KBS for Business Informatics, 4.0 VU, 184.698
Specification Project 2: Answer Set Programming

Winter Term 2017/18

Important: This task description is valid only in the semester specified above! In other semesters, do not assume that the task description remains the same, unless the sheet is redistributed with an updated semester specification!

The deadline for this project is **Wednesday, January 3, 23:55**. You can submit multiple times. Please submit early and do not wait until the last moment.

You will be awarded up to **8 points** for completing this assignment.

For questions please use the appropriate TISS forum. Questions that reveal (part of) your solution should be discussed privately during the tutor sessions (see the timeslots listed in TUWEL) or send an email to `kbsbi-2017w@kr.tuwien.ac.at`.

For this assignment you will encode a combinatorial problem using answer set programming. To test and run your program you will use `DLV`, which is a solver for disjunctive extended logic programs. Use the most current version of `DLV` (`2012-12-17`), which can be be obtained from its official homepage.[1] Documentation on specific features of `DLV` and aggregate predicates in particular can be found there as well.[2]

1 Problem specification

1.1 Intuition

You are given a positioning description of several trays in a snack vending machine. The trays are of different forms and sizes, that is, each is made up of one or more cells on a grid representing the machine. Each tray can be (partially) filled with snacks; specifically, each cell either contains a snack or is empty. Since the trays are hung inside the machine, gravity acts upon the snacks and the preserve will always spread within the tray so that it is filled up to a certain level. The level is the same throughout the entire tray. Trays will never overlap and the cells of each tray will be connected.

For a given number of cells per row and column that contain snacks, your task is to determine those cells on the grid that are filled with snacks. Moreover, your task is to store snacks of different weights, i.e. for each filled cell you can decide the weight of the corresponding snack.

See Figure 1 for a simple example with two trays assuming every snack has weight of 1. The lower subfigure shows the correct solution, that is, all cells belonging to trays are filled. Figure 2 shows a more involved example which demonstrates the constraint imposed by gravity: In row 2 there are 2 filled cells. It would be invalid to only have snacks in either coordinate $(1, 2)$ or $(3, 2)$. Therefore both must be filled in a correct solution.

1.2 Formal description

More formally, a solution to this problem consists of an injective function $f : \{1, \ldots, n\} \times \{1, \ldots, m\} \to \{0, \ldots, 4\}$ assigning each cell a snack of some weight (assuming n columns, m rows and integer weights from 1 up to 4, where weight 0 corresponds to an empty cell) s.t. f is

(1) *balanced*, i.e., the sum of assigned weights in each row resp. column matches the given number $w_{0,j}$ resp. $w_{i,0}$.

$$\sum_{i=1}^{n} f(i, j) = w_{0,j} \quad \text{for any } 1 \le j \le m$$

[1] `http://www.dlvsystem.com/dlv/`

[2] `http://www.dlvsystem.com/html/DLV_User_Manual.html`

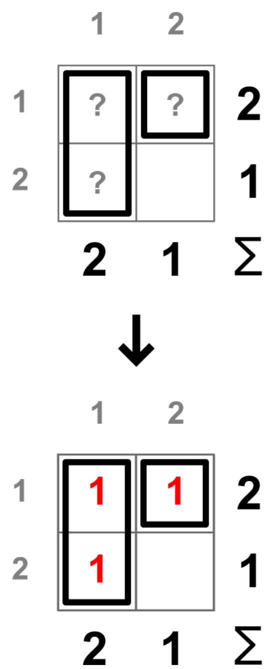

Figure 1: Example with two trays

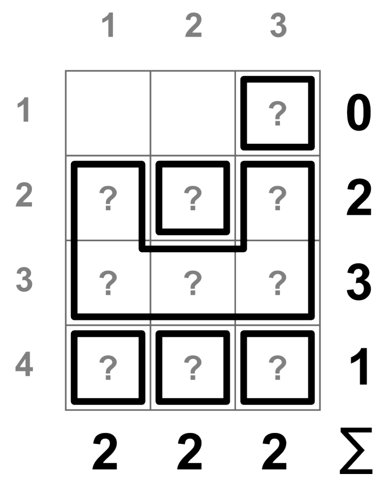

Figure 2: An example involving gravity

$$\sum_{j=1}^{m} f(i,j) = w_{i,0} \quad \text{for any } 1 \leq i \leq n$$

(2) satisfies the rules of *gravity*, i.e., for two consecutive cells on the y-axis that are part of the same tray T, it is not allowed that the predecessor is filled and the successor is not. That is, it is not allowed that a cell is filled if the cell *below* is not, given that both cells belong to the same tray T. Moreover, gravity also acts upon the x-axis, i.e., within one tray T it is forbidden that one cell at some row and column is filled while an other one with different column number but same row number is not.

$$f(i,j) > 0 \implies f(i,j') > 0 \quad \text{for any} \quad \{(i,j),(i,j')\} \subseteq T, j' > j$$
$$f(i,j) > 0 \implies f(i',j) > 0 \quad \text{for any} \quad \{(i,j),(i',j)\} \subseteq T, i' \neq i$$

2 Problem encoding (6 points)

Your assignment is to write a logic program for `DLV` that takes the description of trays and the desired weight sum for each row and each column. The answer sets of your program should describe all possible fillings of the trays such that the constraints for each row and each column are satisfied.

Use the following predicates for the input specification:

`right(C1,C2)` represents that column `C1` is followed by column `C2`.
Ex.: `right(1,2).`

`below(R1,R2)` designates row `R1` to be followed by row `R2`.
Ex.: `below(1,2).`

`column(C,W)` designates that in column `C` there are filled cells of weight `W` (i.e. $w_{C,0} = W$).
Ex.: `column(1,2).`

row(R,W) expresses that row R exhibits filled cells of weight W (i.e. $w_{0,R} = W$).
  Ex.: `row(1,1).`

tray(T,C,R) designates that the cell at column C and row R is part of the tray labelled T (i.e., all cells with the same T together form a single tray).
  Ex.: `tray(1,1,2).`

Use the ternary predicate `snacks` as output of your encoding; `snack(C,R,W)` then indicates that the cell located at column C and row R is filled with a snack of weight $W > 0$. Note that `snack(C,R,0)` is not allowed for any column C and row R.

Note that the grid always has its origin at the absolute column-row-position $(1,1)$, which specifies the upper-left corner of the machine. This means that cells located at (c,r) such that $r > 1$ are below of $(1,1)$, and cells at (c,r) with $c > 1$ are to the right of the origin. Consider this when modelling the constraint that gravity exerts.

2.1 Writing tests (compulsory)

Think of some test cases before you start to encode the problem. Once you have written the actual program you will be able to check whether it works as expected. You should design at least 5 test cases and save them in separate files, name these files vending_test*n*.dl, where n is a number ($1 \leq n \leq 5$).

As an example, consider the respective test case for the machine and constraints depicted in Figure 1:

```
right(1,2).   below(1,2).
tray(1,1,1). tray(1,1,2). tray(2,2,1).
row(1,2). row(2,1).
column(1,2). column(2,1).
```

Later you can run your test cases along with your problem encoding using DLV, e.g.:

```
$ dlv vending_test1.dl vending_guess.dl vending_balance.dl vending_gravity.dl
```

which should produce the approriate answer set(s). One of them is, e.g.,

```
{ tray(1,1,1), tray(1,1,2), tray(2,2,1),
  row(1,2), row(2,1),
  column(1,2), column(2,1),
  right(1,2,1), below(1,2),
  snack(1,1,1),  snack(1,2,1),  snack(2,1,1) }.
```

Now you are ready to write the actual answers-set program by using the *Guess & Check* methodology. A *Guess & Check* program consists of two parts, namely a *guessing* part and a *checking* part. The first being responsible for defining the search space, i.e. generating solution candidates, whereas the latter ensures that all criteria are met and filters out inadmissible candidates. You should create now three files—vending_balance.dl, vending_gravity.dl and vending_guess.dl—each consisting of the corresponding part of the ASP encoding, where the latter two files constitute the checking part.

> **Important**
>
> Creating and submitting test cases is mandatory, i.e. if you do not submit your test files you will get no points.

2.2 Writing the guessing program (1 point)

Let us have a look at the example in Figure 1, the *guessing* part of your ASP encoding should generate $5^3 = 125$ *potential solutions* (taking only those cells into account which are part of a tray):

1. `{}`

2. `{snack(1,1,1)}`

3. `{snack(1,1,2)}`

4. `{snack(1,1,3)}`

5. $\cdots$

6. `{snack(1,1,1), snack(1,2,1), snack(2,1,1)}`

7. `{snack(1,1,2), snack(1,2,1), snack(2,1,1)}`

8. `{snack(1,1,3), snack(1,2,1), snack(2,1,1)}`

9. $\cdots$

After you created the guessing part of your encoding you can run it along with your test cases using `DLV`. E.g., if we store the problem instance shown in Figure 1 to a file called `vending_test1.dl`, we may call

```
$ dlv vending_test1.dl vending_guess.dl
```

to generate all possible solution candidates.

> **Hint**
>
> Note that we can instruct `DLV` to display only the predicates `p1`, `p2`, ... by passing `-pfilter=p1,p2,...` on the command line. E.g., when you are only interested in the extension of the `snack` predicate you might use `-pfilter=snack` and receive the output above.

> **Hint**
>
> During the construction of the program, you can limit the number of generated answer sets by passing the command line argument `-n=K` to compute only the first `K` answer sets.

2.3 Writing the checking program (5 points)

You might have noticed that not all of the potential solutions shown above are actual solutions, i.e., not all of them match the required sums of weights per row or column. For instance, the first candidate cannot be a solution since the number of snacks in each row resp. column is 0. On the other hand, the 5th candidate is indeed a solution and it corresponds to the solution shown in the subfigure on the bottom of Figure 1. Thus, the computation of inadmissible solutions has to be avoided, this is what the *checking* part is for.

2.3.1 Part 1: Balance (3.5 points)

The first part of the checking program should ensure that all rows and columns have the required sum of weights among its cells. Therefore, the file named `vending_balance.dl` should contain *constraints* which ensure that the solutions meet the given criteria. This is usually done by adding integrity constraints to your encoding, each describing non-solutions of the given problem.

Once you have completed the checking part of your encoding, run your test cases you wrote before along with both the guessing and the checking part of your ASP encoding and compare carefully whether the expected fillings will be computed. This can be done with

```
$ dlv vending_test1.dl vending_guess.dl vending_balance.dl
```

which should produce the appropriate answer set(s). Considering the example depicted in Figure 1, there should be just one answer set:

```
{ tray(1,1,1), tray(1,1,2), tray(2,2,1),
  row(1,2), row(2,1),
  column(1,2), column(2,1),
  right(1,2,1), below(1,2),
  snack(1,1,1),  snack(2,1,1),  snack(1,2,1) }
```

> **Hint**
>
> Again, the use of the `-pfilter=p1,p2,...` and `-n=K` options help to restrict the output to a managable size.

2.3.2 Part 2: Gravity (1.5 points)

The second part of the check program should enforce the *gravity* constraint. To this end, create a file named `vending_gravity.dl` containing the required integrity constraints. Once you have completed this part, run your test cases along with the *guessing*, the *balance* and the *gravity* part.

```
$ dlv vending_test1.dl vending_guess.dl vending_balance.dl
      vending_gravity.dl
```

Considering the example shown in Figure 2, your encoding (without using `vending_gravity.dl`) should find several balanced solutions, but only

```
{snack(1,2,1),                snack(3,2,1),
 snack(1,3,1), snack(2,3,1), snack(3,3,1),
               snack(2,4,1)}
```

is satisfactory in terms of *gravity*. Thus, your encoding (also taking into account `vending_gravity.dl`) should only compute this solution.

3 Optimization (2 points)

Consider now that you want to maximize the number of snacks stored in your trays, i.e. since the solution still requires to satisfy all the conditions before, and in particular has to be *balanced*, you are actually preferring lots of light-weighted snacks before a small number of heavy-weighted snacks. You can achieve the goal by using weak constraints as discussed in the lecture. So, in total you are about to follow the *Guess, Check & Optimize* methodology consisting of three parts, namely *guessing*, *checking* and *optimization*.

Write the actual additional rule(s) required to model this preference and save it as `vending_optimize.dl`. Once completed, run it with the test cases you wrote earlier together with programs `vending_balance.dl`, `vending_guess.dl` and `vending_gravity.dl`. Compare it with your results before and check whether the expected (optimized) fillings are generated.

4 Submission

Finally, pack the following files in a ZIP archive called `XXXXXXXX_vending.zip`, where `XXXXXXXX` is replaced by your matriculation number. Make sure that your ZIP file does not contain any directories.

1. `vending_test`n`.dl`, where n is a number ($1 \leq n \leq 5$)
2. `vending_guess.dl`
3. `vending_balance.dl`
4. `vending_gravity.dl`
5. `vending_optimize.dl`