Knowledge-Based Systems for Business Informatics, Project 2: Cargo Loading Problem

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

Intuitively, the *Cargo Loading Problem* is related to generating cargo plans for big container ships, which ensures that ships remain balanced even when a ship is in rough sea, and furthermore evenly loaded along their multi- leg travel routes, where the ship will unload a part of their payload at intermediary ports.

In a simplified setting, the cargo area can be represented as a cuboid consisting of cells. How the payload should be distributed is given by a number of loaded cells for a specific set of cells. More specifically, such requirements are given for consecutive cells along the different axis of the cuboid.

In order to make the specification more precise, we need the notion of ω-*sets* for $\omega \in \{x, y, z\}$.

Definition 1. *Let* $C = \{(x_1, y_1, z_1), \ldots, (x_m, y_n, z_o)\}$ *be a set of coordinates representing an* $m \times n \times o$ *rectangular cuboid. Then, an* x*-set of* y_0 *and* z_0 *is*

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

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

the set of coordinates in C with y-coordinate y_0 and z-coordinate z_0; $y\text{-}sets$ and $z\text{-}sets$ can be defined analogously. Formally, we get the following:

$$x\text{-}set_{y_0,z_0} = \{(x_i, y_0, z_0) \mid 1 \leq i \leq m\} \tag{1}$$
$$y\text{-}set_{x_0,z_0} = \{(x_0, y_i, z_0) \mid 1 \leq i \leq n\} \tag{2}$$
$$z\text{-}set_{x_0,y_0} = \{(x_0, y_0, z_i) \mid 1 \leq i \leq o\} \tag{3}$$

Intuitively, each coordinate in C represents a cell in the cuboid, and an $\omega\text{-}set$ is a set of consecutive cells along the ω-axis. E.g., for the cuboid depicted in Figure 1 we have two $x\text{-}sets$ (corresponding to the rows), two $y\text{-}sets$ (corresponding to the columns), and four $z\text{-}sets$ each consisting of a single cell. Considering the more involved cube in Figure 2, we have four $x\text{-}sets$, $y\text{-}sets$ and $z\text{-}sets$, each set consisting of 2 cells.

Now, the *Cargo Loading Problem* is defined as follows. You are given a positioning description of several containers placed on a ship, and for each $\omega\text{-}set$ in the grid there is an associated number—$x\text{-}sum_{y_0,z_0}$, $y\text{-}sum_{x_0,z_0}$ or $z\text{-}sum_{x_0,y_0}$—between 0 and the number of cells in the corresponding $\omega\text{-}set$. The containers are of different forms and sizes, that is, each is made up of one or more cells on a grid representing the ship. Each container can be (partially) loaded with goods; specifically, each cell either contains goods or is empty. Furthermore, containers will never overlap, but they can have gaps, i.e., a container does not necessarily have to be a set of neighbouring cells.

Now a solution to this problem consists of all those cells on the grid that are loaded with goods, such that the payload is

(1) *balanced*, i.e., the sum of loaded cells in each $x\text{-}set_{y_0,z_0}$, $y\text{-}set_{x_0,z_0}$, and $z\text{-}set_{x_0,y_0}$ matches the given number $x\text{-}sum_{y_0,z_0}$, $y\text{-}sum_{x_0,z_0}$, and $z\text{-}sum_{x_0,y_0}$ of cells, resp., and

(2) satisfies the rules of *gravity*, i.e., for two consecutive cells on the y-axis that are part of the same container, it is not allowed that the successor is loaded and the predecessor is not. That is, it is not allowed that a cell is loaded if the cell *below* is not, given that both cells belong to the same container.

See Figure 1 for a simple example with two containers, where cells of the same colour form a container. The subfigures on the top show the required number of loaded cells per $\omega\text{-}set$. The subfigure on the bottom shows the correct solution where the hatching depicts the loaded cell, i.e., the green-colored cell is loaded. Figure 2 shows a cube which has two *balanced* solutions, but only one of them follows the rules of *gravity*. The subfigures on the bottom depict the two balanced solution, whereas only the left one is valid in terms of gravity.

Please note that there are situations, where it is not possible to find a cargo plan which meets the given requirements. For instance, consider a $1 \times 1 \times 1$ cube and $x\text{-}sum_{y_1,z_1} = y\text{-}sum_{x_1,z_1} = z\text{-}sum_{x_1,y_1} = 2$. Clearly, it is impossible to find a solution in such a case.

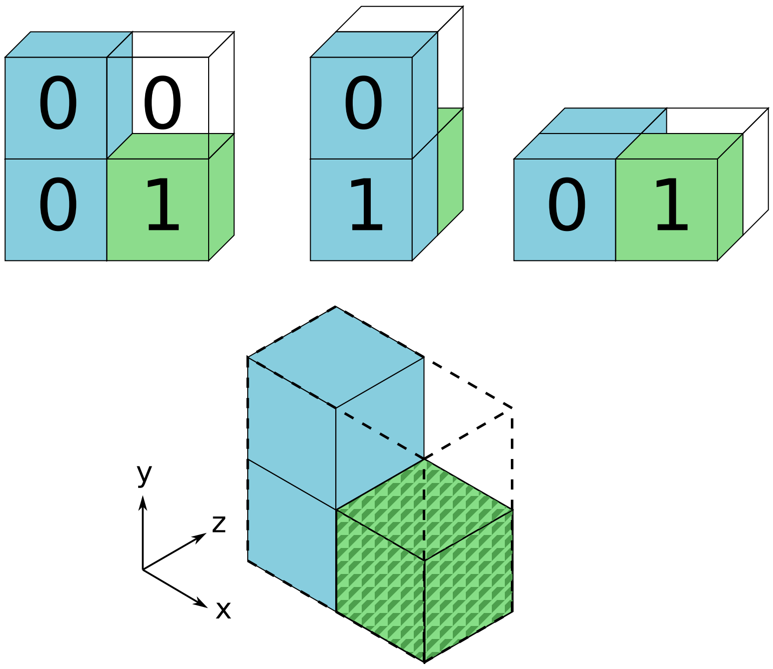

Figure 1: A valid example with two containers.

Important

Note that the grid always has its origin (x_0, y_0, z_0) in the foremost lower-left corner of the cuboid. This means that cells located at (x, y, z) such that $x > x_0$ are to the right of the origin, cells at (x, y, z) with $y > y_0$ are above the origin, and cells at (x, y, z) with $z > z_0$ are behind the origin. Consider this when modelling the constraint that gravity exerts.

2 Problem encoding

Your assignment is to write a logic program for `DLV` that takes the description of containers and the number of loaded cells per ω-*set*. The answer sets of your program should describe all possible loadings of the containers such that the constraints for each ω-*set* are satisfied.

Use the following predicates for the input specification:

`left(X1,X2)` represents that `X1` is followed by `X2` on the x-axis.
Ex.: `left(x1,x2).`

`below(Y1,Y2)` represents that `Y1` is followed by `Y2` on the y-axis.
Ex.: `below(y1,y2).`

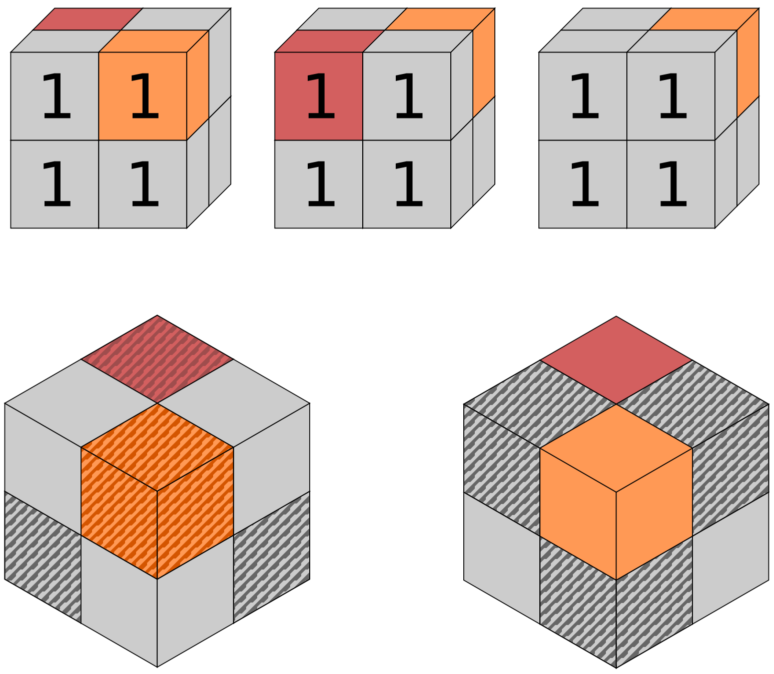

Figure 2: A grid consisting of 3 containers.

`before(Z1,Z2)` represents that `Z1` is followed by `Z2` on the z-axis.
Ex.: `before(z1,z2).`

`xsum(Y,Z,S)` expresses that the set of cells—matching the y-coordinate `Y` and the z-coordinate `Z`—exhibit `S` loaded cells.
Ex.: `xsum(y1,z1,5).`

`ysum(X,Z,S)` expresses that the set of cells—matching the x-coordinate `X` and the z-coordinate `Z`—exhibit `S` loaded cells.
Ex.: `ysum(x1,z1,7).`

`zsum(X,Y,S)` expresses that the set of cells—matching the x-coordinate `X` and the y-coordinate `Y`—exhibit `S` loaded cells.
Ex.: `zsum(x1,y1,9).`

`container(C,X,Y,Z)` designates that the cell at `(X,Y,Z)` is part of the container labelled `C` (i.e., all cells with the same `C` together form a single container).
Ex.: `container(con1,x1,y1,z1).`

Use the ternary predicate `loaded` as output of your encoding; `loaded(X,Y,Z)` then indicates that the cell located at (x, y, z) is loaded with goods.

3 Creating 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 `cargo_test`n`.dl`, where n is a number ($1 \leq n \leq 5$).

As an example, consider the respective test case for the cuboid depicted in Figure 1:

```
left(x1,x2).
below(y1,y2).

container(c1,x1,y1,z1).
container(c1,x1,y2,z1).
container(c2,x2,y1,z1).

xsum(y1,z1,1). xsum(y2,z1,0).
ysum(x1,z1,0). ysum(x2,z1,1).
zsum(x1,y1,0). zsum(x1,y2,0).
zsum(x2,y1,1). zsum(x2,y2,0).
```

Now you are ready to write the actual answer-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. To this end, you should create the three files `cargo_guess.dl`, `cargo_check_balance.dl`, and `cargo_check_gravity.dl`, each consisting of the corresponding part of the ASP encoding.

Important

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

4 Writing the guessing program (1 pts.)

Let us have a look at the example in Figure 1, the *guessing* part of your ASP encoding should generate $2^3 = 8$ *potential solutions* (taking only those cells into account which are part of a container):

1. `{}`
2. `{loaded(x1,y1,z1)}`
3. `{loaded(x1,y2,z1)}`
4. `{loaded(x2,y1,z1)}`

5. `{loaded(x1,y1,z1), loaded(x1,y2,z1)}`
6. `{loaded(x1,y1,z1), loaded(x2,y1,z1)}`
7. `{loaded(x1,y2,z1), loaded(x2,y1,z1)}`
8. `{loaded(x1,y1,z1), loaded(x1,y2,z1), loaded(x2,y1,z1)}`

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 `cargo_test1.dl`, we may call

```
$ dlv cargo_test1.dl cargo_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 `loaded` predicate you might use `-pfilter=loaded` 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.

5 Writing the checking program (7 pts.)

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 number of goods per ω-set. For instance, the first candidate cannot be a solution since the number of goods in each ω-set is 0. On the other hand, the 4th 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.

5.1 Part 1: Balance (4 pts.)

The first part of the checking program should ensure that all ω-*sets* have the required number of loaded cells. Therefore, the file `cargo_check_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 loadings will be computed. This can be done with

```
$ dlv cargo_test1.dl cargo_guess.dl cargo_check_balance.dl
```

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

```
{ loaded(x2,y1,z1) }
```

Hint

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

5.2 Part 2: Gravity (3 pts.)

The second part of the check program should enforce the *gravity* constraint. Therefore, create a file `cargo_check_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 cargo_test1.dl cargo_guess.dl cargo_check_balance.dl
      cargo_check_gravity.dl
```

Considering the example shown in Figure 2, your encoding (without the file `cargo_check_gravity.dl`) should find 2 solutions which are *balanced*, namely

```
{loaded(x1,y1,z1), loaded(x2,y1,z2), loaded(x1,y2,z2),
   loaded(x2,y2,z1)}
```

and

```
{loaded(x1,y1,z2), loaded(x1,y2,z1), loaded(x2,y1,z1),
   loaded(x2,y2,z2)}
```

but only the first one is satisfactory in terms of *gravity*. Thus, your encoding (with `cargo_check_gravity.dl`) should only compute the first answer set.