Answer set programming and CLASP: A tutorial

Conference Paper · April 2014

CITATIONS
6

READS
1,323

2 authors:

Steffen Hölldobler
Technische Universität Dresden
201 PUBLICATIONS **2,234** CITATIONS
SEE PROFILE

Lukas Schweizer
Technische Universität Dresden
8 PUBLICATIONS **19** CITATIONS
SEE PROFILE

Answer Set Programming and CLASP
A Tutorial

Steffen Hölldobler and Lukas Schweizer

International Center for Computational Logic
Technische Universität Dresden, 01062 Dresden, Germany
sh@iccl.tu-dresden.de lukas@janeway.inf.tu-dresden.de

Abstract We provide a tutorial on answer set programming, a modern approach towards true declarative programming. We first introduce the required theoretical background in a compact, yet sufficient way and continue to elaborate problem encodings for some well known problems. We do so by also introducing the tools `gringo` and `clasp`, a sophisticated state-of-the-art grounder and solver, respectively. In that way we cover theoretical as well as practical aspects of answer set programming, such that the interested reader will gather sufficient knowledge and experience in order to continue discovering the field of answer set programming.

1 Introduction

Answer set programming (ASP) is a modern approach to declarative programming, where a user focusses on declaratively specifying his or her problem. ASP has its roots in deductive databases, logic programming, logic based knowledge representation and reasoning, constraint solving, and satisfiability testing. It can be applied in a uniform way to search problems in the classes P, NP, and NP^{NP} as they occur in application domains like planning, configuration, code optimization, database integration, decision support, model checking, robotics, system syntheses, and many more. We assume the reader to be familiar with propositional and first-order logic as well as with logic programming [5,13,3].

2 The Interview Example

A college in the USA uses the following rules for awarding scholarships to students: (1) Every student with a GPA of at least 3.8 is eligible. (2) Every minority student with a GPA of at least 3.6 is eligible. (3) No student with a GPA under 3.6 is eligible. (4) The students whose eligibility is not determined by these rules are interviewed by the scholarship committee.

These rules can be encoded in the following program:

$$\begin{array}{ll} \mathit{eligible}(X) & \leftarrow \mathit{highGPA}(X) \\ \mathit{eligible}(X) & \leftarrow \mathit{minority}(X) \wedge \mathit{fairGPA}(X) \\ \neg\mathit{eligible}(X) & \leftarrow \neg\mathit{fairGPA}(X) \\ \mathit{interview}(X) & \leftarrow \sim\mathit{eligible}(X) \wedge \sim\neg\mathit{eligible}(X) \end{array} \tag{1}$$

where $highGPA(X)$ and $fairGPA$ denote that the GPA of student X is at least 3.8 and at least 3.6, respectively, $\neg$ and $\sim$ denote classical and default negation, respectively, and we assume that all rules are universally closed. One should observe that the last rule specifies that $interview(X)$ holds if neither $eligible(X)$ nor $\neg eligible(X)$ can be determined.

Now suppose the scholarship selection committee learns that John has a GPA of 3.7*, but his application does not give any information on whether he does or does not belong to a minority. What happens with John? Will he be invited to an interview?* This additional information can be specified by:

$$\begin{array}{ll} fairGPA(john) & \leftarrow \\ \neg highGPA(john) & \leftarrow \end{array} \tag{2}$$

3 Answer Sets

Propositional Programs For the moment we restrict ASP to propositional programs. Towards the end of this section we will extend it to first-order logic. The alphabet is the usual alphabet of propositional logic extended by the connective $\sim$denoting default negation. The set of *literals* consists of all propositional variables and their (classical) negations. A *rule* is an expression of the form

$$L_1 \vee \ldots \vee L_k \vee \sim L_{k+1} \vee \ldots \vee \sim L_l \leftarrow L_{l+1} \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n, \tag{3}$$

where all L_i, $1 \leq i \leq n$, are literals and $0 \leq k \leq l \leq m \leq n$. A *program* is a finite set of rules.

For a rule r of the form shown in (3) we introduce the following notation:

$$\begin{array}{ll} head(r) & = \{L_1, \ldots, L_k\} \cup \{\sim L_{k+1}, \ldots, \sim L_l\} \\ head^+(r) & = \{L_1, \ldots, L_k\} \\ head^-(r) & = \{L_{k+1}, \ldots, L_l\} \\ body(r) & = \{L_{l+1}, \ldots, L_m\} \cup \{\sim L_{m+1}, \ldots, \sim L_n\} \\ body^+(r) & = \{L_{l+1}, \ldots, L_m\} \\ body^-(r) & = \{L_{m+1}, \ldots, L_n\} \end{array}$$

Rule r is said to be a *constraint* if $head(r) = \emptyset$, i.e., constraints are of the form

$$\leftarrow L_{l+1} \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n; \tag{4}$$

it is said to be *classical* if $head^-(r) = body^-(r) = \emptyset$, i.e., classical rules are of the form

$$L_1 \vee \ldots \vee L_k \leftarrow L_{l+1} \wedge \ldots \wedge L_m. \tag{5}$$

One should observe that for all classical rules r we find $head(r) = head^+(r)$ and $body(r) = body^+(r)$. A program is said to be *classical* if all its rules are classical.

Answer Sets for Classical Programs Let $\mathcal{P}$ be a classical program and $\mathcal{M}$ be a satisfiable set of literals, i.e., a set which does not contain a complementary pair A, $\neg A$ of literals. $\mathcal{M}$ is said to be *closed* under $\mathcal{P}$ iff for every (classical) rule $r \in \mathcal{P}$ we find that

$$head(r) \cap \mathcal{M} \neq \emptyset \quad \text{whenever} \quad body(r) \subseteq \mathcal{M}. \tag{6}$$

If we identify $\mathcal{M}$ with an interpretation such that $\mathcal{M}(L) = \mathit{true}$ iff $L \in \mathcal{M}$ for all literals L, then condition (6) states that whenever the body of a (classical) rule r is true under $\mathcal{M}$, then at least one literal in the head of r must be true as well. If condition (6) is satisfied for all rules in $\mathcal{P}$, then $\mathcal{M}$ is a *model* for $\mathcal{P}$. $\mathcal{M}$ is said to be an *answer set* for $\mathcal{P}$ iff $\mathcal{M}$ is minimal among the sets closed under $\mathcal{P}$ (relative to set inclusion).

As an example consider the program $\mathcal{P}$ consisting of the following rules:

$$\begin{array}{l} s \vee r \leftarrow \\ \quad \neg b \leftarrow r \end{array} \tag{7}$$

One may read these rules as *either the sprinkler is on* (s) *or it is raining* (r) and *if it is raining, then the color of the sky is not blue*. The sets $\{s, r, \neg b\}$, $\{s, \neg b\}$, $\{s\}$, $\{r, \neg b\}$ are closed under $\mathcal{P}$, but only the latter two are minimal and, thus, are answer sets. On the other hand, the sets $\emptyset$, $\{r\}$ and $\{r, s\}$ are not closed under $\mathcal{P}$. One should observe that if we add to $\mathcal{P}$ the constraint

$$\leftarrow s \tag{8}$$

then $\{s\}$ is no longer an answer set for the extended program. This example illustrates a general property of constraints: adding a constraint to a program affects its collection of answer sets by eliminating the answer sets which *violate* the constraints.

Reducts Let $\mathcal{P}$ be a program and $\mathcal{M}$ a satisfiable set of literals. The *reduct of* $\mathcal{P}$ *relative to* $\mathcal{M}$, in symbols $\mathcal{P}|_{\mathcal{M}}$, is defined as

$$\{head^+(r) \leftarrow body^+(r) \mid r \in \mathcal{P} \wedge head^-(r) \subseteq \mathcal{M} \wedge body^-(r) \cap \mathcal{M} = \emptyset\}, \tag{9}$$

where the literals in $head^+(r)$ and $body^+(r)$ are disjunctively and conjunctively connected, respectively [10].

As an example consider the program $\mathcal{P} = \{\neg p \leftarrow \sim p\}$. We obtain:

$$\begin{array}{ll} \mathcal{P}|_{\emptyset} & = \{\neg p\} \\ \mathcal{P}|_{\{p\}} & = \emptyset \\ \mathcal{P}|_{\{\neg p\}} & = \{\neg p\} \end{array} \tag{10}$$

One should observe that the reduct of a program relative to a satisfiable set of literals is a classical program.

Answer Sets for Programs Let $\mathcal{P}$ be a program and $\mathcal{M}$ a satisfiable sets of literals. $\mathcal{M}$ is said to be an *answer set* for $\mathcal{P}$ iff $\mathcal{M}$ is an answer set for $\mathcal{P}|_{\mathcal{M}}$. Hence, $\mathcal{M}$ is an answer set for $\mathcal{P}$ iff $\mathcal{M}$ is minimal and closed under $\mathcal{P}|_{\mathcal{M}}$.

Returning to the example discussed in the previous paragraph we observe that $\emptyset$ is not closed under $\mathcal{P}|_{\emptyset} = \{\neg p\}$ as $\neg p \notin \emptyset$, $\{p\}$ is closed under $\mathcal{P}|_{\{p\}} = \emptyset$ but not minimal as $\emptyset \subset \{p\}$, and $\{\neg p\}$ is minimal and closed under $\mathcal{P}_{\{\neg p\}} = \{\neg p\}$. Hence, the only answer set for $\mathcal{P} = \{\neg p \leftarrow \sim p\}$ is $\{\neg p\}$. Hence, the rule $\neg p \leftarrow \sim p$ captures *negation by failure*: if one cannot show that p holds, then $\neg p$ is true.

Interested readers may try to compute answer sets for the following programs:

$$\{p \leftarrow \sim q\},\ \{p \leftarrow \sim \neg p\},\ \{q \leftarrow p \wedge \sim q,\ p \leftarrow,\ q \leftarrow\}.$$

What happens if we delete $q \leftarrow$ from the last example?

First-Order Programs Let $\mathcal{P}$ be a first-order program like the programs shown in (1) and (2). Let $\mathcal{T}$ be a set of terms. The set of *ground instances of $\mathcal{P}$ relative to $\mathcal{T}$*, in symbols $g\mathcal{P}$, is defined as follows:

$$\{r\theta \mid r \in \mathcal{P} \text{ and } \theta \text{ is a ground substitution for } r \text{ with respect to } \mathcal{T}\} \tag{11}$$

There is a bijection between the ground atoms occurring in $g\mathcal{P}$ and a suitable large set of propositional variables and, hence, $g\mathcal{P}$ is equivalent to a propositional program.

4 The Interview Example Revisited

Let $\mathcal{P}$ be the set of rules shown in (1) and (2) and let $\mathcal{T} = \{john\}$. Then, $g\mathcal{P}$ contains the following rules:

$$\begin{array}{ll}
eligible(john) & \leftarrow highGPA(john) \\
eligible(john) & \leftarrow minority(john) \wedge fairGPA(john) \\
\neg eligible(john) & \leftarrow \neg fairGPA(john) \\
interview(john) & \leftarrow \sim eligible(john) \wedge \sim \neg eligible(john) \\
fairGPA(john) & \leftarrow \\
\neg highGPA(john) & \leftarrow
\end{array} \tag{12}$$

Now let

$$\mathcal{M} = \{fairGPA(john),\ \neg highGPA(john),\ interview(john)\}. \tag{13}$$

The reduct of $g\mathcal{P}$ relative to $\mathcal{M}$ contains the following rules:

$$\begin{array}{ll}
eligible(john) & \leftarrow highGPA(john) \\
eligible(john) & \leftarrow minority(john) \wedge fairGPA(john) \\
\neg eligible(john) & \leftarrow \neg fairGPA(john) \\
interview(john) & \leftarrow \\
fairGPA(john) & \leftarrow \\
\neg highGPA(john) & \leftarrow
\end{array} \tag{14}$$

$\mathcal{M}$ is minimal and closed under this reduct and, hence, is the only answer set for $g\mathcal{P}$. Reasoning with respect to this answer set tells the selection committee that it should invite John for an interview.

Now suppose that the selection committee learns during the interview that John belongs to a minority. Let

$$\mathcal{P}' = \mathcal{P} \cup \{minority(john) \leftarrow\}. \tag{15}$$

In this case, $\mathcal{M}$ is no longer an answer set because the new rule is an element in $\mathcal{P}'|_{\mathcal{M}}$ and, consequently, $\mathcal{M}$ is not closed under $\mathcal{P}'|_{\mathcal{M}}$. It is easy to see that

$$\mathcal{M}' = \{fairGPA(john),\ \neg highGPA(john),\ minority(john),\ eligible(john)\} \tag{16}$$

is the only answer set for $\mathcal{P}'$. This example demonstrates that reasoning with respect to answer sets is non-monotonic as the addition of new knowledge may lead to the revision of previously drawn conclusions.

5 ASP Modeling in Practice

We now want to provide an inside on the methodology of how problems are encoded into programs that can be processed by ASP tools like `gringo` and `clasp` [6]. The first tool, `gringo`, takes a program and transforms it to its propositional equivalent - i.e. *grounding* the first-order program as defined in Section 3. For the resulting program, `clasp` is able to compute all answer sets – which refers to the *solving* process and `clasp` is therefore a *solver*.

We have chosen some well known problems taken from constraint programming and SAT-solving to demonstrate step by step the modeling process and its underlying paradigm.[1] From now on, we will use the syntax for programs accepted by `gringo` and explain each new construct used.[2]

Sudoku *The famous number riddle Sudoku represents a constraint problem, typically defined on a* 9×9 *board, where numbers* $1 \ldots 9$ *are placed on each cell. The goal is to complete a given board such that in each row, column, and square the numbers* $1 \ldots 9$ *occur exactly once.*

Now lets develop an encoding of Sudokus as programs. We do not bother about how to solve a Sudoku at all, since we take a truly declarative approach and simply describe the problem. Explaining the game to another person, one would start with defining the board layout, i.e. there is a 9×9 grid, and the

[1] We use the terms *modeling*, *encoding* or *reducing* a problem rather than *programming*, because in ASP one does actually not program in terms of control structures specifying a solving strategy.

[2] We use version 3.0.4 of `gringo`, and 2.1.1 of `clasp`. One should be careful with `gringo` versions ≥ 4, since the syntax is different than the one introduced in this tutorial.

notion of nine rows and columns as well as nine non-overlapping squares of size 3×3. We can express these by the following facts:

$$\begin{array}{l} \mathit{number}(1..9).\ \mathit{row}(0..8).\ \mathit{column}(0..8). \\ \mathit{square}(0, 0..2, 0..2).\ \mathit{square}(1, 0..2, 3..5).\ \mathit{square}(2, 0..2, 6..8). \\ \mathit{square}(3, 3..5, 0..2).\ \mathit{square}(4, 3..5, 3..5).\ \mathit{square}(5, 3..5, 6..8). \\ \mathit{square}(6, 6..8, 0..2).\ \mathit{square}(7, 6..8, 3..5).\ \mathit{square}(8, 6..8, 6..8). \end{array} \tag{17}$$

Rules are separated by a dot. *Facts* are rules with an empty body, in which case the implication symbol is omitted in `gringo` and `clasp`. $\mathit{number}(1..9)$ is a syntactical shorthand notation representing the nine facts $\mathit{number}(1)$, $\mathit{number}(2)$, ..., $\mathit{number}(9)$. We use numbers as constants to make use of these syntactic abbreviations. One could of course also use alpha-numeric constants. Saving these lines to a file `sudoku.lp`, we can already have a look at the resulting program by typing the command:

```
gringo sudoku.lp --text
```

The `--text` option prints the grounded program in a human readable way. One should see now that all facts are listed line by line and shorthand expressions are fully spelled out. But there is no need for actual grounding, as we have not used first-order variables yet. Therefore, we want to change the way the squares are defined.

$$\begin{array}{lll} \mathit{square}(0, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X < 3,\ Y < 3. \\ \mathit{square}(1, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X < 3,\ Y > 2,\ Y < 6. \\ \mathit{square}(2, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X < 3,\ Y > 5. \\ \mathit{square}(3, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 2,\ X < 6,\ Y < 3. \\ \mathit{square}(4, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 2,\ X < 6,\ Y > 2,\ Y < 6. \\ \mathit{square}(5, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 2,\ X < 6,\ Y > 5. \\ \mathit{square}(6, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 5,\ Y < 3. \\ \mathit{square}(7, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 5,\ Y > 2,\ Y < 6. \\ \mathit{square}(8, X, Y) & \text{:-}\ \mathit{row}(X),\ \mathit{column}(Y),\ X > 5,\ Y > 5. \end{array} \tag{18}$$

Now, each square is defined via a rule, specifying the corresponding ranges of the X and Y values, where :- is the syntactic equivalent to the implication symbol ($\leftarrow$) used so far. Replacing the *square* facts by these new rules in the `sudoku.lp` file and calling gringo again, yields the same result as before. This is interesting as we would expect grounded rules where X and Y are replaced by the constants $1 \ldots 9$. Instead we only find the grounded *square* atoms in the output as intended. It is easy to see that with a naive grounding approach, the resulting grounded program for these nine rules and the nine constants would already be quite large. The `gringo` grounder applies highly sophisticated grounding strategies. In our case, this even includes the full evaluation of the rule bodies. One should note that this is only possible in our case since in (17) we used the naturals numbers $0 \ldots 8$ as constants for rows and columns and the comparison relations ($<$, $>$, $=$, $!=$, $>=$, $<=$) are already predefined for natural numbers. We are not going

into details of grounding strategies here; the interesting reader might want to look in [8] for details.

So far we have just encoded the basic Sudoku board. Explaining the game further, one would continue with providing the rules of the game, viz. that (a) in each of the 81 cells exactly one number in $\{1 \dots 9\}$ can be placed, and (b) in each row, column and square a number is allowed to occur only once. Regarding (a), we introduce the rules

$$
\begin{aligned}
\mathit{cell}(X,Y,1) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\ \texttt{not } \mathit{cell}(X,Y,5),\\
& \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,2) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\ \texttt{not } \mathit{cell}(X,Y,5),\\
& \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,3) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,4),\ \texttt{not } \mathit{cell}(X,Y,5),\\
& \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,4) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,5),\\
& \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,5) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\\
& \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,6) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\\
& \texttt{not } \mathit{cell}(X,Y,5),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,7) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\\
& \texttt{not } \mathit{cell}(X,Y,5),\ \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,8),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,8) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\\
& \texttt{not } \mathit{cell}(X,Y,5),\ \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,9).\\
\mathit{cell}(X,Y,9) \text{ :- } & \mathit{row}(X),\ \mathit{column}(Y),\\
& \texttt{not } \mathit{cell}(X,Y,1),\ \texttt{not } \mathit{cell}(X,Y,2),\ \texttt{not } \mathit{cell}(X,Y,3),\ \texttt{not } \mathit{cell}(X,Y,4),\\
& \texttt{not } \mathit{cell}(X,Y,5),\ \texttt{not } \mathit{cell}(X,Y,6),\ \texttt{not } \mathit{cell}(X,Y,7),\ \texttt{not } \mathit{cell}(X,Y,8).
\end{aligned}
$$

each expressing that we can assert a number, e.g. 1, to the cell at position (X, Y) if we can ensure that none of the remaining numbers $2 \dots 9$ is already asserted to (X, Y). We do so by using default negation `not`, the syntactic symbol for $\sim$ as used in `gringo` and `clasp`.

Now lets have a closer look on the semantics of these rules, as they have a very special character and role in programs. Assume we only have the first rule, then applying the rule yields a new fact stating that on position (X, Y) number 1 is placed, if X is a row and Y is a column and none of the remaining numbers $2 \dots 9$ is already assigned to position (X, Y). Since we do not have further initial knowledge telling us that there is already another number placed on cell (X, Y), the rule is applicable for every position (X, Y) leading to a grid full of 1's. Considering all nine rules now, we are faced with some *non-determinism*, as for each cell we need to *guess* whether we place a 1 by applying the first rule,

or lets say a 7 by applying the seventh rule. Note that the order of rules in a program does not play any role. In consequence, these nine rules define our (complete) *search space*, namely all 9^{81} number placements possible on a 9×9 board. Usually, such rules are called *generating* or *guessing* rules and represent and essential part of program encodings - we will later discuss this in more detail.

Apparently, we need to encode the Sudoku rules in (b) in order to suppress those guesses not representing a proper number assignment. We define the constraints

$$\begin{array}{l} \text{:- } cell(X, Y1, N),\ cell(X, Y2, N), Y1 \mathrel{!}= Y2. \\ \text{:- } cell(X1, Y, N),\ cell(X2, Y, N), X1 \mathrel{!}= X2. \\ \#hide. \\ \#show\ cell/3. \end{array} \tag{19}$$

which express that we would derive *false*, whenever we have the same number twice in one row (first rule) or column (second rules), respectively. With the help of the statements $\#hide$ and $\#show\ cell/3$ `clasp` is instructed to print only the predicate $cell/3$ and to hide the other elements of an answer set.

Furthermore, we need a notion for a number to be in a square, and consequently rule out answer sets where a number does not occur in every square.

$$\begin{array}{rl} in\_square(S, N) & \text{:- } cell(X, Y, N),\ square(S, X, Y). \\ & \text{:- } number(N),\ \texttt{not}\ in\_square(S, N),\ square(S, \_, \_). \end{array}$$

Predicate *in_square* holds for square S and number N if we find N in a cell (X, Y) which belongs to square S. The second rule, i.e., the constraint, ensures that every number occurs in every square. Note that underscores, as occurring in the very last *square* predicate, represent anonymous variables which can be used if their assignment is not important, i.e. if they are not used in another predicate of the rule.

One might ask why $square(S, \_, \_)$ is actually needed? Are the first two body literals not sufficient? One can try dropping it from the constraint and call the grounder again. The grounder will state that the variable S is *unsafe* and, therefore, the rule is *unsafe*. In fact, `gringo` requires a program to be *safe*, which is the case if every rule occurring in the program is *safe*. A rule is *safe*, if every variable occurring in the rule occurs in some of the rule's positive body literals. State-of-the-art grounders like `gringo` require programs to be safe in order to guarantee termination.

Putting things together, we have fully encoded all rules of Sudoku. One should observe that we have not defined anything instructing how an instance of Sudoku needs to be solved – and we will not do so subsequently. This should emphasize the truly declarative approach. Saving the elaborated rules again to file `sudoku.lp` and calling

```
gringo sudoku.lp | clasp
```

will ground the program and pipe the output to `clasp`. We should get an output similar to the following:

```
clasp version 2.1.1
Reading from stdin
Solving...
Answer: 1
cell(0,1,6) cell(0,2,5) cell(0,4,3) cell(0,8,7) cell(1,0,1)
cell(1,2,7) cell(1,5,5) cell(2,2,8) cell(2,8,1) cell(3,3,2)
cell(3,5,1) cell(4,2,6) cell(4,4,8) cell(4,6,3) cell(5,3,5)
cell(5,5,3) cell(6,0,5) cell(6,6,6) cell(7,3,8) cell(7,6,4)
cell(7,8,3) cell(8,0,4) cell(8,4,7) cell(8,6,2) cell(8,7,1)
cell(0,0,2) cell(0,3,1) cell(0,5,8) cell(0,6,9) cell(0,7,4)
cell(1,1,4) cell(1,3,9) cell(1,4,2) cell(1,6,8) cell(1,7,3)
cell(1,8,6) cell(2,0,3) cell(2,1,9) cell(2,3,6) cell(2,4,4)
cell(2,5,7) cell(2,6,5) cell(2,7,2) cell(3,0,8) cell(3,1,5)
cell(3,2,3) cell(3,4,9) cell(3,6,7) cell(3,7,6) cell(3,8,4)
cell(4,0,9) cell(4,1,1) cell(4,3,7) cell(4,5,4) cell(4,7,5)
cell(4,8,2) cell(5,0,7) cell(5,1,2) cell(5,2,4) cell(5,4,6)
cell(5,6,1) cell(5,7,8) cell(5,8,9) cell(6,1,3) cell(6,2,2)
cell(6,3,4) cell(6,4,1) cell(6,5,9) cell(6,7,7) cell(6,8,8)
cell(7,0,6) cell(7,1,7) cell(7,2,1) cell(7,4,5) cell(7,5,2)
cell(7,7,9) cell(8,1,8) cell(8,2,9) cell(8,3,3) cell(8,5,6)
cell(8,8,5)
SATISFIABLE
Models : 1+
Time : 0.124s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.030ss
```

If `clasp` is called without options, it returns one answer set in case of satisfiability and indicates with `Models: 1+` that there are additional answer sets. The call

```
gringo sudoku.lp | clasp --number 5
```

requests an enumeration of up to five answer sets. Using `--number 0` requests all answer sets and should be used carefully since, like in our case, there are many possibilities to fill an empty Sudoku board.

Our encoding so far produces fully filled Sudoku boards, which all agree on the defined constraints and therefore represent solved Sudoku games. However, we do not yet solve a given partially filled Sudoku board. We need to provide a so-called *problem instance*. For example, we add subsequent *cell* facts.

$$
\begin{array}{l}
\mathit{cell}(0,0,3).\ \mathit{cell}(0,4,8).\ \mathit{cell}(0,6,6).\ \mathit{cell}(0,8,7).\\
\mathit{cell}(1,1,1).\ \mathit{cell}(1,6,4).\ \mathit{cell}(1,8,9).\\
\mathit{cell}(2,0,8).\ \mathit{cell}(2,1,9).\ \mathit{cell}(2,4,6).\ \mathit{cell}(2,5,7).\\
\mathit{cell}(3,1,6).\ \mathit{cell}(3,3,1).\ \mathit{cell}(3,4,9).\ \mathit{cell}(3,6,7).\\
\mathit{cell}(4,2,9).\ \mathit{cell}(4,3,6).\ \mathit{cell}(4,4,5).\ \mathit{cell}(4,8,2).\\
\mathit{cell}(5,2,2).\ \mathit{cell}(5,7,1).\\
\mathit{cell}(6,1,5).\ \mathit{cell}(6,4,4).\ \mathit{cell}(6,8,3).\\
\mathit{cell}(7,1,4).\ \mathit{cell}(7,3,2).\ \mathit{cell}(7,7,9).\ \mathit{cell}(7,8,8).\\
\mathit{cell}(8,1,8).\ \mathit{cell}(8,2,6).\ \mathit{cell}(8,4,3).\ \mathit{cell}(8,6,1).
\end{array}
$$

We save the assertions to a new file `suoku_instance.lp` and call `gringo` and `clasp` again:

```
gringo sudoku.lp sudoku_instance.lp | clasp --number 0
```

With `--number 0` we also instruct `clasp` to enumerate all models, which in our case should be exactly one.

Guess & Check Paradigm Let us reconsider the involved modeling steps in the Sudoku example. We started by specifying a fixed board layout by providing appropriate facts. The possible actions – placing exactly one number on each cell of the board – were modeled as *guessing* rules. We can see that part of an answer set program as the *guessing part*, where the search space for answer sets is defined. Secondly, we encoded constraints in order to *check* whether a generated solution is an answer set according to the rules of the Sudoku game.

This approach of modeling is often referred to as the *guess & check* or *generate-and-test paradigm* (see e.g. [8]). This is also motivated by *NP* problems, having a non-deterministical guessing of prospective solutions and subsequent checking. The interested reader might look at [12], from whom the paradigm originates.

At the end we added to our problem specification a concrete *problem instance*. The problem specification allows us to add any instance of some 9×9 Sudoku game. Therefore, such problem specifications are said to be *uniform*. We will use this separation into *problem specification* following the guess and check paradigm, and *problem instance* for all subsequent examples.

Graph Coloring We want to illustrate the *guess & check* paradigm again with a very concise and nice example - the *graph coloring problem*. In detail, *the problem asks whether there is some coloring of the nodes of a given undirected graph using n colors, such that no two nodes connected via an edge share the same color*. We restrict the example to $n = 3$. For encoding the problem we need the notion of *color*, *node*, *edge* and the *coloring* of nodes. We do so and stick to our paradigm by

$$
\begin{aligned}
&\mathit{color}(\mathit{green}).\ \mathit{color}(\mathit{red}).\ \mathit{color}(\mathit{blue}).\\
&\mathit{coloring}(X,\mathit{green}) \text{ :- } \mathit{node}(X),\ \texttt{not}\ \mathit{coloring}(X,\mathit{red}),\ \texttt{not}\ \mathit{coloring}(X,\mathit{blue}).\\
&\mathit{coloring}(X,\mathit{red}) \quad \text{ :- } \mathit{node}(X),\ \texttt{not}\ \mathit{coloring}(X,\mathit{green}),\ \texttt{not}\ \mathit{coloring}(X,\mathit{blue}).\\
&\mathit{coloring}(X,\mathit{blue}) \quad \text{ :- } \mathit{node}(X),\ \texttt{not}\ \mathit{coloring}(X,\mathit{green}),\ \texttt{not}\ \mathit{coloring}(X,\mathit{red}).\\
&\qquad\qquad\quad \text{ :- } \mathit{coloring}(X1,C),\ \mathit{coloring}(X2,C),\ \mathit{edge}(X1,X2).
\end{aligned}
$$

which fully specifies the problem. We have three colors specified in the first three facts, three guessing rules as well as a single constraint eliminating solutions where two nodes in an edge relation have identical coloring. We encode the *color-guess* analogously to the *number-guess* in the previous Sudoku example.

Because guessing is an essential part, the syntax offers so-called *choice rules*. As defined in `gringo`'s syntax documentation [7], choice rules are of the form

$$\{a_1, \ldots, a_m\} \text{ :- } a_{m+1}, \ldots, a_n, \texttt{not}\ a_{n+1}, \ldots, \texttt{not}\ a_k.$$

where $0 \leq m \leq n \leq k$, and each a_i is an atom for $0 \leq i \leq k$. This allows us to derive any subset of $\{a_1, \ldots, a_m\}$ (head atoms), provided that the body is satisfied. We could encode the following for coloring nodes:

$$\{coloring(X, green), coloring(X, red), coloring(X, blue)\} \text{ :- } node(X).$$

But since we can derive any subset for some node, this would yield nodes having more than one color assigned, or even no color at all. Therefore, the syntax allows an extension to put cardinality restrictions on the subset choice:

$$l\ \{a_1, \ldots, a_m\}\ u \text{ :- } a_{m+1}, \ldots, a_n, \texttt{not}\ a_{n+1}, \ldots, \texttt{not}\ a_k.$$

forcing the subset size to be within the lower bound l and upper bound u. I.e., we assign exactly 1 color to each node by

$$1\ \{coloring(X, green), coloring(X, red), coloring(X, blue)\}\ 1 \text{ :- } node(X). \quad (20)$$

ensuring that whenever we have some node X, we are allowed to add exactly one of the three head atoms to an answer set. We can even generalize rule (20) such that we do not need to partially instantiate the *coloring* predicate in the head. The syntax therefore offers so-called *conditional literals* of the form $l : l_1 : \ldots : l_n$, with $0 \leq i \leq n$. While grounding, l is instantiated under the conditions l_1 to l_n. It can be used for our purpose as follows:

$$1\ \{coloring(X, C)\ :\ color(C)\}\ 1 \text{ :- } node(X). \quad (21)$$

The grounding of expression $\{coloring(X, C)\ :\ color(C)\}$ expands to the one in (20). I.e. the set is generated by instantiating the *coloring* atom where C must be some *color*. We end up with a very compact encoding of the graph coloring problem:

$$\begin{array}{l} color(green).\ color(red).\ color(blue). \\ 1\ \{coloring(X, C)\ :\ color(C)\}\ 1 \text{ :- } node(X). \\ \text{:- } coloring(X1, C), coloring(X2, C), edge(X1, X2). \end{array}$$

For the problem instance, we provide a small graph with 4 nodes and 4 edges:

$$\begin{array}{l} node(1..4). \\ edge(1, 2).\ edge(1, 3).\ edge(3, 2).\ edge(3, 4). \end{array}$$

Saving the problem description to `coloring.lp` and the problem instance to `simple-graph.lp`, we can first get again an impression on the grounder's work:

```
gringo coloring.lp simple-graph.lp --text
```

This yields the following output besides existing facts:

```
1#count{coloring(4,blue),coloring(4,red),coloring(4,green)}1.
1#count{coloring(3,blue),coloring(3,red),coloring(3,green)}1.
1#count{coloring(2,blue),coloring(2,red),coloring(2,green)}1.
1#count{coloring(1,blue),coloring(1,red),coloring(1,green)}1.
:-coloring(3,green),coloring(4,green).
:-coloring(3,red),coloring(4,red).
:-coloring(3,blue),coloring(4,blue).
:-coloring(3,green),coloring(2,green).
:-coloring(3,red),coloring(2,red).
:-coloring(3,blue),coloring(2,blue).
:-coloring(1,green),coloring(3,green).
:-coloring(1,red),coloring(3,red).
:-coloring(1,blue),coloring(3,blue).
:-coloring(1,green),coloring(2,green).
:-coloring(1,red),coloring(2,red).
:-coloring(1,blue),coloring(2,blue).
```

It is interesting to see that the resulting program is already partially evaluated. For example, the *edge* atom in all constraints is missing, which is fine since it was merely used for instantiation of the other two *coloring* atoms. Also the choice rule was turned into four constraints, each for one of the nodes. I.e. the constraint

$$1\ \#count\{coloring(1, blue), coloring(1, red), coloring(1, green)\}\ 1.$$

represents the instantiation of rule (20) for node 1, where the *node* predicate in the body was instantiated with 1 and, therefore, omitted since, in this case, we only care about the head. With this constraint we fail if the *#count* function counts more than 1 occurrences of colorings for node 1 or less, respectively. We compute all answer sets for the union of the problem description (`coloring.lp`) and problem instance (`simple-graph.lp`) with:

```
gringo coloring.lp simple-graph.lp | clasp --number 0
```

For the provided graph, 12 colorings exist. The interested reader should try with an encoding of a more complex graph, e.g. the Petersen graph [15,16], in order to get an impression of `clasp`'s performance.

Traveling Salesman Problem Another combinatorial problem, but with an optimization aspect, is the so-called *traveling salesman problem (TSP)*. We want to demonstrate that answer set programming can also be applied to find optimal solutions. *A salesman is requested to visit some pre-defined cities. In order to be as efficient as possible, he wants to visit every city only once, as well as to travel the shortest roundtrip visiting all cities starting and ending in the same city.*

The problem can be separated into, (*a*) finding roundtrips beginning from and ending in the same city visiting all other cities only once, and (*b*) computing the length of each roundtrip in order to find the shortest. The first is known

as the problem of finding *Hamiltonian cycles* in graphs, another well-known *NP*-complete problem. Therefore, we are faced with the optimization aspect to find the minimal Hamiltonian cycle in a given weighted graph, where the nodes in the graph represent our cities and labeled edges between nodes represent connections (e.g. highways or railways) between the cities. We encode the *TSP* problem description in exactly these steps, starting with appropriate guessing rules:

$$1\ \{cycle(X,Y) : edge(X,Y)\}\ 1 \text{ :- } node(X). \tag{22}$$
$$1\ \{cycle(X,Y) : edge(X,Y)\}\ 1 \text{ :- } node(Y). \tag{23}$$

Apparently, for a node to be in a cycle it must have an incoming edge as well as an outgoing edge. In the case of Hamiltonian cycles, a node in the cycle has exactly one incoming and one outgoing edge. This is specified in the rules (22) and (23). Again we use a cardinality restriction (exactly 1) on the head's choice expression $\{cycle(X,Y) : edge(X,Y)\}$. I.e., for every node X, in the first rule we choose exactly 1 of the instantiated $cycle(X,Y)$ literals, which the grounder instantiates with some Y whenever the conditional $edge(X,Y)$ is fulfilled. In (22) we choose an outgoing edge and derive that it belongs to the cycle, whereas in (23) we do so for incoming edges. Since for every node we non-deterministically pick one outgoing and incoming edge, we might have generated a cycle or not. To rule out model candidates not representing cycles, we check whether every node can be reached by every other node and exclude models where there is an unreachable node.

$$reachable(Y) \text{ :- } cycle(s,Y). \tag{24}$$
$$reachable(Y) \text{ :- } cycle(X,Y),\ reachable(X). \tag{25}$$
$$\text{:- } node(X),\ \texttt{not}\ reachable(X). \tag{26}$$

Rule (24) and (25) define the notion for a node being *reachable* from all other nodes. It is defined recursively, i.e., a node Y is reachable if it is a direct neighbor of the starting node s (recursion base case). Furthermore, we conclude that node Y is reachable if for another reachable node X we find (X,Y) is in the *cycle*. We simply define the constraint (26) to fail whenever we have a node X for which we can not demonstrate its reachability. Saving (22)-(26) to `hamiltonian.lp` we can compute Hamiltonian cycles for some given graph, e.g.

$$\begin{array}{ll}
node(dresden). & node(petersburg). \\
node(novosibirsk). & node(stavropol). \\
node(moscow). & edge(stavropol, novosibirsk). \\
edge(dresden, moscow). & edge(moscow, petersburg). \\
edge(dresden, petersburg). & edge(moscow, stavropol). \\
edge(dresden, stavropol). & edge(moscow, novosibirsk). \\
edge(petersburg, novosibirsk). & edge(Y,X) \text{ :- } edge(X,Y).
\end{array} \tag{27}$$

encodes, without edge weights, the graph depicted in Figure **??**. The last rule models symmetry of the *edge* relation in order to only provide one direction via

the facts. We save (27) to `map.lp` and call:

```
gringo -c s=dresden hamiltonian.lp map.lp | clasp --number 0
```

With `-c s=dresden` the constant s in (24) is rewritten with *dresden*, in order to define that we want to start from the city of Dresden. We should receive 8 answer sets, each including instantiated *cycle* literals representing edges of the Hamiltonian cycle, similar as in subsequent output.

```
clasp version 2.1.1
Reading from stdin
Solving...
Answer: 1
cycle(dresden,stavropol) cycle(moscow,dresden)
cycle(stavropol,novosibirsk) cycle(petersburg,moscow)
cycle(novosibirsk,petersburg)
...
Answer: 8
cycle(dresden,moscow) cycle(moscow,stavropol)
cycle(petersburg,dresden) cycle(stavropol,novosibirsk)
cycle(novosibirsk,petersburg)
SATISFIABLE
Models : 8
Time : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
```

The interested reader might try `simple-graph.lp` from the previous example: Are there any Hamiltonian cycles?

We can compute cycles now, however, we have not respected the optimizing aspect of the *TSP* problem yet – i.e. Hamiltonian cycles with minimal overall distance. We need additional knowledge representing the distances between the cities, by defining the ternary predicate *distance* and providing the following instances:

$$
\begin{array}{ll}
\mathit{distance}(\mathit{dresden}, \mathit{moscow}, 20). & \mathit{distance}(\mathit{dresden}, \mathit{petersburg}, 18). \\
\mathit{distance}(\mathit{dresden}, \mathit{stavropol}, 26). & \mathit{distance}(\mathit{moscow}, \mathit{stavropol}, 14). \\
\mathit{distance}(\mathit{moscow}, \mathit{petersburg}, 7). & \mathit{distance}(\mathit{moscow}, \mathit{novosibirsk}, 34). \\
\mathit{distance}(\mathit{novosibirsk}, \mathit{petersburg}, 38). & \mathit{distance}(\mathit{stavropol}, \mathit{novosibirsk}, 36). \\
\mathit{distance}(X, Y, C) \text{ :- } \mathit{distance}(Y, X, C). &
\end{array}
$$

Intuitively, $\mathit{distance}(\mathit{dresden}, \mathit{moscow}, 20)$ represents the fact, that the distance between Dresden and Moscow is 20. And to keep it less complex, we express symmetry of distances via the last rule. Since we want to minimize a cycle's overall length, we need to know the length of each cycle. The syntax offers some build-in constructs for such purposes called *aggregate functions*. They are used within *aggregate atoms*, which are atoms of the form

$$l \ \#A[l_1 = w_1, \ldots, l_n = w_n] \ u.$$

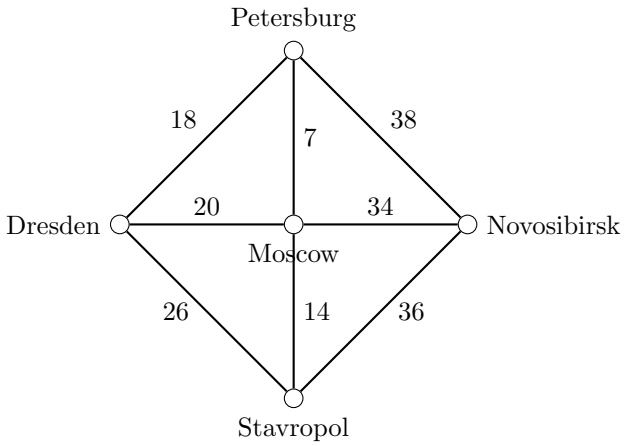

Figure1. Example graph representing cities and their connections.

Each literal l_i has an assigned weight w_i which is 1 if not given explicitly, and the function A is applied to the weights of all literals l_i, $1 \leq i \leq n$. Aggregate atoms can be used as constraints, or on the right hand-side of some variable assignment. For example, we use $\#sum$ in order to assign the sum of all distances between cities in some hamiltonian cycle.

$$circumference(N) \text{ :- } N = \#sum\ [cycle(X,Y) : distance(X,Y,C) = C]. \tag{28}$$

Lets disassemble rule (**??**). We know conditional literals already from choice rules like the one in (22), i.e. the expression $cycle(X,Y) : distance(X,Y,C)$ will be evaluated while grounding and yields instantiated $cycle(X,Y)$ atoms whenever there is a corresponding $distance(X,Y,C)$. The only difference now, is that we take the distance value C and use it as the weight. For $cycle(dresden, stavropol)$ and $cycle(mosow, dresden)$, we obtain the grounded rule

$$\begin{aligned} circumference(46) \text{ :- } 46 = \#sum[&cycle(dresden, stavropol) = 26,\\ &cycle(moscow, dresden) = 20]. \end{aligned}$$

It should be clear to see now how $\#sum$ yields the sum 46, which is assigned to variable N and therefore the new fact $circumference(46)$ is introduced. We can save the distance facts and rule (**??**) to `distances.lp` and again request all answer sets via:

```
gringo -c s=dresden hamiltonian.lp map.lp distances.lp | clasp --number 0
```

One should see that each answer set now also includes *circumference* denoting the cycle's overall length. We now only need to instruct `clasp` to provide only the answer set having minimal *circumference* value. We do so using an objective optimization function, in our case $\#minimize$.

$$\#minimize\ [circumference(N) = N]. \tag{29}$$

It operates on the same input as aggregate functions, but it does actually not represent a constraint nor a rule. It simply instructs `clasp` to print answer sets with optimal value, in our case the hamiltonian cycle with overall length 121.

```
clasp version 2.1.1
Reading from stdin
Solving...
Answer: 1
cycle(novosibirsk,stavropol) cycle(petersburg,novosibirsk)
cycle(moscow,petersburg) cycle(stavropol,dresden)
cycle(dresden,moscow) circumference(127)
Optimization: 127
Answer: 2
cycle(novosibirsk,stavropol) cycle(petersburg,novosibirsk)
cycle(moscow,dresden) cycle(stavropol,moscow)
cycle(dresden,petersburg) circumference(126)
Optimization: 126
Answer: 3
cycle(novosibirsk,stavropol) cycle(petersburg,moscow)
cycle(moscow,novosibirsk) cycle(stavropol,dresden)
cycle(dresden,petersburg) circumference(121)
Optimization: 121
OPTIMUM FOUND
Models : 1
 Enumerated: 3
 Optimum : yes
Optimization: 121
Time : 0.021s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.010s
```

We suggest to have a look at `clasp`'s optimization features documented in [7,8], which is quite involved since there are many command options instructing `clasp` on how to deal with optimal solutions. It is also worth to have a look at all other aggregate functions, namely $\#count$, $\#avg$, $\#min$ and $\#max$.

We provide some more problems, which we suggest the interested reader to model and encode in a similar fashion as we have done in the previous examples. The planning problems *Cannibals and Missionaries* and *Towers of Hanoi* might be slightly more involved. Moreover, one can find more problems, as for example in [9,11,8].

The Seating Problem *Workshop organizers want to arrange the seating of a social dinner in such a way that participants share a table only together with participants they don't know (yet), respectively participants for who it is known that they know each other should not sit at the same table. There are n tables with some fixed number m of chairs available. Naturally, participants can only*

sit at one table and chair, however, tables do not need to be fully seated in case there are less than $m \times n$ seats.

Cannibals and Missionaries We can also encode planning problems, like the famous one proposed by [2]. *Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.*

Towers of Hanoi Another famous and historic planning problem is the hanoi tower problem. *It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:*

- *Only one disk can be moved at a time.*
- *Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.*
- *No disk may be placed on top of a smaller disk.*

6 Conclusion

In this tutorial, we introduced the theoretical background by means of the interview example, i.e. the notion of a *program*, *rule types*, *answer sets* for programs, as well as the *reduct* of a program. Equipped with the theory, we had a closer practical look at some encodings of problems using the syntax of `gringo` in order to solve the problem with the `clasp` solver. Both tools emerged from the potassco project [6]. The introduced language constructs should cover the basics as far as possible, such that the interested reader should be able to model the three open problems from the previous section. All these problems and especially the provided problem instances are quite small, though `clasp` is able to deal with extreme large problem instances, as annual competitions demonstrate [1].

Especially for readers familiar with logic programming (e.g. in *prolog*), it might have immediately become clear, that we do not have to care about the ordering of rules in any way, which imposes the answer set approach to be truly declarative; and, as [8] point out, is a real separation of *logic* and *control* compared to other approaches. Also one might miss data structures as they are known as nested terms e.g. in prolog. In answer set programs n-ary tuples and (flat, since grounded) terms are the choice for data structures.

Further reading We have not covered details, for example, on using arithmetics or other language extensions, which are fully covered in [7,8]. In general we used the syntax accepted by `gringo` in version 3.0, as documented in [7]. Also, entire

lectures could be dedicated to grounding and solving strategies, which is due to the fact that modern solvers use state-of-the-art SAT techniques. In [8] these insights are provided in detail.

We should also note, that although we used the potassco tools in this tutorial, there are several more systems available and used in academical as well as industrial applications. Just to mention, there is the *dlv* system introduced in [11] and professionally maintained by DLVSYSTEM s.r.l., some spin-off company of the university of calabria. Moreover there is the *Smodels* solver and corresponding grounder *Lparse*, which was the very first implementation of the stable model semantics for logic programs, developed by [14]. In fact Lparse imposes some unofficial standard syntax, which is also accepted as input by `clasp`. There is evan an answer set extension to prolog - *Answer Set Prolog (AnsProlog)*. In their comprehensive work, [9] use AnsProlog to introduce the *ASP* approach.

Among these systems, unfortunately their syntax is not standardized yet. Therefore the syntactic structures introduced in this tutorial may not work, or lead to other results when using other tools. However, there is an *ASP* standardization working group elaborating a common input language definition [4]. New versions of `gringo`, up from 4.0, stick to the current version of the standard.

References

1. Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, et al. The fourth answer set programming competition: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning*, pages 42–53. Springer, 2013.
2. Saul Amarel. On representations of problems of reasoning about actions. *Machine intelligence*, 3(3):131–171, 1968.
3. K. R. Apt. *From Logic to Logic Programming.* Prentice Hall, London, 1997.
4. Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format, 2013.
5. M. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer, Berlin, 2nd edition, 1996.
6. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. preliminary draft, 2010.
8. Martin Gebser, Benjamin Kaufmann Roland Kaminski, and Torsten Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.
9. M. Gelfond and Y. Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 2014.
10. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

11. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.
12. Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002.
13. J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, Heidelberg, 1987.
14. Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer, 1997.
15. Julius Petersen. Die theorie der regulären graphs. *Acta Mathematica*, 15(1):193–220, 1891.
16. Eric W. Weisstein. Petersen graph. In *MathWorld–A Wolfram Web Resource.* Visited on 11/04/14. `http://mathworld.wolfram.com/PetersenGraph.html`.