Knowledge Representation and Reasoning 2023

Homework assignment #2

Didier Merk (11037172), Tsatsral Mendsuren (14530775)

Exercise 1. - *Answer Set Equivalence*

- We start by giving two normal logic programs P_1 and P_2 that are weakly equivalent, meaning they have the exact same answer sets. In the second bullet point we will demonstrate why these two logic programs are weakly equivalent and in the third bullet point we will demonstrate that they are not strongly equivalent.

  Logic Program P_1:

  ```
  a :- b.
  a.
  ```

  Logic Program P_2:

  ```
  c :- b.
  a.
  ```

  Answer sets P_1:

  ```
  a
  ```

  Answer sets P_2:

  ```
  a
  ```

- When we follow the rules of answer set programming, we see that for logic program P_1 we have the fact `a.`, meaning that it must be in our answer set. There are no other rules in the logic program that make any other atom true, meaning that with the closed-world assumption, `a` is an answer set and also the only answer set for P_1.

  Similarly, for P_2 we have the fact `a.` and no other rule that enforces any other atom to be true. This means that just like P_1 the one and only answer set for P_2 is `a`. Since, both P_1 and P_2 have the exact same answer sets, they are *weakly equivalent*.

- To demonstrate that P_1 and P_2 are **not** *strongly equivalent*, we propose a third normal logic program P_3 such that $P_1 \cup P_3$ and $P_2 \cup P_3$ do not have the same answer sets.

  Logic Program P_3:

  ```
  b.
  ```

  Logic Program $P_1 \cup P_3$:

  ```
  a :- b.
  a.
  b.
  ```

  Logic Program $P_2 \cup P_3$:

  ```
  c :- b.
  a.
  b.
  ```

  Answer sets $P_1 \cup P_3$:

  ```
  a b
  ```

  Answer sets $P_2 \cup P_3$:

  ```
  a b c
  ```

  When we look at logic program $P_1 \cup P_3$, we see that `a.` and `b.` are facts, meaning they are present in our answer set. We have a third rule `a :- b.`, which states that when `b` is true that `a` must be true (and therefore present in our answer set). However, we already had the fact that `a` must be in our answer set. This means that, since there is no other rule that makes any other atom true, our answer set is `a b`.

  On the other hand, `a b` is **NOT** an answer set for $P_2 \cup P_3$, since we have a rule that states `c :- b.`. This rule indicates that when `b` is true (and therefore present in the answer set), `c` MUST also be in the answer set.

Exercise 2. - *Two-dimensional packing problem*

In this exercise we will look at a two-dimensional packing problem, in which we try to place multiple rectangles inside a container without overlap. We will describe how we can translate this problem to an answer set program P, such that the answer sets of P correspond to the different ways of placing the rectangles inside the container. We will do this using four steps (slide 38, lecture 5):

> **Step 1**: Formalizing the problem: what is the problem input, what are solutions and what properties does a solution require?
>
> **Step 2**: Establishing encoding of problem instances: how to represent the input, what predicates do we use and which constants? How do we translate this to an answer set program?
>
> **Step 3**: Establishing the encoding of candidate solutions: here we *generate* candidate solutions, we decide what rules need to be added and how to obtain an answer set from this solution.
>
> **Step 4**: Establishing the encoding of solution properties: here we *test* candidate solutions; we decide which rules and constraints need to be added so that the only answer sets remaining correspond with actual solutions of our problem.

In the exercise an example of this problem is given, which we will use to illustrate the solution, however the solution will be of a form that is general and will work for any input of the problem that is of the right format.

(I) As explained, first we will formalize the problem. This is mainly already done by the assignment but we will give a quick summary here. The input of the problem consists of two integers h, $w \in \mathbb{N}$, that represent the height and the width of a rectangular container. In addition the input will be a list of pairs of positive integers (h_1, w_1), ..., (h_k, w_k). This list represents the k rectangular items that need to be placed inside this container, with (h_i, w_i) representing the height and the width of the i-th item respectively.

The task is to find out how to put all these k rectangular items in the container, without rotating them. Each item needs to be placed inside of the container exactly once and they may not exceed the boundaries of the container. The items also may not overlap each other, however there may be gaps left in the container. The items also are placed at integer coordinates, meaning they can only move by integer amounts.

In the example we have a rectangular container has dimensions $h = 4$ and $w = 6$. There are four rectangular items that need to be placed in this container with (h_1, w_1) = (1, 3), (h_2, w_2) = (2, 2), (h_3, w_3) = (2, 3) and (h_4, w_4) = (3, 3).

(II) The second step is to look how we can encode this input to be able to use it for an answer set program P. Starting with the dimensions of the container, which we will generally call H and W we can use the `#const` feature as follows, which allows a user to input any dimensions for a container. It will look like this:

```
1 % size of the container
2 #const w = W.
3 #const h = H.
```

Then, we need to a way to input the dimensions of the rectangular items that need to be placed inside the container. To do this we use the tertiary predicate symbol `rect(I,HI,WI)`, where `HI` and `WI` represent the height and the width of the `I`'th item. Any combination of blocks can be inputted as follows:

```
4 % all the rectangles to put in the container
5 rect(1, H1, W1; ...; K, HK, WK).
```

The user is now able to input the dimensions of the container and the rectangle items, which is all the input of the problem.

(III) We are now going to generate possible solutions, for this we will need to add multiple rules and constraints to our program. The first thing we do is, because the items may only move by integer amounts, generate a grid of cells in the container, which corresponds to possible places an item can be placed on.

To generate this grid we use the binary predicate symbol `cell(X,Y)`, which used the variables X and Y to generate a coordinate system with the size of the container. The code `X=0..w-1` and `Y=0..h-1` can be seen as X and Y taking all values between 0 and the width/height. We use `w-1` and `h-1` in order to have to have the top left corner have coordinate be (0,0). In an answer set program it will look like this:

```
6 % generate the grid (to make sure the movement is only at integers)
7 cell(X,Y) :- X=0..w-1, Y=0..h-1.
```

Next is an important step. Every rectangle needs to be placed exactly once and fully within the boundaries of the container. To do this we create a rule demonstrated below in line 9.

In this rule, we look at all our inputted rectangles `rect(I,WI,HI)` and check if their placement does not exceed the boundary: `cell(X,Y), X+WI<=w, Y+HI<=h`. If this is indeed the case, we use the binary predicate symbol `place(R,cell(X,Y))` to denote that the top left corner of a rectangle item has been placed inside on the top left corner of this cell.

In the code, the `1`'s on either side of the curly brackets, called the one-on-one mapping, make sure that every rectangle is associated with being placed on exactly one position on the grid. The place predicate will therefore encode all the placement coordinates of the rectangles and the index of a rectangle corresponding to that placement.

```
8 % place each rectangle in the container exactly once by one-on-one mapping
9 1{place(R,cell(X,Y)): cell(X,Y), X+WI<=w, Y+HI<=h}1 :- rect(R,WI,HI).
```

When a rectangle is placed on a cell, it is not just covering only that cell. It is also covering the surrounding cells, the amount of which is depending on the size of the rectangle. To account for this we use the tertiary predicate symbol `covered(R,X,Y)` which encodes all the cells covered by a rectangle. We do this using the following rule, shown below in line 11.

In this rule, we look at all the different rectangles, given by `rect(R,WI,HI)`; their corresponding placements, which are given by `placement(R,pos(X,Y))`; and their corresponding sizes, which are given by `I=0..WI-1` and `J=0..HI-1`. The covered predicate will then encode all the cells that are covered by a rectangle and include the index of the rectangle covering that cell.

```
10 % describe which cells are covered after the placement of rectangles
11 covered(R, X+I, Y+J) :- rect(R,WI,HI), place(R,cell(X,Y)), I=0..WI-1, J=0..
       HI-1.
```

(IV) So far we have implemented the rules that will place every rectangle in the container, including the constraints that a rectangle will be placed fully within the boundary of the container and exactly once. This will already generate answer sets for us corresponding to placements of rectangles in the container.

The final step that we need to take is to add rules or constraints to our program in order to make sure that only answer sets remain that correspond to actual solutions. When we run the program using the input, rules and constraints we have so far, 23040 possible solutions are generated.

We notice that we have not added an important constraint that enforces our answer sets to correspond to actual solutions; the rectangles may not overlap. This means that we need to remove all the answer sets in which we have rectangles that are overlapping. In other words this can be described as, we need to remove all the cells that are covered by more than one rectangle. We enforce this by adding the constraint below to our answer set program. In this constraint we check for all cells within the container, `cell(X,Y)`, whether they are covered by at most one rectangle, `2{covered(R,X,Y)}`.

```
12 % no cell can be occupied more than once (no overlapping)
13 :- 2{covered(R,X,Y)}, cell(X,Y).
```

Finally, we want the output to just be the cells in which we have to place our rectangles. For this we use the `#show` feature:

```
14 % output only the placement cells
15 #show place/2.
```

This is the full answer set program that we propose. Due to all the implemented rules and constraints, all the valid placements of items in a container will be outputted by this program as an answer set. This is because there are no valid placements that will be 'cut' from the solutions.

To demonstrate how this program can be used to solve the problem we will give a short visualisation, entering the input from the example in our answer set program and running it on https://potassco.org/clingo/run/:

Answer Set Program P:

```
1 % Input (step 2)
2
3 % size of the container
4 #const w = 6.
5 #const h = 4.
6
7 % all the rectangles to put in the container
8 rect(1,3,1; 2,2,2; 3,3,2; 4,3,3).
```

```
9
10 % Generating candidate solutions (step 3)
11
12 % generate the grid (to make sure the movement is only at integers)
13 cell(X,Y) :- X=0..w-1, Y=0..h-1.
14
15 % place each rectangle in the container exactly once by one-on-one mapping
16 1{place(R,cell(X,Y)): cell(X,Y), X+WI<=w, Y+HI<=h}1 :- rect(R,WI,HI).
17
18 % describe which cells are covered after the placement of rectangles
19 covered(R, X+I, Y+J) :- rect(R,WI,HI), place(R,cell(X,Y)), I=0..WI-1, J=0..HI
      -1.
20
21 % Constraints (step 4)
22
23 % no cell can be occupied more than once (no overlapping)
24 :- 2{covered(R,X,Y)}, cell(X,Y).
25
26 % output only the placement cells
27 #show place/2.
```

We will not display the full results of running the program here, it can be found at the bottom of this document. The program returns 20 answer sets, corresponding to 20 different configurations of rectangle placements. One of the answer sets for example is:

```
Answer: 20
place(1,cell(3,0)) place(2,cell(0,2)) place(3,cell(0,0)) place(4,cell(3,1))
```

We can use this, to see that for answer set 20 the placement of rectangles is as follows: rectangle 1 is placed on cell (3,0), which means that its top left corner is placed on the top left corner of the left-most cell of the 4th row; rectangle 2 is placed on cell (0, 2); rectangle 3 is placed on cell (0,0); rectangle 4 is placed on cell (3,1). We can similarly use all returned answer sets to find all the possible rectangle placements.

Exercise 3. - *Logic Programs and Propositional Logic Formulas*

(a) The method proposed in the question for computing answer sets for normal logic programs does **not** always return an answer set of logical program P if P has answer sets. This is mainly due to the 'closed-world assumption' in logical programming which states that *"statements for which we have no reason to conclude that they are true, are false"*. This can be demonstrated by the following example, where we have a normal logic program P:

Logic Program P:

```
a :- not b.
b :- not a.
```

Answer sets:

```
a
b
```

Using the rules given in the assignment, this method will translate this normal logic program P to a CNF formula φ_p by rewriting each rule, fact or constraint as a clause of the CNF formula.

For the first rule `a :- not b.`, the clause $(p_b \vee p_a)$ is added to φ_p; for the second rule `b :- not a.`, the clause $(p_a \vee p_b)$ is added to φ_p. These two clauses are logically equivalent, meaning we end up with:

$$\varphi_p = \quad (p_a \vee p_b)$$

We can immediately see that possible solutions here are that either p_a or p_b has to be true, or that both can be true. According to the method in the exercise, when an SAT solver is called on φ_p, it returns three satisfying assignments α and the corresponding interpretations $\mathcal{M}_\alpha$:

Satisfying assignment α	Interpretation $\mathcal{M}_\alpha$
$p_a = 1, p_b = 0$	`a`
$p_a = 0, p_b = 1$	`b`
$p_a = 1, p_b = 1$	`a b`

In this table we can see the method returns three interpretations $\mathcal{M}_\alpha$, of which the third (`a b`) is **not** an answer set of P. Therefore we can conclude that a satisfying assignment α does **NOT** always return an answer set of P if P has answer sets.

(b) Similar to question a, we demonstrate through an example that the method does **NOT** always return `none` if P has no answer sets. The main reasons for this, again, is the 'closed-world assumption'. Take for example the following normal logic program P that has no answer sets:

Logic Program *P*:

```
a.
c :- a, not b.
d :- c.
b :- d.
```

Answer sets:

```
none
```

Translating P using the rules as before results in the following CNF formula φ_p:

$$\begin{aligned} \varphi_p = \quad & (p_a) \wedge && \text{(clause } c_1\text{)} \\ & (p_c \vee \neg p_a \vee p_b) \wedge && \text{(clause } c_2\text{)} \\ & (p_d \vee \neg p_c) \wedge && \text{(clause } c_3\text{)} \\ & (p_b \vee \neg p_d) && \text{(clause } c_4\text{)} \end{aligned}$$

Which can for example be satisfied by setting $p_a = 1, p_b = 1, p_c = 0, p_d = 0$ demonstrated below:

$$\begin{aligned} \varphi_p = \quad & (p_a) \wedge && \text{(clause } c_1\text{)} \checkmark \\ & (p_c \vee \neg p_a \vee p_b) \wedge && \text{(clause } c_2\text{)} \checkmark \\ & (p_d \vee \neg p_c) \wedge && \text{(clause } c_3\text{)} \checkmark \\ & (p_b \vee \neg p_d) && \text{(clause } c_4\text{)} \checkmark \end{aligned}$$

An overview of all satisfying assignments α and the corresponding interpretations $\mathcal{M}_\alpha$ are given in the table below.

Satisfying assignment α	Interpretation $\mathcal{M}_\alpha$
$p_a = 1, p_b = 1, p_c = 0, p_d = 0$	`a b`
$p_a = 1, p_b = 1, p_c = 1, p_d = 0$	`a b c`
$p_a = 1, p_b = 1, p_c = 0, p_d = 1$	`a b d`
$p_a = 1, p_b = 1, p_c = 1, p_d = 1$	`a b c d`

It can be clearly seen that while P has no answer sets, this method does **not** always return `none`.

(c) We argue that because of the two limitations of the method shown in question a and b, that the method in its current form is **not** a good approach for finding answer sets. In question a we demonstrated that the method returns answer sets that are not answer sets of logical program P; in question b we demonstrated that when a logical program P has no answer sets, this method can still return an interpretation $\mathcal{M}_\alpha$ that is not `none`.

However, with some adjustments, or constraints to the method, it could still be an acceptable approach to finding answer sets. The main issue you want to tackle is to somehow incorporate the 'closed-world assumption' also in the CNF method.

To reiterate, the closed-world assumption basically means that we can never assume an atom to be true, when there was no reason to set it to true. If we want to translate this to the solutions of the CNF formula, we only want to include the solutions from the method where all predicates that are set to true, were *forced* to be true. If you would write code that removes the interpretations from the output for which this is not the case, the remaining interpretations are the answer sets of P.

In the example of 3a this would remove `a b` from the output, since both literal p_a and p_b were 'chosen freely' to be true. In the example of 3b this would remove all four interpretations (since they have all 'freely chosen' to set p_b to true), and the model would correctly output `none`.

We think that the method as it is currently proposed is not a good approach for finding answer sets, however we believe that with extra adjustments it could still be used.

Full output of CLINGO program:

```
clingo version 5.6.0 (c0a2cf99)
Reading from stdin
Solving...
Answer: 1
place(1,cell(0,3)) place(2,cell(3,0)) place(3,cell(3,2)) place(4,cell(0,0))
Answer: 2
place(1,cell(0,3)) place(2,cell(4,0)) place(3,cell(3,2)) place(4,cell(0,0))
Answer: 3
place(1,cell(0,0)) place(2,cell(3,0)) place(3,cell(3,2)) place(4,cell(0,1))
Answer: 4
place(1,cell(1,0)) place(2,cell(4,0)) place(3,cell(3,2)) place(4,cell(0,1))
Answer: 5
place(1,cell(0,0)) place(2,cell(4,0)) place(3,cell(3,2)) place(4,cell(0,1))
Answer: 6
place(1,cell(2,0)) place(2,cell(0,0)) place(3,cell(0,2)) place(4,cell(3,1))
Answer: 7
place(1,cell(3,0)) place(2,cell(0,0)) place(3,cell(0,2)) place(4,cell(3,1))
Answer: 8
place(1,cell(3,0)) place(2,cell(1,0)) place(3,cell(0,2)) place(4,cell(3,1))
Answer: 9
place(1,cell(3,3)) place(2,cell(0,0)) place(3,cell(0,2)) place(4,cell(3,0))
Answer: 10
place(1,cell(3,3)) place(2,cell(1,0)) place(3,cell(0,2)) place(4,cell(3,0))
Answer: 11
place(1,cell(0,3)) place(2,cell(4,2)) place(3,cell(3,0)) place(4,cell(0,0))
Answer: 12
place(1,cell(1,3)) place(2,cell(4,2)) place(3,cell(3,0)) place(4,cell(0,0))
Answer: 13
place(1,cell(0,3)) place(2,cell(3,2)) place(3,cell(3,0)) place(4,cell(0,0))
Answer: 14
place(1,cell(0,0)) place(2,cell(4,2)) place(3,cell(3,0)) place(4,cell(0,1))
Answer: 15
place(1,cell(0,0)) place(2,cell(3,2)) place(3,cell(3,0)) place(4,cell(0,1))
Answer: 16
place(1,cell(3,3)) place(2,cell(1,2)) place(3,cell(0,0)) place(4,cell(3,0))
Answer: 17
place(1,cell(3,3)) place(2,cell(0,2)) place(3,cell(0,0)) place(4,cell(3,0))
Answer: 18
place(1,cell(2,3)) place(2,cell(0,2)) place(3,cell(0,0)) place(4,cell(3,0))
Answer: 19
place(1,cell(3,0)) place(2,cell(1,2)) place(3,cell(0,0)) place(4,cell(3,1))
Answer: 20
place(1,cell(3,0)) place(2,cell(0,2)) place(3,cell(0,0)) place(4,cell(3,1))
SATISFIABLE

Models       : 20
Calls        : 1
Time         : 0.036s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```