaggl

To My Mother and My Father

for all their enormous help and sipirtual support,
gone now but never forgotten.
May Allah grant them the highest level of Paradise.

Without their encouragement, I would never be here.

Declaration

Author	:	Asmaa Afeefi
Matrikel-Nr	:	4020979
Title	:	Navigation Approaches for Answer Sets
Degree	:	Master of Science
Date of Submission	:	4th of December 2015

I hereby declare that this thesis is my own work, only with the help of the referenced literature and under the careful supervision of my thesis supervisor.

Dresden, Germany, December, 2015 Asmaa Afeefi

Acknowledgements

First of all, I express my greatest and deepest gratitude to Allah for providing me all the blessings to complete this work. I am also very thankful to:

- Dr. Gaggl and Prof. Rudolph for supervising and encouraging me in completing my thesis work. I also thank Dr. Gaggl for her concern and motivation which contributed greatly to improve my English and helped me during the writing of my thesis.
- Prof. Hölldobler for his excellent support, guidance, encouragement during my study.
- My fiancé, Ashraf Imraish, for his countless prayers, unconditional support, care, and encouragement. Not only that, he also motivated me to pursue my master degree in this field.
- My wonderful sisters and brother, Areej Afeefi, Inas Afeefi, and Amjad Afeefi, who have always been patient and heartedly supported me, listened to me and given good advice, in my life and my study.
- Amna Dridi for all her big help from the beginning up to the very end of my study, who has always motivated, encouraged, and spiritually supported me to finish this program.
- Ishraq Jarrar for the funny meetings, support, and cheerfulness, especially when I had a difficult time.
- Ms. Ghada Shubietah, Ms. Dana Adas, Nazifa Karima, and Sarah Kohail for their time and patience to read my thesis, comment and correct my English.
- Lukas Schweizer for his help during my thesis.
- Adrian Nuradiansyah for his help and support me during my thesis.
- Emira Ziberi for her spiritual support, help, and cheerfulness.
- Alifah Syamsiyah, Nur Alfi, Tanu Shre Sahu, and all of my EMCL friends for the nice moments in these two years.
- Nazmya Zoghayer, Sabrin Imraish, and Hijazi Imraish, for their continuous prayers and support.
- Dr. Huda Errabti for her countless prayers and cheerfulness, especially when I had a difficult time.
- European Master in Computational Logic program and its director, Prof. Hölldobler, for their support and the European Commission for their Erasmus Mundus scholarship to support my study.

Abstract

Answer set programming (ASP) is one of the most popular modeling languages in knowledge representation. In recent years, many integrated development environments (IDE) for ASP programs including editors and debuggers are developed. However, none of them focuses on analyzing the answer sets. With the availability of a huge number of answer sets, it is increasingly important to provide a solution to navigate them. We study and analyze the answer sets to perform the user access to specific answers. To this end, we aim at conducting and exploring different navigation approaches, such as, filtering, sorting, finding diverse/similar solutions, and faceted browsing. Afterward, we implement a tool performing the above approaches in order to simplify the search task. We conclude by testing the performance of the proposed tool into two different real world examples of ASP programs.

Keywords: Answer set programming, navigation approaches, filtering, soring, diversity, similarity

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to the thesis. First, we introduce the motivation of the work in Section 1.1. Following in Section 1.2, the problem statements are presented. Then, a running example is given in Section 1.3. Next, the related work is reviewed in Section 1.4 and our contribution is highlighted in Section 1.5. Finally, the thesis structure is shown in Section 1.6.

1.1 Motivation

Answer set programming (ASP) [2] is a fully declarative programming paradigm, rooted in logic programming and non-monotonic reasoning. ASP is one of the most popular modeling language in Knowledge Representation and Reasoning (KRR). The main idea of answer set programming is to represent a problem at hand by a logic program, such that its answer sets corresponds to solutions. These solutions characterize the solutions of the original problem, and then, use an asnwer set solver to find such solutions [27]. In the last few years, many solvers are developed, such as, Clasp (in conjunction with Gringo) [17, 18], DLV [26], Clingo[1] [15], and SMODELS[2] [30].

On one hand, the researchers have turned their attention to develop different integrated development environment (IDE) for ASP programs including editors and debuggers (e.g., APE [31], iGROM[3], and SeaLion [6]). On the other hand, some of them have developed tools to visualize the answer sets and their relations by means of a directed graph, such as, ARVi tool [1].

Despite these improvements, a lack attention to analyze the answer sets themselves. In some particular problems, a massive amount of answer sets could be available. However, the user is not interested in all of them. Thus, a navigation of the search space could be a solution to help the user to access the specific answer sets. To this end, we are looking into different navigation methods, such as, filtering, sorting, finding diverse/similar solutions, and faceted browsing. The intuition behind these navigation methods is to make the search faster and explore information that is related to the user's query.

[1] `http://sourceforge.net/projects/potassco/files/clingo/`
[2] `http://www.tcs.hut.fi/Software/smodels/`
[3] `http://igrom.sourceforge.net/`

1.2 Problem Statement

ASP can profitably be used for real-world applications. Some applications in ASP, either real or not real have a vast amount of answer sets. The analysis of a huge number of answer sets is a worthwhile issue. Answer sets consist of atoms that are true (rendering the missing ones false). For example, an ASP program about *chair configuration* has many configurations. Each answer set represents a chair. The chair products have many properties, like name, category, type, price, location and so on. Each answer set has many atoms corresponding to the properties of the products. To get the desired answer sets with specific atoms, one needs to study the navigation approaches of the search space of the answer sets.

As mentioned in the previous section, the lack of suitable tools for navigating the search space of the answer sets, push us to launch a tool to solve the problems. This thesis is proposed to solve the following problems:

- How to filter the answer sets depending on existence or absence of particular atoms?
- How to sort the answer sets corresponding to cardinality, arity, alphabetically?
- How to find similar/diverse solutions (answer sets)?
- How to combine aforementioned issues?
- How to implement a tool with the previous issues such that a user-friendly interface?

1.3 Running Example

In this section, we provide the running example called *chair configuration*. Chair configuration is a real-world application that users usually need to find good solutions with respect to some well-described criterion. Additionally, the users may want to examine only a few solutions to pick one that matches their preferences.

Problem Description

Consider, for instance, a variation of features of the example about buying a chair. Suppose there is a product advisor that asks the users about their preferences/constraints about a chair, and then lists the available ones that match their preferences/constraints. As an illustration, there are many attributes that describe a chair, such as, color, size, parts of the chair (e.g., some of them contain legs and mid rails, and others contain swivel base and wheels), design style (e.g., traditional), and so on. However, such a list of features may be too long. In that case, the user might ask for a particular chair with specific features. Thus, chair configurations need to study which features suit each other.

ASP Program

The example is implemented in ASP to generate different configurations of a chair. The input of the program are the attributes, the features, and the parts of the chair. To accomplish this task, several rules and constraints on these properties have to be satisfied. Then, the

program is run with an ASP solver to find such solutions. Finally, we have a huge number of answer sets correspond to solutions (chair configurations).

In real chair configuration, there are a huge number of the chairs. Similarly, we implement an ASP program (*chair configuration*) such that each answer set represents a chair. More precisely, Each answer set consists of many atoms which are the attributes of the chairs. The example below shows the facts of the *chair* which is taken from the program *chair configuration*. The program is included in Appendix A.

```
part(backrest).
part2(armrest).
part2(chair_seat).
col(white).
size(small).
property1(single).
%type armrest
type1(acryclic).
%type for chair seat
type2(acryclic).
%type for legs and mid rail
type3(metal).
weight(light).

name(ingolf).
place2(ingolf,kitchen).
place2(ingolf,living_room).
has_price(ingolf,33).
```

Through this example, we have a product *chair* with many attributes, such as, the parts of the chair (`part(backrest)`, `part2(armrest)`), the color (`col(white)`), the size (`size(small)`), the type of the chair seat is (`acryclic`), the place where the chair is put in either (`living_room`) or (`kitchen`), its name (`ingolf`), the price (`33`), and so on. We add many facts to increase the search space. Execution the program with many facts by an ASP solver give us many answer sets. We introduce below a sample of the answer sets after running the *chair configuration* program.

```
Answer: 1
part(backrest) part(action_mechanism,swivel_base,wheels) chair(gamleby)
traditional place(living_room) color(white) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(33)
Answer: 2
part(backrest) part_type(mid_rail,acryclic) part_type(legs,acryclic)
place(kitchen) place(dining_room) chair(tobias) modern color(red)
property(medium,heavy) property(single) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) price(80) part(legs,mid_rail)
Answer: 3
part(backrest) part_type(legs,metal) part_type(mid_rail,metal)
place(kitchen) place(dining_room) place(garden) chair(ingolf) modern
place(living_room) color(white) property(small,heavy) property(single)
```

```
part_type(armrest,acryclic) part_type(chair_seat,acryclic) price(20)
part(legs,mid_rail)
```

However, usually there are too many answer sets of *chair configuration* example computed by an ASP solver. The user needs to compare these answer sets, by analyzing the similar/diverse ones with respect to some distance measure, to pick the most plausible ones, or by filtering and sorting them to get good configurations that suit his preferences. To this end, we use this example in the remaining chapters to show how to navigate the search space of the answer sets and apply our approaches on them.

1.4 Related Work

Analysis of answer-set programming (ASP) is one wide field that is increasingly growing in the last few years. At first, different tools for developing ASP programs have been proposed including editors and debuggers. In [31], Sureshkumar et al. implement an Integrated Development Environment (IDE) for ASP, the AnsProlog* Programming Environment (APE). It offers many features, like syntax highlighting, automatic syntax checking, integration of editor; LPARSE and SMODELs, and display dependency graph of program.

Other IDEs for ASP have been developed in the last few years. Febbraro et al. present in [13] ASPIDE which is a comprehensive IDE for ASP. It integrates composition features, like dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc. ASPIDE has more features than APE such as visual editor, integration with databases, and user friendly result visualization.

Koziarkiewics implements iGROM[4] which is an IDE for ASP programs specifically those written in DLV (and its frontends) and Smodels. It provides some features, such as, syntax highlighting for DLV and its dialects, error detection for DLV and dialects.

Recently, Oetsch et al. in [6] have designed an IDE for ASP (SeaLion) as a plug-in for Eclipse platform. This tool provides source-code editors for the languages of Gringo and DLV. It offers functionalities, like syntax highlighting, syntax checking, code completion, visual program outline, and refactoring functionality. It also implements a stepping-based debugging approach. SeaLion is the first IDE for ASP programs that provides debugging features. It supports ASP development with extended UML class diagrams and visualization of answer sets in corresponding instance diagrams.

Another tool for visualizing the answer sets is ASPViz. Cliffe et al. [8] introduce a graphical representation tool that enables the users and ASP programmers to visualize answer sets using the declarative nature of ASP itself to produce graphical representation of solutions. Their approach is based on visualizing the answer sets of a given program depending on a second program that contains the visualization information specified via predefined predicates. Unfortunately, this approach will not always be successful, because only some programs have a natural graphical representation.

As we mentioned above, several tools have been designed to support the user in developing ASP applications, and the visualization aspects of these tools focus on the representation of single answer sets. In other words, explicit treatment of one of the main aspects of ASP, multiple solutions, has received less attention within these tools. Ambroz et al. [1] present a

[4] http://igrom.sourceforge.net/

new tool, ARVis. The main purpose of ARVis is to visualize answer sets and their relations by means of a directed graph. The general idea for this tool is passing the answer sets of a first user-specified ASP encoding to a second user-specified encoding which specified the relations between them. Obviously, ARVis is not designed to obtain a high performance since a potential exponential number of answer sets of the first program has to be processed by the second one.

Eiter et al. in [11] introduce offline and online methods to find similar or diverse solutions of a given problem in answer set programming in phylogeny reconstruction. They study two kinds of computational problems related to finding similar/diverse solutions of a given problem, in the context of ASP: one problem asks for a set of n solutions that are k-similar (resp. k-diverse), the other one asks for a solution that is k-close (k-distant) to a given set of solutions. They analyze the computational complexity of the given problem to compute similar/diverse solutions. Furthermore, they modify the search algorithm of an ASP solver (CLASP) to implement one of the online methods.

As discussed above, some approaches are much more focused on editing and debugging ASP programs. Others are developed for a certain problem. To the best of our knowledge there does not exist a tool yet that is capable of navigating the space of answer sets for general problems.

1.5 Contributions

This work provides both theoretical and practical values, which can be summarized as follows:

- We lay the foundation and develop a suitable engineering tool for navigating the space of answer sets of any ASP program so-called NavAS (Navigation Approaches for Answer Sets).

- We extract the answer sets from running an ASP solver. Also, we provide a specific representation of answer sets and thus, make them ready to use in practice and easy to deal with. For example, we provide filtering options for the answer sets depending on the existence or absence of one or several ground instances.

- We provide several ways of sorting the answer sets for a given program. For instance, we basically enumerate the answer sets depending on the cardinality, and order the answer set itself by arity or alphabetically.

- We internally present the answer sets by means of an undirected graph and use an appropriate distance such as Hamming distance or Jaccard distance for edges of the graph to find the similar/diverse solutions.

- We implement faceted browsing technique to make NavAS easier for the users to quickly spot the most relevant answer sets. With faceting, answer sets are grouped under common features.

- We also make a comprehensive evaluation of the performance of two different examples of ASP programs.

1.6 Thesis Structure

The remainder of this thesis is structured as follows. We begin with the preliminaries for the notion and terminology we use through the thesis in Chapter 2. Chapter 3 provides the methods for navigating the space of answer sets. Following in Chapter 4, we give a description for NavAS system. Next, the evaluation results of the proposed tool are shown in Chapter 5. Finally, we summarize the thesis with the conclusion and point to some future work in Chapter 6.

Chapter 2

Preliminaries

The chapter starts by providing some terminology for essential logical concepts. We continue with the definition of the syntax of ASP programs, followed by the grounding and how to obtain the grounded version of a program. Then, we introduce the semantics for ASP programs. Finally, we give an overview of the complexity of ASP programs. The content of this chapter is based on material from [23, 16].

2.1 Logical Preliminaries and Terminology

Answer set programming is built from a language $\mathcal{L}$ composed of predicate and variable symbols as basic building blocks. We usually denote:

- *predicate* symbols by lower-case letters, such as $a, b, ...$ for predicates of zero arity (also called propositions), and otherwise use $p, q, ...$ or strings starting with a lower-case letter, such as $hasMother$ and
- *variable* symbols by upper-case letters, such as X, Y, Z, or strings starting with an upper-case letter, like $Mother$.

A *Term* is a variable or a constant. The *constant* is denoted by a symbol or a string starting with a lower-case letter. An *atom* is an expression of the form $p(t_1, ..., t_n)$ where p is a predicate of arity n and $t_1, ..., t_n$ are terms. An atom is called *ground* if it does not contain any variable. A *literal* is either an atom a or a negated atom $\neg a$ (called classically or strongly negated literal). An (extended) literal is either a literal or an expression of the form *not* l where l is a literal and *not* is called negation-as-failure. An (extended) literal is called ground if its underlying atom is ground.

For a set of literals L, we usually denote

- $\neg L$ with the set $\{ \neg l \mid l \in L\}$,
- *not* L with the set $\{$ *not* $l \mid l \in L\}$,
- L^+ with the positive part of L, i.e. $L^+ = \{ a \in L \mid$ a is an atom$\}$, and
- L^- with the set of literals underlying negation-as-failure literals, i.e. $L^- = \{ l \mid$ *not* $l \in L \}$.

We say that L is *consistent* for a set of ground literals if and only if $L \cap \neg L = \emptyset$.

Definition 2.1.1 (Ground Instance)**.** A *ground instance* of an atom is obtained by replacing all of its variables by ground terms. *Ground atoms* are identified with propositions and we denote them by lower-case letters.

Definition 2.1.2 (Herbrand Universe)**.** The *Herbrand universe* $\mathcal{U}_\mathcal{L}$ for the language $\mathcal{L}$ is the set of all ground terms which can be formed from constants appearing in the language.

Definition 2.1.3 (Herbrand Base)**.** The *Herbrand base* $\mathcal{B}_\mathcal{L}$ for the language $\mathcal{L}$ is the set of all ground atoms which can be formed by using predicate symbols from the language with ground terms from $\mathcal{U}_\mathcal{L}$ as arguments.

2.2 Syntax of ASP Programs

In logic, syntax is concerned with the rules in a formal language which are used for constructing the program without regards to any meaning given to them. We give a formal introduction to propositional logic programs. A *propositional normal logic program* over a set $\mathcal{A}$ of literals consists of a finite set of normal rules.

Definition 2.2.1 (Normal Rule)**.** A *normal rule* r is an expression of the form

$$a \leftarrow b_0, ..., b_m, not\ b_{m+1}, ..., not\ b_n$$

where $0 \leq m \leq n$ and each $b_i \in \mathcal{A}$ is a literal for $0 \leq i \leq n$.

The left-hand side a is called the *head* of the rule, denoted *head(r)*, whereas the right-hand side $b_1, ..., b_m, not\ b_{m+1}, ..., not\ b_n$ is called the *body* of the rule, denoted *body(r)*. For *body(r)*, we then have that $body(r)^+ = \{b_1, ..., b_m\}$ and $body(r)^- = \{b_{m+1}, ..., b_n\}$.

There is another rule type extending the idea of a normal rule. We call such a rule *disjunctive*, if the head of a rule contains a disjunction of atoms, such as, $a_0 \mid ... \mid a_l$ where $0 \leq i \leq l$.

Rules can be classified depending on conditions satisfied by the *head(r)* or/and *body(r)*:

- A *constraint* is a rule where the *head(r)* is empty.
- A *fact* is a rule where the *body(r)* is empty.
- A *positive rule* is a rule where $body(r)^- = \emptyset$, $body(r)^+ = body(r)$ and *head(r)* is either an atom or empty.

Definition 2.2.2 (ASP Program)**.** An *answer set program* is a countable set of rules.

Example 2.2.1. *Consider the following program* P_1*:*

$$\begin{array}{lll} a & \leftarrow & & (2.1) \\ c & \leftarrow & not\ b, not\ d & (2.2) \\ d & \leftarrow & a & (2.3) \end{array}$$

In this program, Rule (2.1) is a fact. Rule (2.3) is a positive rule with no negation-as-failure literals. There are two negation-as-failure literals in the Rule (2.2).

Using the types of rules introduced above we can consider the following types of programs:

- A *positive program* contains only positive rules.
- A *normal program* consists of normal rules.
- A *disjunctive program* contains disjunctive rules.

A positive, normal program or disjunctive program is called *constraint-free* if it does not contain any constraints.

There are several language constructs which enrich the core language of ASP [16].

2.3 Grounding

In the formulation of the semantics of ASP programs, we consider the programs without variables. In this section, we explain how we can obtain a grounded version from a normal program P that contains variables.

Definition 2.3.1 (Van Nieuwenborgh [23])**.** Let P be a program. The set of all constants appearing in P is called the Herbrand universe, denoted by $\mathcal{U}_P$. The Herbrand base $\mathcal{B}_P$ of P is the set containing all grounded atoms that can be constructed from the predicates in P and the terms in $\mathcal{U}_P$.

Let r be a rule in the program P. A grounded instance of r is any rule obtained from r by substituting every variable X in r by $\sigma(X)$, where σ is a mapping from the variables to the terms in $\mathcal{U}_P$. The grounded program P is defined as the set of all ground instances of a rule $r \in P$ by $gnd_{\mathcal{U}_P}(r)$, i.e. $gnd(P) = \bigcup_{r \in P} gnd_{\mathcal{U}_P}(r)$.

Example 2.3.1. *Consider the following program* P_2

$$
\begin{array}{lcl}
q(X) & \leftarrow & r(X) \\
r(a) & \leftarrow & \\
r(b) & \leftarrow & \\
r(c) & \leftarrow &
\end{array}
$$

The following is $gnd(P_2)$:

$$
\begin{array}{lcl}
q(a) & \leftarrow & r(a) \\
q(a) & \leftarrow & r(b) \\
q(a) & \leftarrow & r(c) \\
q(b) & \leftarrow & r(a)
\end{array}
\qquad
\begin{array}{lcl}
q(b) & \leftarrow & r(b) \\
q(b) & \leftarrow & r(c) \\
q(c) & \leftarrow & r(a) \\
q(c) & \leftarrow & r(b)
\end{array}
\qquad
\begin{array}{lcl}
q(c) & \leftarrow & r(c) \\
r(a) & \leftarrow & \\
r(b) & \leftarrow & \\
r(c) & \leftarrow &
\end{array}
$$

The size of the grounding can be exponential in the size of the program. For the previous example, the full grounding amounts to just four rules, as repeated literals in the bodies of the rules, can be eliminated without affecting the answer sets. Intelligent grounding

techniques incorporate such equivalences and many further optimizations. Therefore, the researchers have turned their attention to study more efficient and intelligent grounding methods [19].

The example for the intelligent grounding of the previous program is as follows:

$$\begin{array}{lll} q(a) & \leftarrow & r(a) \\ q(b) & \leftarrow & r(b) \\ q(c) & \leftarrow & r(c) \\ r(a) & \leftarrow & \\ r(b) & \leftarrow & \\ r(c) & \leftarrow & \end{array}$$

2.4 Semantics of ASP Programs

The semantics of ASP programs is concerned with the meaning of the language (syntax) of the program. Intuitively, modeling a certain problem with an ASP program, means we want the semantics of the program to capture the knowledge from a language notion.

In general, the semantics of a program is represented by interpretations.

Definition 2.4.1 (Interpretation)**.** An *interpretation I* is a mapping from all ground atoms to the set of truth values {*true, false*}. A *Herbrand interpretation* consists of a non-empty domain where the domain is a set of ground terms.

We define an interpretation I as a subset of $\mathcal{B}_P$ with the understanding that all atoms which are in the set are considered true, while all the others which are not, are considered false. For example, the interpretation $\{p(1) \rightarrowtail \mathit{true}, p(2) \rightarrowtail \mathit{false}, a \rightarrowtail \mathit{true}\}$ is represented by $\{p(1), a\}$.

We say that the rule $r \in P$ is *satisfied* by I, denoted by $I \models r$, iff $\mathit{head}(r) \cap I \neq \emptyset$ whenever $\mathit{body}^+(r) \subseteq I$, $\mathit{body}^-(r) \cap I = \emptyset$.

Definition 2.4.2 (Model)**.** Let I be an interpretation of the program P. I is a *model* of P, denoted by $I \models P$ iff each $r \in P$ is satisfied by I.

Definition 2.4.3 (Ordering Among Interpretations)**.** Let $\mathcal{I}$ be a collection of interpretations. An interpretation $I \in \mathcal{I}$ is called *minimal* in $\mathcal{I}$ if and only if there is no interpretation $J \in \mathcal{I}$ such that $J \subset I$. An interpretation I is called *least* in $\mathcal{I}$ if and only if $I \subseteq J$ for any interpretation $J \in \mathcal{I}$. A model of a program P is called minimal (resp. least) if it is minimal (resp. least) among all models of P.

Definition 2.4.4 (Gelfond and Lifschitz [21])**.** Let P be a normal program and I an interpretation of P. The *reduct* P^X of P relative to a set X of atoms is

$$P^X = \{\mathit{head}(r) \leftarrow \mathit{body}(r)^+ \mid r \in P \, , \, \mathit{body}(r)^- \cap X = \emptyset\}.$$

We aim to find the stable model of P which is represented by X. In other words, X is the stable model of P, if X is the $\subseteq$- minimal model of P^X.

Now, the question is raised how can we compute the stable model. We do this in two steps. The reduct P^X of a program P is obtained by:

- removing all rules containing a negation-as-failure literal *not l* in its body with $l \in X$ and then
- deleting all negation-as-failure literals of the form *not l* in the bodies of the remaining rules.

Let us use the previous steps to show that a logic program can have zero, one, or multiple answer sets (stable models).

Example 2.4.1. *Consider program P_3 as follows*

$$\begin{array}{lll} a & \leftarrow & not\ b \\ b & \leftarrow & not\ a \end{array}$$

The program P_3 has two answer sets: $A_1 = \{a\}$ and $A_2 = \{b\}$.

Example 2.4.2. *Consider the following program P_4*

$$a \leftarrow not\ a$$

This program does not have any answer set, although it has a single model containing a.

Example 2.4.3. *Consider the following program P_5 with non-ground atoms*

$$\begin{array}{lll} p(a) & \leftarrow & \\ r(X) & \leftarrow & p(X), not\ q(X) \end{array}$$

The minimal models of P_5 are

$$M_1 = \{p(a)\ ,\ q(a)\}\ and\ M_2 = \{p(a)\ ,\ r(a)\}$$

Computing the reducts $P_5^{M_1}$, $P_5^{M_2}$ gives us
$P_5^{M_1}$:

$$p(a) \leftarrow$$

$P_5^{M_2}$:

$$\begin{array}{lll} p(a) & \leftarrow & \\ r(a) & \leftarrow & p(a) \end{array}$$

It is easy to show that M_1 is not a stable model of the program P_5. The minimal model M_2 is an answer set (stable model) of P_5.

2.5 Computational Complexity

As for ASP and model computation, we are interested in the following decision problems [3]:

- **Existence**: Decide whether there is an answer set.
- **Set-membership**: Decide whether there is an answer set containing a specific atom.

- **Set-entailment**: Decide whether a specific atom is in all the answer sets.

We give a summary of the complexity of these decision problems in Table 2.1. Recall that $\prod_2^P = co\,\sum_2^P$, where $\sum_2^P$ is the class of problems that can be solved on a non-deterministic machine using an NP oracle in polynomial time. The following complexity results apply to propositional programs only.

Table 2.1: Complexity of ASP programs [12]

	Existence	Set-membership	Set-entailment
Normal	NP-complete	NP-complete	$coNP$-complete
Disjunctive	$\sum_2^P$-complete	$\sum_2^P$-complete	$\prod_2^P$-complete

Chapter 3

Methods for Navigating the Space of Answer Sets

In some particular problems, an enormous number of answer sets could be available. However, the user is not interested in all of them. Thus, a navigation of the search space could be a solution to help the user to access the specific answer sets. To this end, we are looking into different navigation methods, such as, filtering, sorting, finding diverse/similar solutions, and faceted browsing. The thinking behind these navigation methods is to make the search faster and explore information that is related to the user's query. In this chapter, we start with the data setting and how the answer sets are stored. Next, we discuss the procedures and techniques that are already operational in our work. We begin with filtering procedure and continue with sorting. Next, we study finding similar/diverse solutions in answer sets. Finally, we combine all these methods in the so-called faceted browsing.

3.1 Data Setting

Answer sets are taken after running an ASP solver and then stored in form of structured data, such as, a relational database. For our purpose, we use MySQL Relational Database Management (RDBMS). In this work, we build a table consisting of three columns: ID, Answer set, and Cardinality, respectively. Table 3.1 provides an example of such a database table. Our intuition behind building a table with three columns is to generalize our work and make it suitable for any ASP program. Note that building a table with n columns for n atoms in an answer set is not a good choice, as the number of atoms in answer sets might vary for a program, as well as it might be a very large number. In addition, the ground atoms are extracted from the answer sets and represented in term of vectors. These vectors are, then, used to compute the distance between the answer sets to find similar/diverse solutions.

3.2 Filtering

The first procedure in navigating the answer sets is filtering. Filtering is a key word search that means "show me only the data which contains a specific term". In ASP, however, filtering means "show me all the answer sets which contain a specific atom". The motivation

Table 3.1: A sample of database table of *chair configuration* example

AnswerSets

ID	AnswerSet	Cardinality
1	part(backrest) part_type(legs,metal) part_type(mid_rail,metal) place(kitchen) place(dining_room) place(garden) chair(ingolf) modern place(livin_room) color(white) property(single) part_type(armrest,acryclic) part_type(chair_seat,acryclic) property(small,heavy) price(20)	15
2	part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby) traditional place(living_room) color(white) property(small,light) property(triple) part_type(armrest,wooden) part_type(chair_seat,leather) price(70)	11

for using filters is that they reduce the massive amounts of data to a much smaller significant subset, which represents the answers most relevant to the user query.

Filtering the answer sets according to their atoms is displaying only the answer sets with the atoms that the user wants to see. The user can view the answer sets with the existence or absence of a specific atom. A filter allows the user to do so quickly and easily. In this work, we filter the answer sets which some of them might not contain a specific atom. Accordingly, the user has the ability to show the answer sets with or without a specific atom. Additionally, if the atom contains one parameter and this parameter is a numerical value, then the user can type the range of the numerical values. Thus, the answer sets are ordered ascending depending on the numerical values.

We implement the filtering by using a full-text search (FTS). MySQL supports indexing and re-indexing data in the full-text search. For the full-text search, we use a *select* SQL query with *match* and *against* keywords. There is a *like* operator that is used for the filtering (or search) in MySQL. The *like* operator forces MySQL to scan the whole table to find the matching rows. Therefore, it does not allow the database engine to use the index for fast searching. As a result, the performance of the query that uses the *like* operator degrades when we have huge data. Thus, we use a *select* SQL query with the full-text search to filter the answer sets that the user specifies.

At first, the user specifies the predicates (or/and constants) that are contained or not contained in the desired answer sets. We combine all these selected predicates (or/and constants) as two vectors. One vector is used for the predicates (or/and constants) that are contained in the answer sets. The other one is used for the predicates (or/and constants) that are not contained in the answer sets. Then, we pass these two vectors as a term to a *select* SQL query with the *match* and *against* keywords. The SQL query below shows the filtering of the answer sets.

```
ALTER TABLE AnswerSets ADD FULLTEXT(AnswerSet)
```

```
"SELECT * FROM AnswerSets"
 + "WHERE MATCH(AnswerSet)"
 + "AGAINST('"+predicates_contained+"' '"+predicates_not_contained+"'"
 + "IN BOOLEAN MODE)";
```

The time complexity of filtering which represents by full-text search is $O(n)$ where n

is the number of answer sets (the number of records in the database). In ASP programs, we can use the built-in predicates such as *#show* to only output specific atoms, but here the input of the filtering is presented in a more comfortable way.

Going back to our example (chair configuration), if the user needs to see all the answer sets AS such that `traditional` $\in AS$, and `modern` $\notin AS$. The SQL query will be as follows.

```
"SELECT * FROM AnswerSets"
 + "AGAINST ('+traditional* -modern*' "
 + "IN BOOLEAN MODE)";
```

A sample of the answer sets is shown below.

```
Answer1
part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby)
traditional place(living_room) color(white) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(70)
Answer2
part(backrest) part(action_mechanism,swivel_base,wheels) color(white)
chair(kaustby) place(living_room) property(small,light) property(triple)
traditional part_type(armrest,wooden) part_type(chair_seat,leather)
price(40)
Answer3
part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby)
property(small,light) place(living_room) color(white) traditional
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(66)
```

We provide in Section 3.4 the option of showing the answer sets with specific ground instance.

3.3 Sorting

Arranging the data in ordered sequences (ordered sets) is a common approach in many applications. It is useful when one is looking for one or several properties of the data. Sorting arranges the data alphabetically or numerically in ascending or descending order. First, we introduce the ordered sets basics. Then, we explain the types of sorting.

3.3.1 Ordered Sets Basics

Definition 3.3.1 (Binary Relation)**.** A *binary relation* R between two sets L and M is a set of pairs $(l, m) \in R$ (or lRm) with $l \in L$ and $m \in M$. R^{-1} is an inverse relation to R between L and M denoted by $mR^{-1}l \Leftrightarrow lRm$.

Definition 3.3.2 (Total Order)**.** A binary relation R on a set L is called a *total order* (*totally ordered set*, or *linearly ordered set*) that satisfies the conditions for a *partial order* and an additional condition known as the comparability condition. R is a total order if the following conditions for all elements $a, b, c \in L$ hold:

1. aRa for all $a \in L$ (reflexivity)
2. aRb and $bRa \Rightarrow a = b$ (antisymmetry)
3. aRb and $bRc \Rightarrow aRc$ (transitivity)
4. For any $a, b \in L$, either aRb or bRa (comparability)

For R we often use the symbol $\leq$ (for R^{-1} the symbol $\geq$).

The first three are the axioms of a partial order, while addition of the forth axiom defines a total order. The details about totally ordered sets is indicated in [33].

Ordering the answer sets depends on the structure of how the atoms are built-in. Therefore, the sorting might arrange the atoms of the answer set itself, or arrange the whole answer sets according to the common features between them. We consider the sorting of the answer sets in our work. In particular, we provide some types of sorting to order the answer sets. At first, the sorting types are summarized below, and then described in more detail. The following kinds of sorting are:

- Cardinality. This allows the user to sort the answer sets according to the length of each answer set (number of ground atoms).
- Alphabetically. The user can quickly and easily sort the answer set itself alphabetically. Indeed, in this case, an alphabetical order is a good choice for navigating the ground atoms within an answer set.
- Arity. We take into account a common structure of the answer sets to navigate them. Basically, the answer set composed of predicates with different arity. Therefore, we sort the ground atoms within an answer set corresponding to their arity.

For sorting, we use the *quicksort* algorithm. In Algorithm 1, we see how to sort the elements of an array (or list). Quicksort has $O(n \log n)$ in average case and $O(n^2)$ in worst case performance, that is the best average case a sort algorithm can have. In [25], Knuth studies the exact average case results for many sorting algorithms. These results indicate that the quicksort is the fastest.

In the following, we describe in more detail the above mentioned types of sorting.

3.3.2 Cardinality

The answer sets are sorted according to the length of each answer set. In other words, the number of atoms of each answer set is considered. We use here the ascending order.

Definition 3.3.3 (Cardinality Ordering)**.** A *cardinality ordering* of a relation R between two answer sets A_1 and A_2, denoted by $\prec_{CR}$, is a total ordering of the length (l) of each answer set such that

$$A_1 \prec_{CR} A_2 \quad \textit{iff} \quad l(A_1) \leq l(A_2)$$

where l in $l(A_1)$ and $l(A_2)$ is the function to compute the number of ground instances in A_1 and A_2, respectively. The cardinality ordering for a specific atom in the answer sets is expressed by

$$A_1 \prec_{CR_p} A_2 \quad \textit{iff} \quad nc(p_{A_1}) \leq nc(p_{A_2})$$

where nc is the function to compute the number of occurrences of predicate p in A_1 and A_2.

Algorithm 1: `Quicksort(array)`

```
1  if length(array) > 1 then
2      pivot := select any element of array; left := first index of array; right := last index
       of array while left ≤ right do
3          while array[left] < pivot do
4              left := left + 1
5          while array[right] > pivot do
6              right := right - 1
7          if left ≤ right then
8              swap array[left] with array[right]
9              left := left + 1
10             right := right - 1
11     Quicksort(array from first index to right)
12     Quicksort(array from left to last index)
```

In fact, we need to provide a powerful tool that meets the user requirements. To this end, we classify the sorting into two options:

- General: sorting the answer sets without any restrictions.
- Specific: sorting the answer sets depending on a specific predicate.

In the general option, we order the answer sets by using the *select* SQL query with the *orderby* keyword. The table of answer sets in the database contains already the column cardinality which we can use by the *select* SQL query as shown below. MySQL uses for sorting a *filesort* algorithm which is essentially a quicksort[1]. The filesort algorithm is modified by MySQL server team to incorporate an optimization to avoid reading the rows twice: It records the sort key value, but instead of the row ID, it records the columns referenced by the query. We ignore the general option for cardinality ordering for the answer sets with the same length. Thus, the ordering of the answer sets is the same as in database.

```
"SELECT * FROM AnswerSets"
 + "WHERE AnswerSet ORDERBY Cardinality ASC ";
```

For the specific option, the answer sets are only ordered based on a specific atom which the user specified. We use the quicksort algorithm as mentioned above to perform the sorting. Below are the corresponding the ordered answer sets by cardinality with respect to general and specific options, respectively. The specific option sort the answer sets with the *place* atom.

```
Answer97
part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby)
traditional place(living_room) color(white) property(small,light)
```

[1] `https://dev.mysql.com/doc/internals/en/filesort.html`

```
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(66)
Answer98
part(backrest) part_type(legs,metal) part_type(mid_rail,metal)
place(kitchen) place(garden) chair(ingolf) modern place(living_room)
color(white) property(single) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) property(small,heavy) price(20)
Answer99
part(backrest) part_type(legs,metal) part_type(mid_rail,metal)
place(kitchen) place(dining_room) place(garden) chair(ingolf) modern
place(living_room) color(white) property(single)
part_type(armrest,acryclic) part_type(chair_seat,acryclic)
property(small,heavy) price(20)
```

The answer sets are ordered according to the length in increasing order, such as, Answer97 $\prec_{CR}$ Answer98 that means l(Answer97) $\leq$ l(Answer98) (l(Answer97) = 11 and l(Answer98) = 14).

```
Answer60
chair(melltorp) traditional place(office) place(living_room) color(white)
part(backrest) part(action_mechanism,swivel_base,wheels)
property(small,light) property(triple) part_type(armrest,acryclic)
part_type(chair_seat,leather) price(100)
Answer61
contemporary place(dining_room) place(garden) chair(ingolf)
place(living_room) color(white) part(backrest) property(single)
part_type(legs,metal) part_type(mid_rail,metal) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) property(small,heavy) price(20)
Answer62
chair(ingolf) traditional place(kitchen) place(dining_room) place(garden)
place(living_room) color(white) part(backrest) property(single)
part_type(mid_rail,acryclic) part_type(legs,acryclic)
part_type(armrest,acryclic) part_type(chair_seat,acryclic)
property(medium,heavy) price(20)
```

We show above the answer sets are sorted depending on `place` predicate. To clarify, Answer60 $\prec_{CR_{place}}$ Answer61 $\prec_{CR_{place}}$ Answer62, that means $2 \leq 3 \leq 4$.

3.3.3 Alphabetically

Definition 3.3.4 (Alphabetical Ordering)**.** Let $p(x)$ and $q(y)$ are elements (ground instances) of the answer set A. The predicates p and q are sequences of letters, such as, $p = \{a_1, ...a_n\}$ and $q = \{b_1, ...b_m\}$ where $1 \leq i \leq n$ and $1 \leq j \leq m$, respectively. An *alphabetically ordering* of a relation R with the elements of A, denoted by $\prec_{AR}$, is a total ordering of the elements of A, such that

$$p(x) \prec_{AR} q(y) \quad \textit{iff} \quad \exists i \in \mathbb{N} : a_i \neq b_i \textit{ and } a_i < b_i \text{ in the alphabet.}$$

Also, x and y are sequences of letters. To clarify, p appears before q iff at the first index i where a_i and b_i differ, a_i comes before b_i in the alphabet. We summarize the *alphabetically*

ordering of the relation R with the elements of A ($\prec_{AR}$) as follows

$$\begin{cases} x \prec_{AR} y & \text{if } p = q \\ p \prec_{AR} q & \text{otherwise} \end{cases}$$

This sorting helps the user quickly visualize the answer sets by sorting alphabetically. Similarly as in sorting by cardinality, the user has two options to order the answer sets alphabetically. To demonstrate, in general option that means that the answer set itself is sorted alphabetically without considering any specific predicate.

Additionally, this work offers the possibility of typing a specific predicate to sort all the ground atoms that are similar to the predicate. The sorted atoms are moved at the beginning of the answer set. Therefore, the remaining ground atoms of the same answer set are concatenated to the end of the ordered atoms without any sorting. For implementation, we use the quicksort algorithm. Therefore, the execution time is $O(n \log n)$ in average case.

The following two blocks show the sorted answer sets alphabetically with the general option, and the sorted answer sets alphabetically with the specific option according to the *place* predicate, respectively.

```
Answer70
chair(tobias) color(white) part(action_mechanism,swivel_base,wheels)
part(backrest) part_type(armrest,acryclic) part_type(chair_seat,leather)
place(living_room) price(20) property(small,light) property(triple)
traditional
Answer71
chair(tobias) color(white) part(action_mechanism,swivel_base,wheels)
part(backrest) part_type(armrest,acryclic) part_type(chair_seat,leather)
place(living_room) price(40) property(small,light) property(triple)
traditional
Answer72
chair(ingolf) color(white) modern part(backrest)
part_type(armrest,acryclic) part_type(chair_seat,acryclic)
part_type(legs,metal) part_type(mid_rail,metal) place(dining_room)
place(garden) place(kitchen) place(living_room) price(20)
property(small,heavy)
```

`Answer70` is sorted alphabetically that means `chair(tobias)` $\prec_{AR}$ `color(white)` $\prec_{AR}$ `part(action_mechanism,swivel_base,wheels)` $\prec_{AR}$ `part(backrest)` $\prec_{AR}$ `part_type(armrest,acryclic)` $\prec_{AR}$ `part_type(chair_seat,leather)` $\prec_{AR}$ `place(living_room)` $\prec_{AR}$ `price(20)` $\prec_{AR}$ `property(small,light)` $\prec_{AR}$ `property(triple)` $\prec_{AR}$ `traditional`.

```
Answer4
place(garden) place(living_room) chair(ingolf) modern part_type(legs,metal)
part(backrest) property(single) property(small,heavy)
part_type(mid_rail,metal) part_type(armrest,acryclic) color(white)
part_type(chair_seat,acryclic) price(20)
Answer5
place(dining_room) place(garden) modern color(white) chair(ingolf)
part(backrest) property(single) property(small,heavy) part_type(legs,metal)
```

```
part_type(mid_rail,metal) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) price(20)
Answer6
place(dining_room) place(kitchen) place(living_room) chair(ingolf) modern
color(white) price(20) part(backrest) property(single) property(small,heavy)
part_type(legs,metal) part_type(mid_rail,metal) part_type(armrest,acryclic)
part_type(chair_seat,acryclic)
```

In the above block, the answer sets are sorting according to the *place* predicate. We note that in `Answer4` the name of the predicate *place* is the same, but the atoms are different. Thus, `garden` $\prec_{AR}$ `living_room`, and in `Answer5` we show that `dining_room` $\prec_{AR}$ `garden`. The remaining atoms in each answer set exclude *place* atom are in the same order without any sorting.

3.3.4 Arity

Definition 3.3.5 (Arity Ordering)**.** Let $p(x)$ and $q(y)$ are elements (ground instances) of the answer set A. An *arity ordering* of a relation R with the elements of A, denoted by $\prec_{TR}$, is a total ordering of the elements of A, such that

$$p(x) \prec_{TR} q(y) \quad \textit{iff} \quad \textit{arity}(p) \leq \textit{arity}(q).$$

where x has a number of arguments n and y has a number of arguments m, $n, m \in \mathbb{N}$. The function *arity* computes the number of arguments in x and y.

The answer sets consist of many ground atoms. Each ground atom has different arity which means it has different ground terms as its arguments. The user is allowed to arrange the answer set according to the arity of its atoms in an increasing order. To clarify, the answer set starting with constant (*const*) -if there is a constant- and ending with the predicate (p) which has the maximum number of arguments. To put it in another way:

$$const \quad p_1(a) \quad p_2(b,c) \quad p_3(d,e,f) \quad ...$$

Likewise in the alphabetical ordering, there are two options to order the answer sets by arity; general, and specific. In the general option, we sort the answer sets depending on the number of arguments of the atoms without any limitation. In addition, we sort the answer set according to a specific predicate with different arity. After sorting, the answer sets start with this predicate with the least arity till the same predicate with the most arity. Then, the answer set will concatenate with the remaining atoms in the answer set. We implement this kind of the sorting by applying the quicksort algorithm as alphabetically sorting. The following two blocks show the answer sets ordering by arity with the general option, and the answer sets ordering by arity depending on a *property* predicate, respectively.

```
Answer27
traditional chair(kaustby) color(white) part(backrest) place(living_room)
price(35) property(double) part_type(armrest,wooden)
part_type(chair_seat,leather) property(small,light)
part(action_mechanism,swivel_base,wheels)
```

```
Answer28
traditional chair(kaustby) color(white) part(backrest) place(living_room)
price(25) property(double) part_type(armrest,wooden)
part_type(chair_seat,leather) property(small,light)
part(action_mechanism,swivel_base,wheels)
Answer29
traditional chair(kaustby) color(white) part(backrest) place(living_room)
place(office) price(120) property(triple) part_type(armrest,acryclic)
part_type(chair_seat,leather) property(small,light)
part(action_mechanism,swivel_base,wheels)
```

In the Answer27, we note that

$$\texttt{traditional} \prec_{TR} \texttt{chair(kaustby)} \prec_{TR} ... \prec_{TR} \texttt{part(action\_mechanism,swivel\_base,wheels)},$$

that means the answer set is ordered depending on the number of arguments for each atom, such as, $0 \leq 1 \leq ... \leq 3$.

```
Answer82
property(double) property(small,light) chair(idolf) traditional
place(office) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) part_type(armrest,acryclic)
part_type(chair_seat,leather) price(57)
Answer83
property(triple) property(small,light) chair(idolf) traditional
place(office) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) part_type(armrest,acryclic)
part_type(chair_seat,leather) price(73)
Answer84
property(triple) property(small,light) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) part_type(armrest,wooden)
part_type(chair_seat,leather) price(57)
```

We show that the ordering of each answer set depending on the atom *property*. The Answer82 is ordered like property(double) $\prec_{TR}$ property(small,light), where the number of arguments are 1 and 2 for property(double) and property(small,light), respectively.

3.4 Computing Similar/Diverse Solutions

It is worth finding solutions which are similar/diverse to each other. For instance, in planning, it might be useful to compute a set of similar plans. Therefore, when the execution of the plan fails, one can switch to a very similar one. Towards this goal, we study finding similar/diverse solutions in answer set programming. We deal with the answer sets, so the approach is independent of the actual program. The computation of similar and diverse solutions is symmetric. Thus, we focus on finding the diverse solutions. This section starts with the preliminaries; we recall the formal definitions and methods required for the formulation of our problem. Next, we discuss the computation of similar and diverse solutions. In the following, given an ASP program, we introduce a preprocessing and so-called (modified) interactive method to compute the diversity of solutions.

3.4.1 Preliminaries

We introduce the graph structures used to internally represent the answer sets. Then, we continue with Hamming and Jaccard distances which are the measures for similarity/diversity of the solutions.

Graph

Graphs are common fundamental data structures in knowledge representation. We use graphs to represent a set of objects and the relationship between pairs of objects. A graph is defined as the structure $G = \langle V, E \rangle$ representing a set of *vertices* V (also called *nodes*) and a set of *edges* $E \subseteq V \times V$. There are two types of graphs, *directed* and *undirected*. In our work, we consider an undirected graph [24]. All the edges in the undirected graph are bidirectional. A complete graph is a simple undirected graph in which every pair of distinct nodes is connected by a unique edge. The complete graph on n nodes has $n(n-1)\,/\,2$ edges (see Figure 3.1).

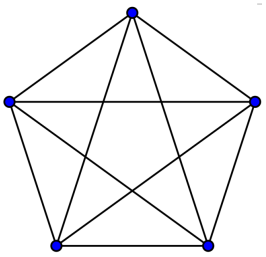

Figure 3.1: A complete graph with 5 nodes

Given a graph G, each edge in E of G might be associated with a real number, then called its *weight*. G together with these weights on its edges, is called a *weighted* graph. We therefore exploit this property to express the weight of the edges by a distance, e.g., Hamming distance or Jaccard distance. In this work, the answer sets are internally represented as the nodes of the graph. The edges are labeled by Hamming or Jaccard distances between pairs of nodes.

Clique

A *clique* in a graph G is a *complete subgraph* in G, that means, it is a subset S of the vertices V such that each two vertices in S are joined by an edge in G. A *maximal* clique is a clique with the maximum number of vertices; no more vertices can be added. In this work, we are interested to find a maximal clique to obtain a largest complete subgraph. To demonstrate, if the user needs n answer sets that differ in k atoms, we need to find a maximal clique with size n or greater than n that differ in k atoms.

Reporting the maximal cliques of a graph is a major problem arising in graph structures. The output of maximal clique enumeration algorithm may be exponentially sized, so that an algorithm with provably good running time w.r.t. the input size is not possible. However, any algorithm reporting all maximal cliques should be output sensitive.

There is a comprehensive bibliography of clique enumeration algorithms. For instance, Bron-Kerbosch (BK) algorithm [5], output-sensitive algorithms by Tsukiyama et al. [32], and new algorithms with an alternative strategy based on matrix multiplication [28]. Recently, all papers acknowledged BK algorithm as the best one in practice [4]. We choose herein BK algorithm to find diverse/similar answer sets.

In [5], Bron and Kerbosch report two algorithms, version 1 and version 2. Version 2 is an optimization of version 1 based on pivots or fixed points. In this work, we implement Bron-Kerbosch (version 2) algorithm to find maximal cliques of the graph whose nodes are the answer sets. We choose Bron-Kerbosch (version 2) to reduce the size of the recursion tree of Bron-Kerbosch and consequently the execution time of algorithm is reduced. We will explain in more detail in the next sections.

The Bron-Kerbosch (BK) algorithm

To begin with, we recall the classic Bron-Kerbosch (version 1) algorithm. Algorithm 2 shows the steps how to compute maximal cliques of the graph. To illustrate, on the first call R and X are set to $\emptyset$, and P contains all vertices of the graph. The algorithm can be explained as following: Pick a vertex v from P to expand. Add v to R and remove its non-neighbors from P and X. Then pick another vertex from the new P set and repeat the process. Continue until P is empty. Once P is empty, if X is empty then report the content of R as a new maximal clique. Now backtrack to the last vertex picked and restore P, R and X as they were before the choice, remove the vertex from P and add it to X, then expand the next vertex.

Notations. We shall denote the graph $G = \langle V, E \rangle$. Given a node $u \in G$, $nbrs(u)$ denotes the neighbors of u, i.e. $nbrs(u) = \{v \mid (u, v) \in E\}$.

Algorithm 2: `Bron-Kerbosch (Version 1)`

Input: A graph G.
Output: All maximal cliques in G.

$P = \{V\}$ //set of all vertices in Graph G
$R = \{\}$
$X = \{\}$
BronKerbosch(P, R, X)
if $P \cup X = \emptyset$ **then**
    report R as a maximal clique
for *each vertex* $v \in P$ **do**
    BronKerbosch($P \cap nbrs(v)$, $R \cup v$, $X \cap nbrs(v)$)
    $P \leftarrow P \setminus v$
    $X \leftarrow X \cup v$

The classic BK algorithm exhibits very poor output sensitivity in the worst case. Because the algorithm knows that a clique is non-maximal only when it has been fully formed in R. A variant of the algorithm so-called version 2 which aims to reduce recursive calls by using a pivot (or fixed point). Algorithm 3 shows version 2 of BK algorithm. The heuristic is based on the identification and elimination of equal sub-trees appearing in different branches of the recursion tree which lead to the formation of non-maximal cliques. Thus, the time execution of BK (version 2) algorithm is less than the time execution of (version 1).

Algorithm 3: `Bron-Kerbosch (Version 2)`

Input: A graph G.
Output: All maximal cliques in G.

$P = \{V\}$ //set of all vertices in Graph G
$R = \{\}$
$X = \{\}$
BronKerbosch(P, R, X)
if $P \cup X = \emptyset$ **then**
    report R as a maximal clique
choose a pivot $u \in P \cup X$
for *each vertex* $v \in P \backslash nbrs(u)$ **do**
    BronKerbosch($P \cap nbrs(v)$, $R \cup v$, $X \cap nbrs(v)$)
    $P \leftarrow P \setminus v$
    $X \leftarrow X \cup v$

The strategy in Algorithm 3 is to choose the pivot as the node in $P \cup X$. The pivoting consists of the following: instead of iterating at each expansion on P set, chose a pivot. The results will have to contain either the pivot or one of its non-neighbors, since if none of the non-neighbors of the pivot is included, then we can add the pivot itself to the result.

The Bron-Kerbosch algorithm is not an output-sensitive algorithm, so it does not run in polynomial time per maximal clique generated. However, it is efficient in a worst-case sense: any n-vertex graph has at most $3^{n/3}$ maximal cliques, and the worst-case running time of the Bron-Kerbosch algorithm (with a pivot strategy that minimizes the number of recursive calls made at each step) is $O(3^{n/3})$.

Hamming Distance

Hamming distance is used to measure similarity and diversity between two sequences. It is limited to cases when two sequences have the same length. The Hamming distance is defined to be the number of positions at which the corresponding symbols are different. The sequences may be strings or binary vectors.

At first, we should ensure that the length of two sequences are the same. After that, we compare the first two bits in each string. If they are the same, record a "0" for that bit. If they are different, record a "1" for that bit. At the end, we add all the ones and zeros to obtain the Hamming distance [29].

Similarly, for answer sets, the length of two answer sets (number of atoms) should be the same. Each answer set is represented as a vector of boolean values. We compare the first contents of the two indexes in each vector. If they are the same, record a "0", otherwise, record a "1" for that index. For the following two answer sets the hamming distance between the following two answer sets with the same length is 1.

```
Answer1
part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby)
traditional place(living_room) color(white) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(66)
Answer2
part(backrest) part(action_mechanism,swivel_base,wheels) chair(kaustby)
traditional place(living_room) color(white) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(42)
```

After comparing two answer sets, we get a vector $\langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 \rangle$. All "0" symbols in the vector means that the corresponding atoms of the answer sets are the same and "1" symbol means that the two answer sets are different in one atom. This atom is `price`.

Jaccard Distance

A very simple and often effective approach to measure the similarity and dissimilarity between non-empty finite sample sets is the *Jaccard index*. The Jaccard index [9], also known as *Jaccard coefficient* is used to compare the similarity and diversity of non-empty finite sample sets. The Jaccard coefficient is defined as the size of the intersection divided by the size of the union of the sample sets.

$$J(A, B) = \frac{| A \cap B |}{| A \cup B |} \qquad 0 \leq J(A, B) \leq 1$$

The *Jaccard distance* is complementary to Jaccard coefficient and is obtained by subtracting the jaccard coefficient from 1.

$$d_J = 1 - J(A, B)$$

Similarly, we can define the Jaccard distance by dividing the difference of the sizes of the union and the intersection of two sets by the size of the union:

$$\frac{| A \cup B | - | A \cap B |}{| A \cup B |}$$

The Jaccard coefficient measures the similarity between the non-empty finite sample sets, but the Jaccard distance measures the dissimilarity between the non-empty finite sample sets. For the answer sets, we use this measure when we have different lengths of the answer sets. In the following, we have two answer sets with different length.

```
Answer1
part(backrest) part_type(legs,metal) part_type(mid_rail,metal)
place(kitchen) place(dining_room) place(garden) chair(ingolf) contemporary
place(living_room) color(white) property(single) property(small,heavy)
part_type(armrest,acryclic) part_type(chair_seat,acryclic) price(20)
Answer2
part(backrest) part_type(legs,metal) part_type(mid_rail,metal)
chair(ingolf) modern place(living_room) color(white) property(single)
property(small,heavy) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) price(20)
```

The Jaccard coefficient is

$$J(A, B) = \frac{11}{16} = 0.6875$$

that means `Answer1` $\cap$ `Answer2` $= 11$ and `Answer1` $\cup$ `Answer2` $= 16$.

3.4.2 Preprocessing

In the preprocessing method, we compute all the answer sets of an ASP program by running an ASP solver. The answer sets are stored with their cardinality in a database. At the same time, we get and store the set of all ground atoms of the answer sets in a text file. After that, we create a hash mapping data structure which maps each ground atom in the file to an integer number. From the set of all ground atoms, we can check whether is in an answer set or not in order to build the boolean vector for computing distance purpose. We build a complete undirected graph $K = \langle S, E \rangle$ whose nodes S correspond to the answer sets AS and edges $E = \{\{s_i, s_j\} \mid \forall s_i, s_j \in S, s_i \neq s_j\}$ are labeled by a function $L : e \rightarrow \mathbb{N}$ that maps each $e \in E$ to a natural number (the distance), such that, $L(\{s_i, s_j\}) = d(s_i, s_j)$. The distances between the corresponding answer sets are calculated by Jaccard or Hamming distance. Additionally, we store the value of maximum (resp., minimum) distance, denoted by d_{max} (resp., d_{min}) between the answer sets for computing diversity/similarity. Figure 3.2 shows the process starting with the ASP program (file.lp) and running the Clingo solver to obtain the answer sets till the data sources.

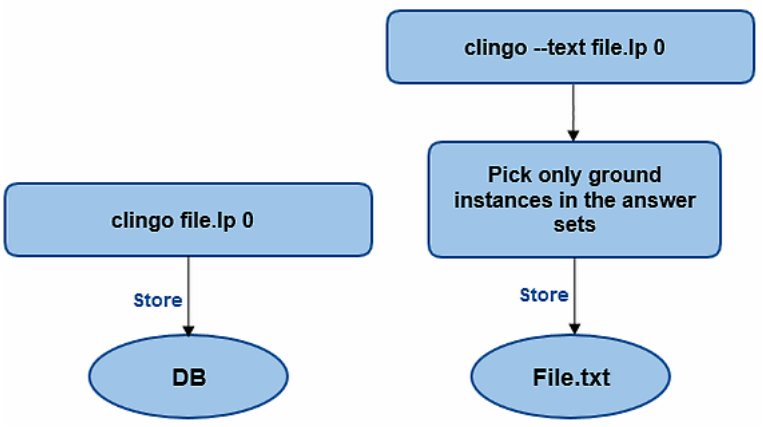

Figure 3.2: Preprocessing

3.4.3 Interactive Method (IM)

We study various problems to find similar/diverse answer sets of the given ASP program. As in illustration, the user can specify the number of the answer sets that differ in a certain number of k atoms. More precisely, if the user needs n different answers and specifies a relational operator (e.g., $\leq$) and k atoms then, the result n should be different in $\leq k$ atoms. There are two distances we use for edges in this work. Hamming distance for the answer sets with the same length, and the Jaccard distance for the answer sets with different lengths. After we build a complete undirected graph K in the preprocessing method, we check whether there exists a complete subgraph (or a clique) of size n in K whose distance is specified by the user. In this work, we find a maximal clique to obtain the largest complete subgraph. In detail, if the user needs n answer sets that differ in k atoms, a maximal clique will be found with size greater than or equal to n. Each node in the maximal clique corresponds to an answer set, so it represents exactly one answer set.

Definition 3.4.1. Let A and B be sets. The set A corresponds to the set B, denoted by $\approx_c$, where $A = \{a_1, ..., a_n\}$ and $B = \{b_1, ..., b_n\}$, such that

$$A \approx_c B \quad iff \quad \forall b_i \in B \quad a_i = b_i$$

where $1 \leq i \leq n$.

We are mainly interested in four cases of problems related to computation of a diverse/similar answer sets:

Case 1 (n k-Diverse Answer Sets (resp., n k-Similar Answer Sets))

Instance. Given a complete graph $K = \langle S, E \rangle$ whose nodes $S \approx_c AS$ (answer sets) of an ASP program P where $E \subseteq S \times S$, two non-negative integers n and k, and a relational operator op with $op \in \{<, >, =, \leq, or \geq\}$.

Question. Does there exist a set S_1 with the cardinality $|S_1| \leq n$ where a complete subgraph (clique) $K{\downarrow_{S_1}} := (S_1, E_1 \subseteq S_1 \times S_1)$, $E_1 \subseteq E$ and $S_1 \subseteq S$, and the distance of the set S_1,

denoted by $d_s(S_1)$, is the distance between each pair of its elements such that

$$d_s(S_1)\ op\ k \quad iff \quad \forall s_i, s_j \in S_1 \quad d(s_i, s_j)\ op\ k$$

where $1 \leq i, j \leq n$.

As an illustration, given a complete undirected graph K whose nodes are the answer sets and the edges are labeled by the distance between pairs of nodes. The user specifies two integer numbers n and k; n is the number of the answer sets that differ in k atoms. A clique of size n with distance $d\ op\ k$ is picked from K. The motivation for finding the clique with distance d is to find corresponding answer sets which are different in k atoms. Algorithm 4 shows Case 1 of the interactive method.

Algorithm 4: `IMCase1`

Input: A complete graph $K = \langle S, E \rangle$.
Two non-negative integers n and k.
A relational opertor op is one of $\{=, <, >, \leq, \geq\}$.
Output: A set S_1 of at most n answer sets whose distance is $d_s(S_1)\ op\ k$.
$E' \leftarrow \{\{s_j, s_i\} \mid s_j \neq s_i$ denote $s_j, s_i \in S$, $d(s_j, s_i)\ op\ k\}$
$P = \{S\}$
$R = \{\}$
$X = \{\}$
S_1 = **BronKerbosch(**P**,** R**,** X**,** E'**)**

Notations. We shall denote the graph $K' = \langle S, E' \rangle$. Given a node $u \in S$, $nbrs(u)$ denotes the neighbors of u, i.e. $nbrs(u) = \{v \mid (u, v) \in E'\}$.

Algorithm 5: `BronKerbosch(`P`,`R`,`X`,`E'`)`

if $P \cup X = \emptyset$ **then**
  if $|R| \geq n$ **then**
    pick R as a maximal clique of size at least n
    $S_1 \leftarrow$ report the first n elements of R
    Output S_1
    terminate
choose a pivot $u \in P \cup X$
for *each vertex* $v \in P \backslash nbrs(u)$ **do**
  BronKerbosch($P \cap nbrs(v)$, $R \cup v$, $X \cap nbrs(v)$)
  $P \leftarrow P \setminus v$
  $X \leftarrow X \cup v$

To illustrate, the interactive method (Case 1) algorithm can be textualized as following: Pick a set S of K such that the distance of each pair of the vertices is equal to $op\ k$. Three disjoint sets of vertices P, R, and X. R and X are set to $\emptyset$, and P contains all vertices in S.

Then apply Bron-Kerbosch (version 2) algorithm. Once P and X are empty, and the size of R is greater than or equal to n, pick R as a maximal clique. Report the first n elements of R and terminate the algorithm. For example, we give a sample of the answer sets below. These answer sets are obtained after applying this scenario: "find $n = 3$ answer sets differ in $k = 1$".

```
Answer1
chair(ingolf) contemporary place(living_room) color(white) part(backrest)
property(small,heavy) property(single) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) part_type(mid_rail,acryclic)
part_type(legs,acryclic) price(20)
Answer2
chair(ingolf) contemporary place(office) color(white) part(backrest)
property(small,heavy) property(single) part_type(armrest,acryclic)
part_type(chair_seat,acryclic) part_type(mid_rail,acryclic)
part_type(legs,acryclic) price(20)
Answer3
chair(ingolf) contemporary place(office) place(living_room) color(white)
part(backrest) property(small,heavy) property(single)
part_type(armrest,acryclic) part_type(chair_seat,acryclic)
part_type(mid_rail,acryclic) part_type(legs,acryclic) price(20)
```

From the listing above, we note that the diversity between `Answer1` and `Answer2` is 1, for instance, `place(living_room)` $\in$ `Answer1` and `place(office)` $\in$ `Answer2`. The difference between `Answer1` and `Answer3` is `place(office)`. To clarify, the number of atoms in `Answer1` is 12, but the number of atoms in `Answer3` is 13 atoms, so `place(office)` $\in$ `Answer3` but `place(office)` $\notin$ `Answer1`. Similarly, the difference between `Answer2` and `Answer3` is 1.

Case 2 (k-Distant Answer Set (resp., k-Close Answer Set))

Instance. Given answer sets (AS) of an ASP program P, two non-negative integers n and k, an answer set as, and a relational operator op with $op \in \{<, >, =, \leq, or \geq\}$.

Question. Does there exist a set H with the cardinality $|H| \leq n$ where a set $S \in AS$ and $H \approx_c S$ ($as \notin S$), and the distance of the set H, denoted by $d_s(H)$, is the distance between as and its elements such that

$$d_s(H)\ op\ k \quad iff \quad \forall s_i \in H \quad d(as, s_i)\ op\ k$$

where $1 \leq i \leq n$.

In this case, we have the answer sets (AS). The user specifies two integers k and n, and an answer set as. The answer set as is considered as a pivot for other answer sets. A subgraph of size at most n with $d\ op\ k$ is built from AS. The intuition behind building a subgraph not a clique is to find the answer sets (nodes) that have only edges with as. In fact, we do not need a clique (complete subgraph) in this case (see Algorithm 6).

To illustrate, applying this query: "find $n = 3$ answer sets differ in $k = 3$ with respect to answer set as". For instance, the chosen answer set (as) is

Algorithm 6: `IMCase2`

Input: The answer sets (AS).
1 Two non-negative integers n and k.
2 A relational opertaor op is one of $\{=, <, >, \leq, \geq\}$.
3 An answer set as.
Output: A set H of n answer sets whose distance is $d\ op\ k$ with respect to as.
4 $S \leftarrow$ Define a set of $|AS|$ vertices
5 $E \leftarrow \{\{as, s_i\} \mid as \neq s_i$ denote $as \notin S, s_i \in S, d(as, s_i)\ op\ k\}$
6 $H \leftarrow$ Find a set H of size at most n in $\langle S, E\rangle$
7 **return** H

```
chair(melltorp) traditional place(living_room) color(white)
part(backrest) part(action_mechanism,swivel_base,wheels)
property(small,light) property(triple) part_type(armrest,wooden)
part_type(chair_seat,leather) price(35)
```

```
Answer1
chair(kaustby) traditional place(office) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(57)
Answer2
chair(kaustby) traditional place(garden) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(60)
Answer3
chair(kaustby) traditional place(office) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(76)
```

The above sample of the result of the query indicates that the difference between *as* and other answer sets is 3. For example, the difference between *as* and `Answer1` is 3 atoms, such as, `chair`, `place`, and `price`. The same diversity between *as* and other answers.

Case 3 (n k-Diverse Answer Sets with Specific Ground Instance(s) (resp., n k-Similar Answer Sets with Specific Ground Instance(s)))

Instance. Given a complete graph $K = \langle S, E\rangle$ whose nodes $S \approx_c AS$ (answer sets) of an ASP program P where $E \subseteq S \times S$, two non-negative integers n and k, specific ground instance(s) x , and a relational operator op with $op \in \{<, >, =, \leq, or \geq\}$.

Question. Does there exist a set C with the cardinality $|C| \leq n$ where a complete subgraph (clique) $K{\downarrow_C} := (C, F \subseteq C \times C)$, $F \subseteq E$ and $C \subseteq S$, $x \in a_i\ \forall a_i \in AS$, and the distance of

the set C, denoted by $d_s(C)$, is the distance between each pair of its elements such that

$$d_s(C)\ op\ k \quad iff \quad \forall c_i, c_j \in C \quad d(c_i, c_j)\ op\ k$$

where $1 \leq i, j \leq n$.

To illustrate, given a complete undirected graph K whose nodes are the answer sets and the edges are labeled by the distance between pairs of the nodes. The user specifies two integer numbers n and k, and ground instance(s) x. All the answer sets contain ground instance(s) x are selected. Similarly to Case 1, a clique of size at most n with distance $d\ op\ k$ is picked from K. The motivation for finding the clique with the distance d is to find corresponding answer sets which are different in k atoms and contain x. Algorithm 7 shows Case 3 of the interactive method.

Algorithm 7: `IMCase3`

Input: A complete graph $K = \langle S, E \rangle$.
Two non-negative integers n and k.
A relational opertaor op is one of $\{=, <, >, \leq, \geq\}$.
Ground instance(s) x that all answer sets contain.
Output: A set C of at most n answer sets whose distance is $d\ op\ k$ and contain x.
$S' \leftarrow$ Select a set of $|S|$ vertices contain x
$E' \leftarrow \{\{s_j, s_i\} \mid s_j \neq s_i$ denote $s_j, s_i \in S'$ and $d(s_j, s_i)\ op\ k\}$
$P = \{S'\}$
$R = \{\}$
$X = \{\}$
$C =$ **BronKerbosch**(P, R, X, E')

As an illustration, given an a query "find $n = 3$ answer sets differ in $k = 2$ of all the answer sets have $x =$ `chair(vilmar)`". The selected ground instance of an atom `chair` is

```
chair(vilmar)
```

```
Answer1
chair(vilmar) traditional place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(57)
Answer2
chair(vilmar) traditional place(office) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) property(small,light)
property(triple) part_type(armrest,wooden) part_type(chair_seat,leather)
price(58)
Answer3
chair(vilmar) traditional place(office) place(living_room) color(white)
part(backrest) part(action_mechanism,swivel_base,wheels)
property(small,light) property(triple) part_type(armrest,wooden)
part_type(chair_seat,leather) price(47)
```

We note that all the answer sets contain `chair(vilmar)` and they are different in 2 atoms. For instance, `place(living_room)` $\in$ `Answer1` and `price(57)` $\in$ `Answer1`.

These two ground instances are different from `Answer2` which has `place(office)` and `price(58)`. `Answer1` and `Answer3` are also different in 2 atoms, such as, `place` and `price`. More precisely, the length of `Answer3` is 12 and it has `place(office)` and `place(living_room)`, but `place(office)` $\notin$ `Answer1`. The length of `Answer1` is 11.

Case 4 (n-Most Diverse Answer Sets (resp., n-Most Similar Answer Sets))

Instance. Given a complete graph $K = \langle S, E\rangle$ whose nodes $S \approx_c AS$ (answer sets) of an ASP program P where $E \subseteq S \times S$, a non-negative integer n, and the value of the maximum distance d_{max}.

Question. Does there exist a set S_1 with the cardinality $|S_1| \leq n$ where a complete subgraph (clique) $K{\downarrow_{S_1}} := (S_1, E_1 \subseteq S_1 \times S_1)$, $E_1 \subseteq E$ and $S_1 \subseteq S$, and the distance of the set S_1, denoted by $d_s(S_1)$, is maximum (resp., minimum) distance between each pair of its elements such that

$$d_s(S_1) = d_{max} \quad iff \quad \forall s_i, s_j \in S_1 \quad d_{max} = max\{d(s_i, s_j) \mid \{s_i, s_j\} \in E\}$$

where $1 \leq i, j \leq n$.

To demonstrate, given a complete undirected graph K whose nodes are the answer sets and the edges are labeled by the distance between pairs of the nodes. The user specifies an integer number n and the value of maximum distance d_{max} (and for minimum distance d_{min}) are stored during the preprocessing method; n is the number of the answer sets that differ in d_{max} atoms. A clique of the size at most n with distance equal to d_{max} is picked from K. The motivation for finding the clique with distance d is to find corresponding answer sets which are different in d_{max} atoms. Algorithm 8 shows Case 4 of the interactive method.

Algorithm 8: `IMCase4`

Input: A complete graph $K = \langle S, E\rangle$.
A negative integer n.
The maximum distance d_{max} and the minimum distance d_{min}.
Output: A set S_1 of at most n answer sets whose distance is $d_s(S_1) = d_{max}$.

```
S_1 = IMCase1(⟨S, E⟩, n, d_max, =)
if |S_1| = ∅ then
    while |S_1| = ∅ and d_max ≥ d_min do
        d_max ← d_max − 1
        S_1 = IMCase1(⟨S, E⟩, n, d_max, =)
return S_1
```

As an illustration, we find n-most diverse answer sets (resp., n-most similar answer sets) by calculating the maximum (minimum) value of the distance and check whether there exists a maximal clique with this distance. If there is no maximal clique, the distance will be decreased by 1 (resp., increased by 1) until the maximal clique is found. This case is a special case of the Case 1 (IMCase1 algorithm) which the relational operator *op* is always "=". Thus, IMCase1 algorithm is invoked for several times until the maximal clique is found and the value of $d_{max} \geq d_{min}$.

3.4.4 Modified Interactive Method (MIM)

Instead of building a graph $K = \langle S, E\rangle$ of all the answer sets (nodes S) in the preprocessing method and then a clique is picked with specific distance value in the interactive method, we can build a complete subgraph (clique) with only the edges with the specific distance value that the user specifies. Our intuition behind building a clique during the interactive method which is so-called modified interactive method is to present only to the user the answer sets (AS) he is interested in. Thus, we save the memory since the complete graph K is not built in the preprocessing method. Additionally, the execution time is reduced of the preprocessing method. We implement the same cases as we mentioned in Section 3.4.3.

In Chapter 5, we will show the execution time and the space for the preprocessing and the modified interactive method. Algorithm 9 shows Case 1 of the modified interactive method.

Algorithm 9: `MIMCase 1`

Input: The answer sets (AS)
Two non-negative integers n and k.
A relational opertaor op is one of $\{=, <, >, \leq, \geq\}$.
Output: A set S_1 of at most n answer sets whose distance is $d_s(S_1)$ op k.
$S \leftarrow$ Define a set of $|AS|$ vertices
$E \leftarrow \{\{s_j, s_i\} \mid s_j \neq s_i$ denote $s_j, s_i \in S$, $d(s_j, s_i)$ op $k\}$
S_1 = **IMCase1**($\langle S, E\rangle$, n, k, op)

The inputs of the above algorithm are the answer sets (AS) that the user specifies, the two non-negative integer numbers n and k; n is the number of the answer sets that differ in k atoms, and a relational operator op with $op \in \{<, >, =, \leq, or \geq\}$. At first, a complete undirected graph K_m is built from AS whose nodes are S and the edges E are labeled by the distance between pairs of the nodes. Then, the IMCase1 algorithm is invoked with specific arguments, such as, the complete graph K_m, n, k, and the relational operator op, to find the maximal clique and return a set S_1 of at most n answer sets whose distance is $d_s(S_1)$ op k. The other cases are the same as before. The interactive method and the modified interactive method are different only in the inputs.

3.5 Faceted Search

Faceted search, also called faceted browsing or faceted navigation is a common approach for accessing data. It allows users to navigate a collection of data by using several attributes. In addition, it offers the users the ability to create their own custom navigation by combining several attributes instead of forcing them to follow a specific path.

Intuitively, a facet represents a certain perspective of information. The values within a facet can be a list that allows only one choice or a hierarchical list that allows the user to navigate data through multiple levels. A faceted taxonomy is a combination of all facets and values.

We exploit this approach in NavAS to be more valuable and easy to use. In NavAS, the user have the ability to combine any value of filter with any type of sorting and at

the same time he allows to find similar/divers solutions. The sorting types are associated with different options as aforementioned. The following block shows this scenario : " show me 5 answer sets that are different in 1 predicate. Simultaneously, these answer sets should contain an `traditional` constant and a `part_type(chair_seat,leather)` as ground atom. These answer sets should be sorted by alphabetically with a specific predicate `property`. Additionally, the answer sets are allowed with the same length 11."

```
Answer1
property(small,light) property(triple) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels)  part_type(armrest,wooden)
part_type(chair_seat,leather) price(57)
Answer2
property(small,light) property(triple) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) part_type(armrest,wooden)
part_type(chair_seat,leather) price(85)
Answer3
property(small,light) property(triple) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels) part_type(armrest,wooden)
part_type(chair_seat,leather) price(90)
Answer4
property(small,light) property(triple) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels)  part_type(armrest,wooden)
part_type(chair_seat,leather) price(53)
Answer5
property(small,light) property(triple) chair(melltorp) traditional
place(living_room) color(white) part(backrest)
part(action_mechanism,swivel_base,wheels)  part_type(armrest,wooden)
part_type(chair_seat,leather) price(25)
```

As we show above, the length of the answer sets is the same. The Hamming distance is used to find the diversity. The answer sets are followed to the scenario which means that all of them have `part_type(chair_seat,leather)` and `traditional`. Additionally, they are different in one atom which is `price`. They are sorted by alphabetically with respect to `property`.

Chapter 4

NavAS System

In this chapter, we start with architecture and implementation principles of our work. Then, we describe the graphical user interface of NavAS and explain step by step how to use the tool.

4.1 Architecture and Implementation Principles

The main key aspects in the work design are extensibility and flexibility in handling any ASP program. They provide a powerful system that can be used for navigating the answer sets. During the requirement analysis, we identify NavAS as a system which facilitates navigating the space of solutions. It provides many procedures and techniques such as filtering, sorting, and faceted browsing. In order to obtain a rich system, we also study the way of finding similar/diverse answer sets of a given ASP program.

The architecture of the work is shown in Figure 4.1. It consists of three layers: ASP programming layer, data sources layer, and graphical user interface.

Starting from bottom, the first layer is responsible for answer set programming. The idea of this layer is to represent a given problem by an ASP program whose answer sets correspond to the solutions. There are many solvers for running an ASP program and computing the stable models (answer sets), such as, Clasp (in conjunction with Gringo) [17, 18], DLV [26], Clingo[1] [15], and SMODELS[2] [30]. In our case, we use Clingo solver.

The second layer includes the data sources (i.e. the relational database (MySQL) and a text file). We store the answer sets after running the ASP solver in a database, and the third layer is a graphical user interface (GUI). The GUI is developed in Java. It produces the most portable applications since there is a Java virtual machine (VM) for almost every platform.

Java is an object-oriented programming language. The object-oriented program is abstracted to a collection of classes and methods (also routines or function members). We implement each ground atom of an answer set as a class *Predicate* with several methods. To illustrate this, consider the design model of a Java program with a *Predicate* class (see table 4.1).

[1] `http://sourceforge.net/projects/potassco/files/clingo/`
[2] `http://www.tcs.hut.fi/Software/smodels/`

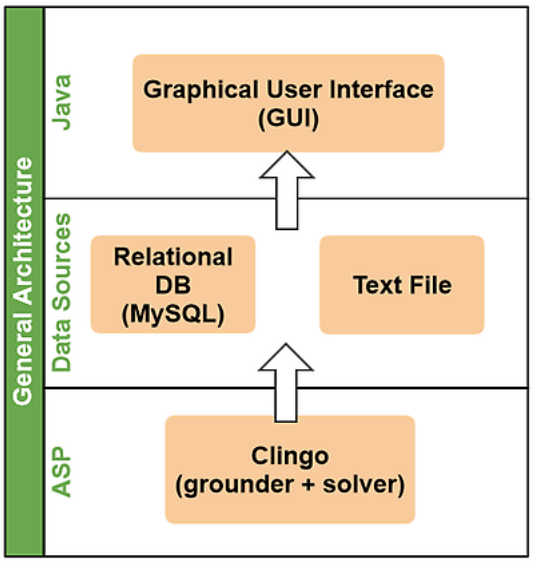

Figure 4.1: General Architecture

Table 4.1: A predicate class in Java program and its denotation

public class *Predicate* {
//methods
⋮
}

The design model of this class consists of the following methods: Predicate() Predicate(String, int, ArrayList<Object>) String getName() void setName(String) int getArity() void setArity(int) ArrayList<Object> getArgs() void setArgs(ArrayList<Object>) void printData()

We see herein the class *Predicate* has two constructors: the first one is *Predicate()* which creates an empty object and the other one creates an object with three arguments; the name of the atom, the arity of the atom, and the content of its parameters, respectively. The remaining methods which belong to the class are *get* and *set* methods. These methods are used to retrieve and change the value of what is specified in the method. For example, the *getArgs* method returns the content of the parameters of the atom as an ArrayList. It is opposite to the *setArgs()* method. Finally, *printData()* is invoked to print the atoms.

The class *AnswerSet* has a method *AnswerSet (LinkedList<Predicate>)* that recalls the

Predicate class for defining each ground atom in the answer set. Table 4.2 gives the contents of the class *AnswerSet*. We use both mentioned classes in the methods for navigating the space of answer sets.

Table 4.2: An answer set class in Java program and its denotation

public class AnswerSet {
//methods
⋮
}

The design model of this class consists of the following methods: AnswerSet() AnswerSet(LinkedList<Predicate>) LinkedList<Predicate> get AnswerSet() void printData() void printDataTextArea()

4.2 Description

The start window of NavAS tool is as depicted in Figure 4.2.

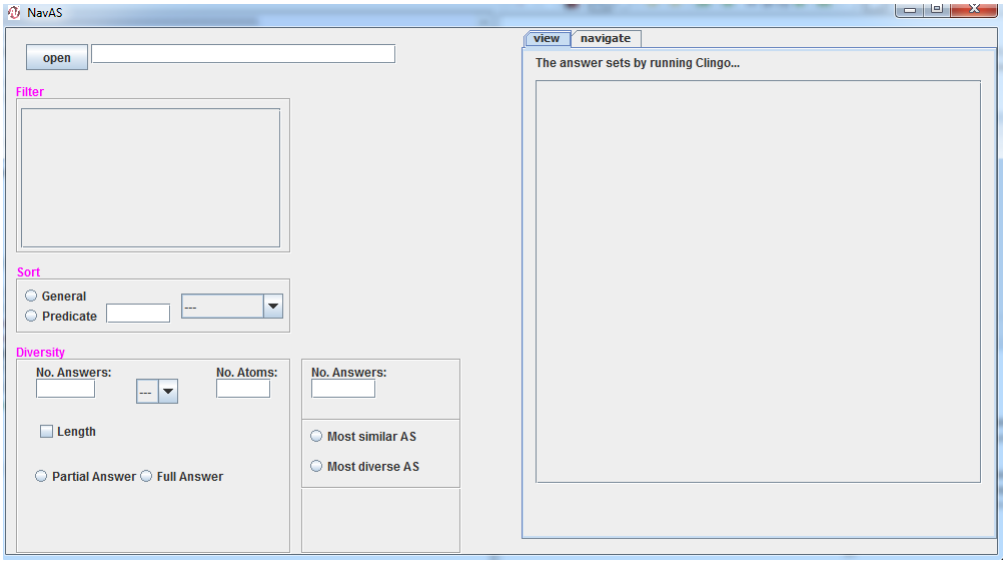

Figure 4.2: NavAS user interface

The starting point is to select a file with an *.lp* extension by the *open* button which is at the left top corner. Clicking on this button brings up a dialog box allowing to browse for the data file on the local file system as it is shown in Figure 4.3.

Once the user clicks the open button, the answer sets as a result of running the Clingo solver, are displayed in a scrollable text area in the tab *view* in the right top corner, as in

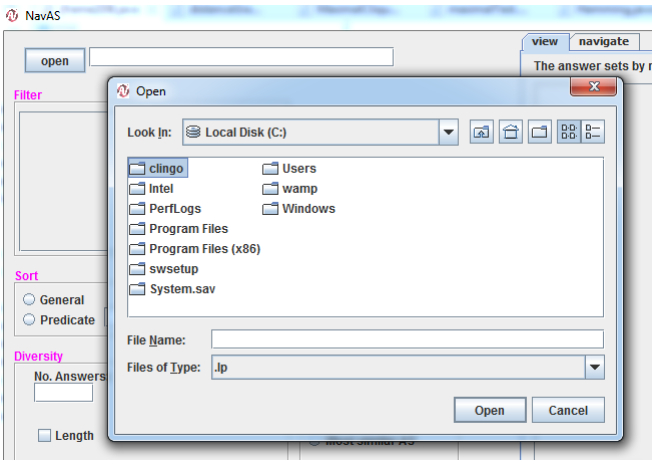

Figure 4.3: Open dialog

Figure 4.4. Keep in mind that the Clingo solver should be in the same folder or directory as the chosen file.

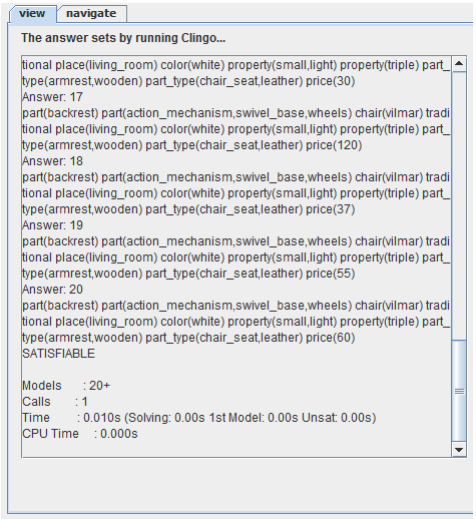

Figure 4.4: The output of ASP program

4.2.1 Filter Box

Below the open button is a box titled *Filter*. The box has a list with four columns:

- **No**. A number that identifies the atom in the order they are specified (or mentioned) in the ASP program.
- **Name**. The name of the atom and its arity, as it was declared in the ASP program (in the command line #*show*).

- **Selection check boxes**. These allow the user to select which atoms are present or not present in the answer sets (third and forth columns).

When the user selects the check boxes on different rows in the list of atoms, the contents change in a scrollable text area of the *navigate* tab. It is not allowed to select both the check boxes at the same row. Figure 4.5 shows the output of the answer sets after selecting the check boxes.

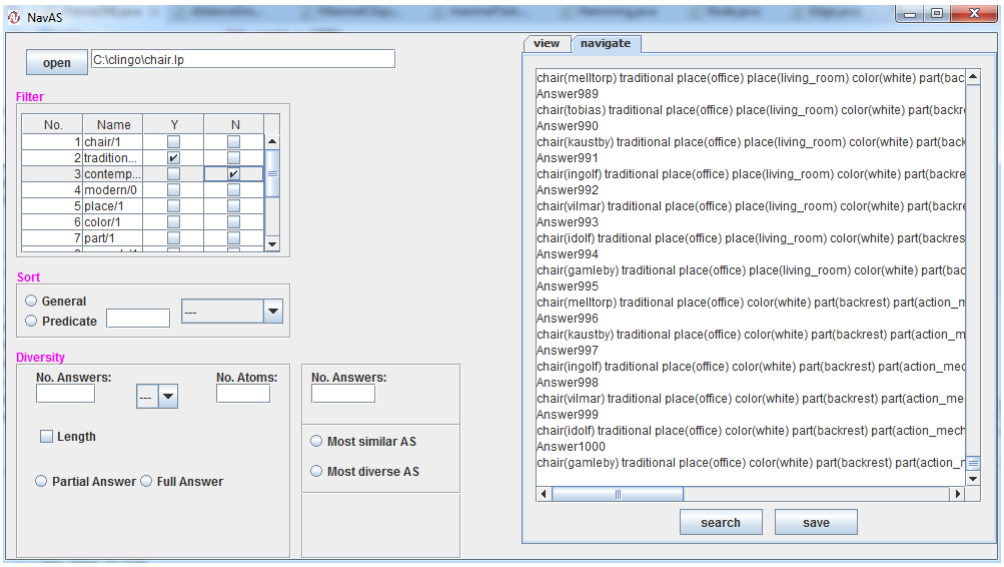

Figure 4.5: The output of filtering

As mentioned in Section 3.2, if the selected atom is numerical value, a separate panel will appear to specify the range of the numerical value of the atoms. Then, the ranking of the answer sets is proceeded. Figure 4.6 depicts the separate panel for the numerical value of the atoms. NavAS offers the feature to save the current result of the answer sets by clicking the *save* button.

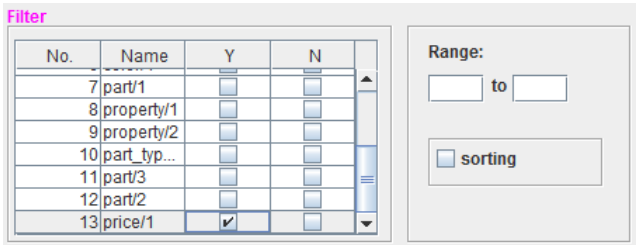

Figure 4.6: Choosing a numerical atom

4.2.2 Sort Box

The sort box appears at the left middle of the window as it is shown in Figure 4.7. It contains a drop-down list to choose from. There are three elements of the list:

- **Cardinality**. This tool allows to sort the answer sets according to the length of each one (number of ground atoms).
- **Alphabetically**. An alphabetical order is a good choice for navigating the ground atoms within an answer set.
- **Arity**. The ground instances are sorted within an answer set corresponding to their arity.

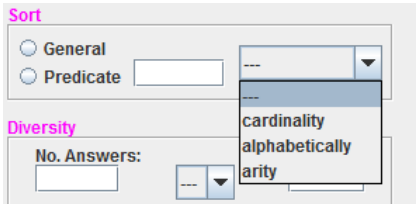

Figure 4.7: Sort box

The result of applying the type of sorting will be displayed according to the options that are set by clicking on toggled on/off boxes individually. There are two toggled on/off boxes:

- **General**. The answer sets are sorted without any limitation.
- **Specific**. The user checks a toggle box and type a specific atom's name in the adjacent text field.

To recap, the working scenario of the mentioned two boxes is: when the user choses the sorting type and considering the checked features for filtering, all results are shown, in the scrollable text area.

4.2.3 Diversity Box

The box contains many components to find the diversity between the solutions (see Figure 4.8).

To illustrate, there are three components available on this box:

- **No. Answers**. NavAS allows the user to type the number of solutions that he wants to show.
- **No. Atoms**. The number of atoms that the solutions are different in.

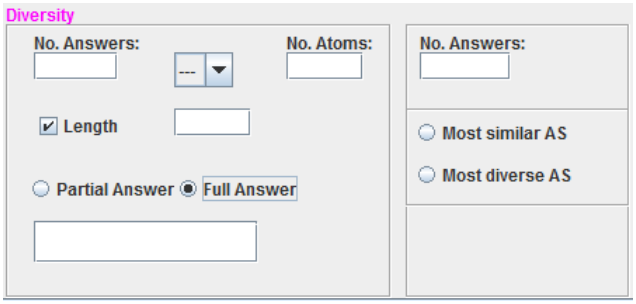

Figure 4.8: Diversity box

The working scenario of the box is as following: "Show me the answer sets (No. Answers) that are different in one of $\{=, <, >, \leq, \geq\}$ of (No. Atoms)."

There is a length option which is applied when the solutions have a different number of ground instances. The user can decide which answer set he wants as a pivot to find the diversity by either typing the full answer set or a part of it. To the right of the diversity box is the box for finding the most similar and diverse answer sets. The number of answer sets specifies by the user (No. Answers).

Note: all text fields and text areas have a drag and drop property to make it easier for the user. To illustrate, the user can drag an answer set or a part of the answer set from one text area to another.

Chapter 5

Evaluation

In this chapter, we make a comprehensive evaluation of the performance of the NavAS tool. We consider two examples; pizza and chair configuration. We discuss the experiment of results on these examples. We ran the experiments on an Intel machine with processor 2.30 GHz Intel Core i5 and memory 4 GB 665.1 MHz DDR3. The chapter begins with the pizza example and continues with the chair configuration.

5.1 Pizza Example

In real life, there are many types of pizza, and pizza consists, at its core, of three things: dough, sauce, and toppings. As an illustration, each pizza has several toppings. The type of a pizza is specified according to its toppings. More precisely, it is not possible to put all toppings (or some of them) together because of some consideration, such as, healthy and taste. Thus, pizza configurations need to check which toppings suit to each other. This example is implemented in ASP to generate different pizza configurations. The input of the ASP program are the toppings with their categories to specify the type of pizza and the price of each of them. To accomplish this task, several rules and constraints on toppings have to be satisfied. Thus, a different number of toppings provide many different configurations of pizza.

The ASP program of this example is included in Appendix A. We consider herein a sample of the code to explain the facts of the program as input. In the following, we give a listing of some facts of the program.

```
price(bacon,110).
has_category(bacon,meat).
price(chicken,120).
has_category(chicken,meat).
price(peas,140).
has_category(peas,veg).
```

We have a topping `bacon` which is indicated under `meat` category, such as, `has_category(bacon,meat)`, and its price is `price(bacon,110)`. The same thing for other facts. We add many facts to the pizza program to increase the search space. Execution

of the program with many facts by an ASP solver gives us many answer sets. We introduce below a sample of the answer sets after running the program.

```
Answer: 1
on(dough) on(tomato_sauce) on(mozzarella) on(oregano) on(bacon)
on(broccoli) on(caper) total(870) normal
Answer: 2
on(dough) on(tomato_sauce) on(mozzarella) on(oregano) on(caper) on(basil)
total(750) normal
Answer: 3
on(dough) on(tomato_sauce) on(mozzarella) on(oregano) vegetarian on(caper)
on(basil) total(750)
Answer: 4
on(dough) on(tomato_sauce) on(mozzarella) on(oregano) on(bacon) on(caper)
on(basil) total(860) normal
```

Through this sample, we have many atoms; toppings, like `on(caper)`, `on(basil)`, and `on(bacon)`, two kinds of pizza which are `vegetarian` and `normal`, and the price of each pizza, such as, for `Answer4`, the price is `8.60`. However, usually there are too many answer sets of pizza example computed by an ASP solver. The user needs to compare these answer sets, by analyzing the similar/diverse ones with respect to some distance measure, or by filtering and sorting them to get good configurations that suit his preferences. To this end, we use the tool on this example to evaluate the performance of it.

For evaluation, several parameters are considered to asses the performance of the tool, namely: (1) number of answer sets where the range of the number of atoms per answer set changes from 6 to 14 in the pizza example, (2) preprocessing method, (3) navigation methods, such as, filtering and sorting. Finally, we record the execution time in minutes of the preprocessing method and the time needed to get the output of filtering and sorting. Table 5.1 summarizes the experimental evaluation.

Table 5.1: The performance (measured in minutes) of preprocessing and two navigation methods (pizza example)

No	Answer Sets	Preprocessing	Filtering	Sorting		
				Cardinality	Alphabetically	Arity
1	1000 (137KB)	0.4669	0.00121	0.0793	0.2414	0.2319
2	5000 (691KB)	2.3302	0.00163	0.0836	0.2446	0.2447
3	10000 (1.41MB)	4.7482	0.00215	0.1051	0.2523	0.2505
4	50000 (7.5MB)	25.1397	0.00423	0.1156	0.2912	0.3198
5	100000 (15.7MB)	50.2355	0.00531	0.1214	0.3360	0.3849

For Table 5.1, we note that for the preprocessing method the time is quite high but this depends on the host machine, the solver, and the implemented code for storing the answer sets in the database. For filtering, the execution time for the different number of the answer sets is almost the same and it is faster because of using the *select* SQL query. Figure 5.1 shows the performance of sorting. The time needed to obtain the output for cardinality sorting is lower than alphabetically and arity sorting because it is executed by SQL query.

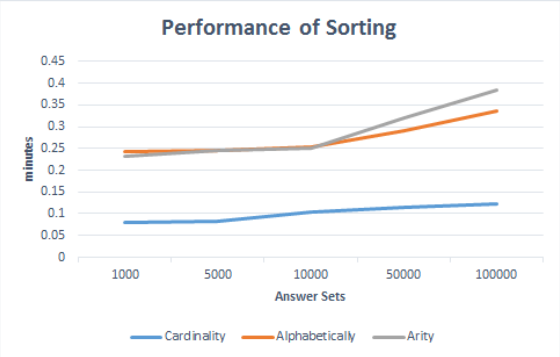

Figure 5.1: A graph of experiment result from Table 5.1

It can be clearly seen that, the alphabetically and arity sorting are similar. And they are both based on the quicksort algorithm. We note that for more than 10000 answer sets the execution time of the alphabetically and arity sorting is different. The execution time of the arity sorting is higher than the execution time of the alphabetical type of the sorting. To clarify, in arity sorting we do two steps; we check the arity of all predicates in each answer set, then we sort the atoms depending on the arity of the predicates for each answer set. In alphabetical type of sorting, we just sort the atoms alphabetically.

For similarity and diversity, we did another experiment for testing the computation time of finding diverse/similar solutions. There is a similar approach [11], but the implementation is not available any more. However, we measured the execution time of the preprocessing and the (modified) interactive method which is represented in three cases. We took five samples for each value and compute the average of them. We consider several parameters to asses the performance of the tool with respect to finding similar/diverse answer sets: (1) number of answer sets, (2) preprocessing method, (3) (modified) interactive method with three cases and each case is assessed with and without filtering. Table 5.2, 5.3, 5.4, and 5.5 report the time of three cases of interactive and modified interactive method with different values of k (number of different atoms), $k = 1, 2$, respectively, and the number of solutions n (the maximum size of the maximal cliques corresponding to the value of k).

Table 5.2: Time execution (in minutes) for finding similar/diverse solutions ($k = 1$ and $n = 3$) for pizza example

Answer Sets (AS)	Pre-processing	Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (137KB)	0.5441	0.0063	0.0046	8.3E-05	7.6E-05	0.0038	0.0028
2000 (278KB)	1.3237	0.021	0.0150	0.000103	0.00009	0.0135	0.0113

Table 5.3: Time execution (in minutes) for finding similar/diverse solutions ($k = 2$ and $n = 8$) for pizza example

Answer Sets (AS)	Pre-processing	Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (137KB)	0.5441	0.0189	0.0107	0.00021	0.00016	0.0065	0.0056
2000 (278KB)	1.3237	0.0640	0.0505	0.00056	0.00053	0.0245	0.0219

Table 5.4: Time execution (in minutes) for finding similar/diverse solutions with modified interactive method ($k = 1$ and $n = 3$) for pizza example

Answer Sets (AS)	Pre-processing	Modified Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (137KB)	0.4669	0.0716	0.0525	0.00019	7.6E-05	0.0020	0.0018
2000 (278KB)	0.7868	0.5430	0.4039	0.00061	0.00053	0.0061	0.0051

Table 5.5: Time execution (in minutes) for finding similar/diverse solutions with modified interactive method ($k = 2$ and $n = 8$) for pizza example

Answer Sets (AS)	Pre-processing	Modified Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (137KB)	0.4669	0.0677	0.0454	0.00025	0.00018	0.0064	0.0060
2000 (278KB)	0.7868	0.6081	0.4073	0.00077	0.0007	0.0593	0.0565

From Table 5.2 and 5.3, we note that the execution time for the preprocessing method is quite high because of the storing the answer sets in a database and building a complete undirected graph K. There are different values of the execution time for three cases of the interactive method. For Case 1 from Figure 5.2, we can see that the execution time for the number of answer sets 2000 is higher than 1000. Additionally, the execution time for the interactive method with filtering is less than the time needed for the same method without filtering. This can be explained with a reduction of the search space by filtering. Thus, the user only shows the answer sets what he is interested in. For the number of atoms that the answer sets are different, we note that the execution time for $k = 2$ is higher than the execution time for $k = 1$. In fact, this depends on the configurations of the answer sets of a problem, the distance between them, and the ordering of the answer sets in the complete graph K.

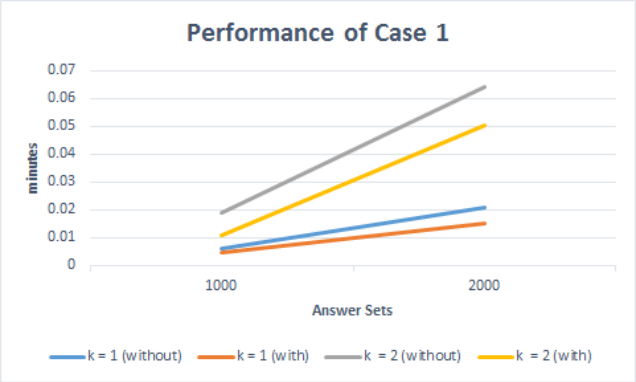

Figure 5.2: A graph of experiment result for Case 1 (with interactive method) from Table 5.2 and 5.3

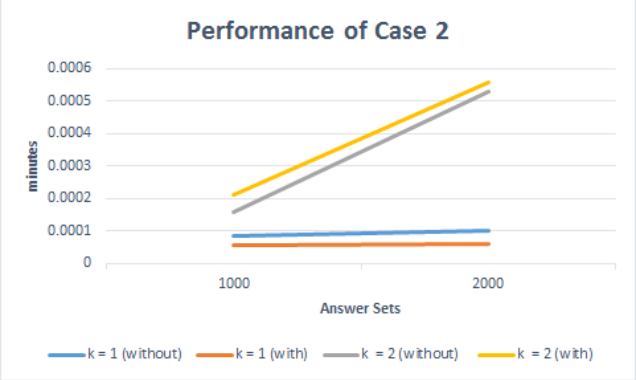

Figure 5.3: A graph of experiment result for Case 2 (with interactive method) from Table 5.2 and 5.3

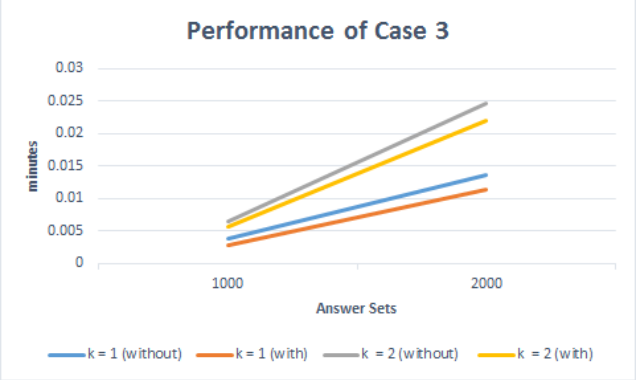

Figure 5.4: A graph of experiment result for Case 3 (with interactive method) from Table 5.2 and 5.3

For Case 2 and Case 3, we note from Figure 5.3 and 5.4 that the execution time for the interactive method with filtering is less than without filtering because of the same reason as we mentioned above. The remaining properties are the same as in Case 1. For Case 2, we note that the execution time is less than the execution time in both Case 1 and Case 3. To illustrate, in Case 2 we do not use a complete graph K to find a clique. We only compute the distance that the user specifies between a specific answer set as a pivot and other answer sets and get a subgraph.

From Table 5.4 and 5.5, we note that the execution time for the preprocessing method is less than the execution time for the preprocessing method in the previous tables. The preprocessing method herein is only used to store the answer sets. The graph K is built during the modified interactive method with a specific answer sets that the user specifies. Similarly, there are three cases for the modified interactive method. We measure the execution time for all of them.

In general, we note that the execution time for the modified interactive method for all three cases is higher than the execution time for the interactive method in the previous tables, because the time needed to build the graph K is included in the execution time for the modified interactive method. However, in order to make the visualization clearer, only a part of the graph is presented. Thus, some answer sets are shown to the user. The execution time herein depends on how many filtering options the user chooses, consequently, the number of answer sets is reduced. Thus, the number of nodes (answer sets) for building the graph K is reduced.

We compare the execution time for the interactive and modified interactive methods with the filtering option. To this end, we calculate the execution time by the summation of preprocessing time and (modified) interactive method (see Figure 5.5).

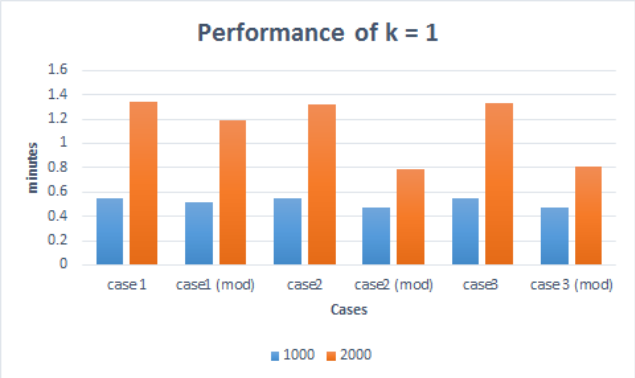

Figure 5.5: A graph of experiment result for comparing the performance of interactive and modified interactive (mod) methods

We note that the execution time for the modified interactive method is less than the time for the interactive one with the filtering option. In the filtering option, we focus on presenting the user the answer sets he is interested in. As mentioned, the size of a complete graph K in the interactive method is higher than the size of K in the modified interactive method.

5.2 Chair Configuration Example

We discuss the experiment of the results using *chair configuration* example which is mentioned in the previous chapters. For the evaluation, we consider the same parameters as mentioned before. Table 5.6 summarizes the experimental evaluation.

Table 5.6: The performance (measured in minutes) of preprocessing and two navigation methods (chair configuration example)

No	Answer Sets	Preprocessing	Filtering	Sorting		
				Cardinality	Alphabetically	Arity
1	1000 (237KB)	0.50086	0.0037	0.0842	0.23	0.2349
2	5000 (1.159MB)	2.4719	0.006	0.0888	0.2511	0.25
3	10000 (2.3MB)	6.1663	0.00755	0.1211	0.2743	0.2671
4	50000 (11.53MB)	25.48635	0.02271	0.1295	0.35	0.322
5	100000 (23.065MB)	51.2536	0.04103	0.1369	0.4511	0.3954

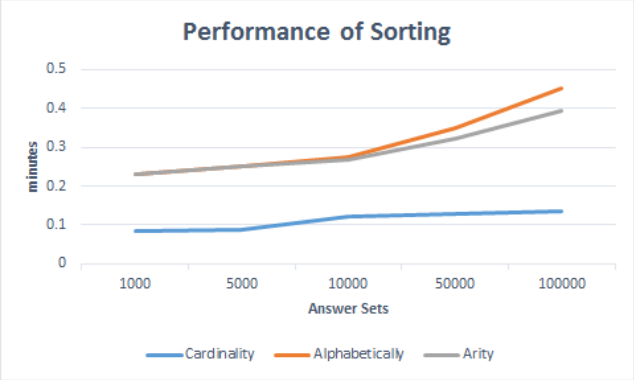

Figure 5.6: A graph of experiment result from Table 5.6

From Table 5.6, we note that for the preprocessing method the execution time is higher than the execution time in the pizza example. This difference comes from the size of the answer sets in both examples, such as, for the pizza example we note that the size of 10000 answer sets is 1.41 MB and the time execution for the preprocessing method is 4.7482 minutes, but for the chair configuration example, we note that the size is 2.3 MB for the same number of the answer sets and the needed time for the preprocessing method is 6.1663 minutes. The execution time of the preprocessing method increases with the increasing of the size and the number of the answer sets. For filtering, we note that the time execution is less than the time execution for sorting. It depends on the implemented method, such as, we use the *select* sql query for filtering, but we implement the quick sort algorithm for sorting.

Figure 5.6 shows the performance of sorting. The execution time of the cardinality sorting is less than alphabetically and arity sorting because of the same reason as mentioned before. For alphabetically and arity, we note that for more than 10000 answer sets the execution time is different because of the same reasons in the previous section. We show

that the time is almost the same as the previous section because the filtering and cardinality sorting are executed by SQL query. The execution time for alphabetical and arity sorting is a little bit higher, since the the number of the atoms per answer set changes from 11 to 17 in chair configuration example. Thus, it is more than the number of the atoms per answer set in the pizza example. Additionally, the size and the number of the answer sets in chair configuration example is higher.

For similarity and diversity, we did another experiment for testing the computation time of finding diverse/similar solutions. We measured the execution time of preprocessing and the (modified) interactive method which is represented in three cases. We took five samples for each value and compute the average of each value. We consider the same parameters as the previous section to asses the performance of the tool. Table 5.7 and 5.8, report the time of three cases of interactive method and modified interactive method with $k = 1$ (number of different atom), and the number of solutions n (the maximum size of the maximal cliques corresponding to the value of k).

Table 5.7: Time execution (in minutes) for finding similar/diverse solutions with interactive method ($k = 1$ and $n = 25$) for chair configuration example

Answer Sets (AS)	Pre-processing	Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (237KB)	0.57671	0.0433	0.0132	0.0004	0.0002	0.0114	0.0094
2000 (474KB)	1.37251	0.0599	0.0404	0.0005	0.0003	0.0374	0.0359

Table 5.8: Time execution (in minutes) for finding similar/diverse solutions with modified interactive method ($k = 1$ and $n = 25$) for chair configuration example

Answer Sets (AS)	Pre-processing	Modified Interactive Method					
		Case 1		Case 2		Case 3	
		Without	With	Without	With	Without	With
1000 (237KB)	0.50086	0.0983	0.0857	0.00051	0.0004	0.0636	0.0627
2000 (474KB)	0.90111	0.5881	0.4896	0.00056	0.0001	0.5907	0.5696

From Table 5.7, we note that the execution time for the preprocessing method is quite high because of storing the answer sets and building a complete undirected graph K. In addition, the execution time for the interactive method with the filtering is less than the time needed for the same method without filtering for the same reason as mentioned in the pizza example.

There are three cases for the interactive method. We note from Figure 5.7, 5.8 and 5.9 that the execution time for the interactive method with the filtering is less than without filtering because of the same reasons as mentioned before in the pizza example.

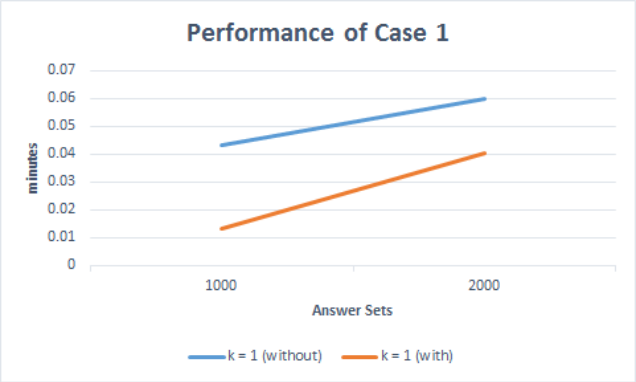

Figure 5.7: A graph of experiment result for Case 1 from 5.7

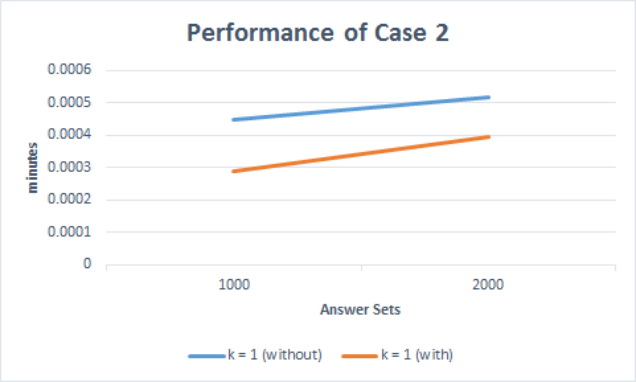

Figure 5.8: A graph of experiment result for Case 2 from 5.7

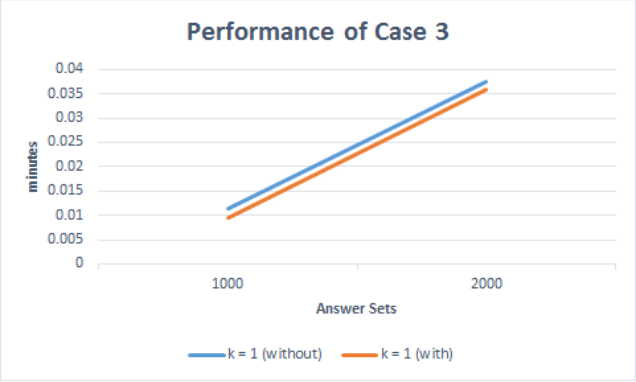

Figure 5.9: A graph of experiment result for Case 3 from 5.7

From Table 5.8, we note that the execution time for the preprocessing method is less than the execution time for the preprocessing method in the interactive method for the same reasons in the previous section. Similarly to the pizza example, the execution time for the

modified interactive method for all three cases is higher than the execution time for the interactive method in Table 5.7.

We compare the execution time for the interactive and the modified interactive methods with the filtering option. We calculate the execution time by the summation of preprocessing time and (modified) interactive method (see Figure 5.10).

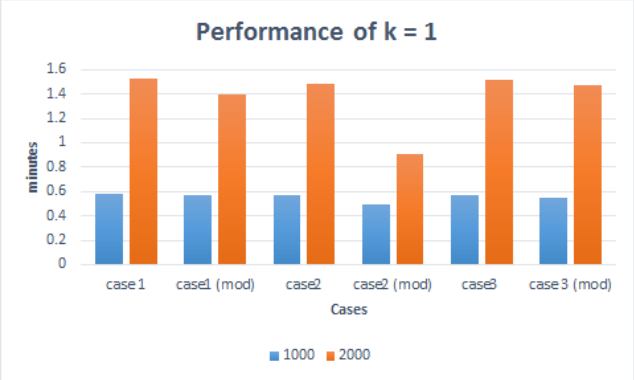

Figure 5.10: A graph of experiment result for comparing the performance of interactive and modified interactive (mod) methods

From Figure 5.10, we note that the execution time for the modified interactive method is less than the time for the interactive one with the filtering option.

We see the execution time for finding diverse solutions for chair configuration example is higher than the execution time for the pizza example, since the size of the answer sets and the database in chair configuration is larger than in the pizza example. However, we reduce the execution time for the preprocessing method by building the graph in the modified interactive method with n answer sets. However, the drawback of this is that, it takes more time in the interactive method. Additionally, preprocessing is executed once but the interactive method is executed one or several time(s). The advantage of building the graph K during the modified interactive method is given by choosing some answer sets depending on the selected filtering option. Indeed, the question has been raised, why we consume the memory to build a complete graph of whole answer sets and we visualize some of them. Thus, the modified interactive method could be a solution to save the memory and the time.

Chapter 6

Conclusions and Future Work

There is a large number of the answer sets available of an ASP program, all of which are not of the user's interest. Thus, a navigation of the search space could be a solution to help the user to access the specific answer sets. As far as we are aware, not much is available in literature on studies of the answer sets themselves. In this respect, we push our selves to study and analyze the answer sets. The motivation underlying this thesis stems from the fact that although some tools are available for developing ASP programs including editors and debuggers (e.g., APE and iGROM), it is neither considered the answer sets nor the navigation of the search space of them. To this end, we are looking into different navigation approaches that make the search faster and explore information easier.

In this thesis, we study some procedures and techniques to navigate the answer sets. We provide filtering procedure for the answer sets depending on the existence or absence of one or several ground instances. The motivation for using filtering is to reduce the number of answer sets which represent the answers most relevant to the user query. The other procedure we consider in the work is sorting. In particular, we offer some types of sorting to order the answer sets; cardinality, alphabetically, and arity. Cardinality sorting, it allows to sort the answer sets according to the length of the answer sets. While the alphabetical sorting sorts the atoms within the answer set alphabetically. The last type is arity sorting which sorts the atoms within an answer set corresponding to their arity. We study finding similar/diverse solutions of the answer sets. We offer scenarios to find similar/diverse solutions depending on a specific answer set or a specific ground instance. To this purpose, we introduce preprocessing and (modified) interactive methods and apply some distance measures. Additionally, we are also interested in combination of these approaches. We implement faceted browsing technique to combine previous approaches instead of following a specific approach.

Regarding practical use, we present NavAS, a tool for navigating the answer sets for general ASP programs. We implement the previous approaches in a graphical user interface to facilitate finding the desired answer sets and easy to use. Finally, we make a comprehensive evaluation of the performance of NavAS tool with two different ASP programs. We record the execution time for filtering, sorting, and finding diverse/similar answer sets.

As far as future work is concerned, we plan to extend NavAS tool by improving/introducing additional features. In particular, we are interested in visualizing the answer sets and the relations between them in a graph. But the huge number of the answer sets is very difficult

to see all of them. Thus, we plan to do some filters and focus on subset of answer sets.

In general, we can use the tool with any solver. In this work, we use a Clingo solver and syntactically the output is a bit different from other solvers, such as, the output of each answer set by running a DLV solver is enclosed by parentheses. As a result, we plan to make some syntactical modification on the tool to meet the requirement.

Appendix A

ASP Programs

A.1 Chair Configuration

```
%base case
part(backrest).
part2(armrest).
part2(chair_seat).

col(white).
name(ingolf).
place2(ingolf,kitchen).
place2(ingolf,living_room).
has_price(ingolf,33).
has_price(ingolf,47).
size(small).
property1(single).

%type armrest
type1(acryclic).
%type for chair seat
type2(acryclic).
%type for legs and mid rail
type3(acryclic).
     .
     .
     .

type3(metal).
weight(heavy).
weight(light).

%Guess
1{chair(X):name(X)}1.
```

```
1{contemporary;modern;traditional}1.
1{place(X):place2(Y,X)}.
1{color(X):col(X)}1.
1{property(X,Y): size(X),weight(Y)}1.
1{property(X): property1(X)}1.
1{part_type(armrest,Y):type1(Y)}1.
1{part_type(chair_seat,Y):type2(Y)}1.
1{price(X) : has_price(Y,X)}1.

%the chair might be with wheels or legs
1{part(action_mechanism,swivel_base,wheels); part(legs,mid_rail)}1.

1{part_type(legs,Y): type3(Y)}1:- part(legs,mid_rail).
1{part_type(mid_rail,Y): type3(Y)}1:- part(legs,mid_rail).

%Check
:- part_type(legs,X),part_type(mid_rail,Y), X != Y.
:- part_type(legs,_), place(office).

:- part(action_mechanism,swivel_base,wheels), place(X), X != office, X != living_room.
:- part(headrest), place(X), X != office, X != living_room.
:- part(headrest),part(legs,mid_rail).

%to display
#show chair/1.
#show traditional/0.
#show contemporary/0.
#show modern/0.
#show place/1.
#show color/1.
#show part/1.
#show property/1.
#show property/2.
#show part_type/2.
#show part/3.
#show part/2.
#show price/1.
```

A.2 Pizza

```
price(bacon,110).
has_category(bacon,meat).
price(chicken,120).
has_category(chicken,meat).
                .
```

```
                .
                .
has_category(provolone,cheese).
price(gouda,240).
has_category(gouda,cheese).

topping(X) :- price(X,Y).
%base case
on(dough;tomato_sauce;mozzarella;oregano).

0{on(X):topping(X)}.

%either normal or vegetarian
1{normal;vegetarian}1.
%constraints for vegetarian
:- on(X), has_category(X,meat), vegetarian.
:- on(X), has_category(X,fish), vegetarian.

%pizza hawai
on(ham) :- on(pineapple).
:- on(ham), on(pineapple), on(X), X!=ham, X!=pineapple.

%not more than one fish topping
%:- #count{X: on(X), has_category(X,fish)}>1.
:-2{has_category(X,fish):on(X)}.

%not more than 3 vegetables
:- #count{X: on(X), has_category(X,veg)}>3.
:- 2{on(rocket); on(basil); on(coriander)}.

%no meat and fish together
:- on(X), on(Y), has_category(X,fish), has_category(Y,meat).

%order on toppings for price calculations
lt(X,Y) :- topping(X), topping(Y), X<Y.
nsucc(X,Z) :- lt(X,Y), lt(Y,Z).
succ(X,Y) :- lt(X,Y), not nsucc(X,Y).
ninf(X) :- lt(Y,X).
nsup(X) :- lt(X,Y).
inf(X) :- not ninf(X), topping(X).
sup(X) :- not nsup(X), topping(X).

%calculate the price
base(540).
price_upto(P,X) :- inf(X), not on(X), base(P).
price_upto(P,X) :- inf(X), price(X,P2), on(X), base(P1), P=P1+P2.
```

```
price_upto(P,X) :- price_upto(P,Y), succ(Y,X), not on(X).
price_upto(P,X) :- price_upto(P1,Y), succ(Y,X), on(X), price(X,P2), P=P1+P2.

total(P) :- price_upto(P,X), sup(X).

#show on/1.
#show vegetarian/0.
#show normal/0.
#show total/1.
```

Bibliography

[1] T. Ambroz, G. Charwat, A. Jusits, J. P. Wallner, and S. Woltran. ARVis: Visualizing relations between answer sets. In *Proc. of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148 of *Lecture Notes in Computer Science*, pages 73–78, Berlin Heidelberg, 2013. Springer-Verlag.

[2] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, England-UK, 2003.

[3] M. Blondeela, S. Schockaertb, D. Vermeira, and M. D. Cock. Complexity of fuzzy answer set programming under łukasiewicz semantics. *International Journal of Approximate Reasoning*, 55(9):1971–2003, 2014.

[4] D. Braum. Finding all maximal cliques of a family of induced subgraphs. Technical report, Konrad-Zuse-Zentrum, 2003.

[5] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[6] P. Busoniu, J. Oetsch, J. Pührer, P. Skocovsky, and H. Tompits. SeaLion: An eclipse-based ide for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, 13(4-5):657–673, 2013.

[7] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[8] O. Cliffe, M. D. Vos, M. Brain, and J. Padget. ASPViz: Declarative visualisation and animation using answer set programming. In *Proc. of the 24th International Conference on Logic Programming (ICLP'08)*, Lecture Notes in Computer Science, pages 724–728, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] F. Deng, S. Siersdorfer, and S. Zerr. Efficient jaccard-based diversity analysis of large document collections. In *Proc. of the 21st ACM international conference on Information and knowledge management (CIKM '12)*, pages 1402–1411, 2012.

[10] A. H. Eden and R. Kazman. Architecture, design, implementation. In *Proc. of the 25th International Conference on Software Engineering (ICSE '03)*, pages 149–159, DC, USA, 2013. IEEE Computer Society Washington.

[11] T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar or diverse solutions in answer set programming. In *Proc. of 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2009.

[12] T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *Proc. of the International Logic Programming Symposium*, pages 266–278, 1993.

[13] O. Febbraro, K. Reale, and F. Ricca. ASPIDE: Integrated development environment for answer set programming. In *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2011.

[14] V. Gabriel. *Algorithms on trees and graphs*. Springer, 2002.

[15] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, and T. Schaub. A user's guide to gringo, clasp, clingo, and iclingo. Technical report, University of Potsdam, 2008.

[16] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Morgan & Claypool, 2013.

[17] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, Lecture Notes in Computer Science, pages 386–392. Morgan Kaufmann Inc., 2007.

[18] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 509–514. Springer, 2009.

[19] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczynski. The first answer set programming system competition. In *Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, pages 3–17. Springer-Verlag, 2007.

[20] M. Gelfond and N. Leone. Logic programming and knowledge representation-the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[21] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *R. Kowalski and K. Bowen, editors, Proc. of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[22] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 2011.

[23] J. Janssen, S. Schockaert, D. Vermeir, and M. D. Cock. *Answer Set Programming for Continuous Domains: A Fuzzy Logic Approach*. Springer, 2012.

[24] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, Volume 31(1):7–15, 1989.

[25] D. E. Knuth. *The art of computer programming: seminumerical algorithms*, volume 2. Addison-Wesley, 1997.

[26] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

[27] V. Lifschitz. Answer set planning. In *Proc. of the 16th International Conference on Logic Programming (ICLP 1999)*, pages 23–37, New Mexico, USA, 1999. MIT Press.

[28] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Algorithm theory-SWAT 2004*, volume 3111 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2004.

[29] G. D. McKenzie and D. Media. How to calculate hamming distance. `http://classroom.synonym.com/calculate-hamming-distance-2656.html`.

[30] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal lp. In *Proc. of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265, pages 420–429, Berlin, 1997. Springer.

[31] A. Sureshkumar, J. Fitch, M. Brain, and M. D. Vos. APE: An AnsProlog* environment. In *Proc. of the 1st International Workshop on Software Engineering for Answer-Set Programming (SEA 2007)*, pages 71–85, 2007.

[32] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[33] E. W. Weisstein. *CRC concise encyclopedia of mathematics*. Springer, 1999.