VU Einführung in Wissensbasierte Systeme
ASP Project: ASP Modelling

WS 2022/23

You have two turn-in possibilities for this project. The procedure is as follows: your solutions will be tested with automatic test cases after the first turn-in deadline and tentative points will be made available in TUWEL. You may turn-in (repeatedly) until the second turn-in deadline two weeks later, where the total points for your project are calculated as follows (p_i are the achieved points for the i-th turn-in, $i \in \{1, 2\}$):

- if you deliver your project at both turn-in 1 and 2, you get the maximum of the points and the weighted mean:
$$\max\left\{p_1, \quad 0.8 \cdot p_2, \quad \frac{p_1 + 0.8 \cdot p_2}{1.8}\right\};$$
- if you only deliver your project at turn-in 1: p_1;
- if you only deliver your project at turn-in 2: $0.8 \cdot p_2$; and
- 0, otherwise.

The deadline for the first turn-in is **Wednesday, December 07, 23:55**. You can submit multiple times, please submit early and do not wait until the last moment. The deadline for the second turn-in is **Wednesday, December 21, 23:55**.

This project deals with encoding problems in terms of ASP Core-2 programs and using Clingo as solving tool (see `https://potassco.org/clingo` to download the system). The project is divided into three separate exercises: the first exercise gives up to 7 points, whereas the second and third give both up to 5 points. Thus, the maximum score amounts to **17 points**. In order **to pass** this project, you are required to attain at least **9 points**. There are no minimum points required on a specific exercise. **(Please note that solving only one exercise, will NOT be sufficient for a positive grade!)** Detailed information on how to submit your project is given in Section 5. Make sure that you have named all files and all predicates in your encodings according to the specification. Failing to comply with the naming requirements will probably result in losing all points of the corresponding exercise.

For general questions on the assignment, please consult the TUWEL forum. Questions that reveal (part of) your solution should be discussed privately during the tutor sessions (see the time slots listed in TUWEL) or send an email to:

ewbs-2022w@kr.tuwien.ac.at

1 Distance-Preserving Graph Homomorphism (7 points)

1.1 Problem Specification

A *graph homomorphism* between two (directed) graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is a function f that maps every vertex of V_G to a vertex in V_H such that if there is an edge $(a, b) \in E_G$ then $(f(a), f(b)) \in E_H$. Your task is to find a homomorphism f between two graphs. As a side-condition, a set $E_P \subseteq V_G \times V_G$ of distance-preserving vertex pairs is given. Let $dist_K(p_1, p_2)$ be the minimal distance between vertex p_1 and p_2 in a graph K.

In case $(p_1, p_2) \in E_P$, the function f has to be such that $dist_G(p_1, p_2) = dist_H(f(p_1), f(p_2))$. For all vertex pairs not in E_P, the minimum distance may change arbitrarily.

Your task is to give an ASP Core-2 program that finds all possible homomorphisms f modulo the distance-preserving side-condition.

1.2 Problem Encoding

Use the following predicates for the input specification:

- `edge1(V1,V2)`: `(V1,V2)` is a directed edge in the graph G;
- `edge2(W1,W2)`: `(W1,W2)` is a directed edge in the graph H;
- `numNodes1(N)`: the number of nodes in graph G;
- `numNodes2(N)`: the number of nodes in graph H;
- `preserve(V1,V2)`: the minimum distance between the vertices `V1` and `V2` of graph G should be preserved in the homomorphism.

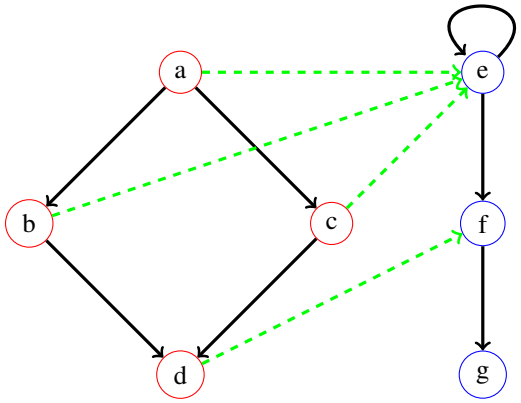
Important

- The vertices of the graphs are not explicitly given. However, you can derive them from the edges.
- You may assume that the elements in `edge1` and `edge2` are disjoint.
- Maybe you will not need `numNodes1` or `numNodes2`. However, you can use them to simplify your program.
- As the graph is directed, `preserve(a,b)` does not necessarily imply `preserve(b,a)`.

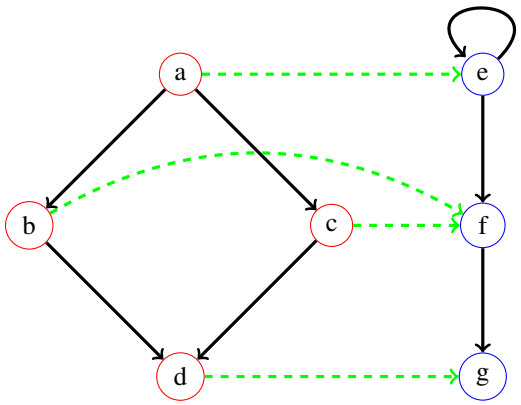
The output of your program should be given by the binary predicate `maps(V,W)`, where `V` is a vertex of G and `W` is a vertex of H. Each element of G may be mapped only to a single element of H (but not necessarily vice versa).

1.2.1 Example

Consider the following two graphs G ($\{a,b,c,d\}$ – red) and H ($\{e,f,g\}$ – blue) and a homomorphism (green):



If we add the constraint `preserve(a,d)`, the result is not a solution anymore, as the minimum distance between a and d decreased from 2 to 1. The *only* solution with this additional constraint is the following, as it preserves the minimum distance of 2 between (a,d):



The input for this problem is:

`edge1(a,b). edge1(a,c). edge1(b,d). edge1(c,d).`

`edge2(e,f). edge2(f,g). edge2(e,e).`

`numNodes1(4). numNodes2(3). preserve(a,d).`

The homomorphism depicted above looks as follows:

`maps(a,e). maps(b,f). maps(c,f). maps(d,g).`

2 Generalised Dominating Bishop Problem (5 points)

2.1 Problem Specification

We consider the following problem:

> Given an $N \times N$ chess board, a number of indistinguishable chess bishops, and a function that maps each chess board tile to a natural number, find a placement of the bishops on the board such that each tile is *at least* as often attacked as specified by the function.

Here, a bishop can attack other tiles in all four diagonal directions but it cannot attack horizontally or vertically. Furthermore, they cannot attack tiles in case there are other bishops in between.

2.2 Problem Encoding

Construct an ASP Core-2 encoding such that the answer sets of your program describe all positionings of bishops that satisfy the given constraints.

Your program can assume the existence of three predicates:

- `size(N)`, where `N` is the size of the `N` $\times$ `N` chess-board;
- `bishopCnt(B)`, where `B` is the exact number of bishops to place on the chess board;
- `attackCnt(X,Y,A)`, where `A` is the number of times the tile at position `(X,Y)` is to be attacked at least.

Important

- You can assume that the first two predicates will exist only a single time, and the last one for every valid position on the board.
- Positions on the board are given by coordinates ranging from $[1, n]$.

- Placing a bishop on a tile does not release you from attacking this tile `A` times, for the given number `A`.
- A bishop on tile $(\texttt{X}, \texttt{Y})$ does not attack the tile $(\texttt{X}, \texttt{Y})$.

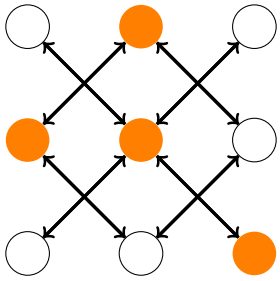
The output of your program should be given by the binary predicate $\texttt{bishopPos}(\texttt{X}, \texttt{Y})$ that represents that there is a bishop placed on position $(\texttt{X}, \texttt{Y})$ on the board.

2.2.1 Example

Consider a situation with four bishops on a 3×3 board on which each tile should be attacked at least once:

$$\texttt{size}(3).\ \texttt{bishopCnt}(4).\ \texttt{attackCnt}(1..3, 1..3, 1).$$

One feasible assignment, in graph representation, would be the following:



In the desired ASP Core-2 encoding, the corresponding answer set is then as follows:

$$\texttt{bishopPos}(2,2).\ \texttt{bishopPos}(1,2).\ \texttt{bishopPos}(2,1).\ \texttt{bishopPos}(3,3).$$

Note, that there is no solution with only three bishops.

3 University Salary Database Query (5 points)

3.1 Problem Specification

In the lecture, it was mentioned that database queries can be simulated in ASP using the observation that predicates are similar to tables in a database and the conjunction of literals in the body of a rule is similar to joins of database tables. For this exercise, we assume that a relational database containing salary information of some university is given as ASP facts and certain queries should be evaluated on that data using ASP rules.

The relational database schema is as follows:

Person		Assigned		Works For	
PersonName : Str	Salary : Int	PersonName : Str	GroupId : Str	Assistant : Str	Superior : Str

There are two tasks: the first is to compute the deficit of each group, and the second task is to also compute the deficit of the whole university. The first means to calculate the total salary received by persons in each group, where a person works in a group if at least one of the following conditions is fulfilled:

- the person is assigned to the group (table *Assigned*);
- the person works for someone who works for the group (table *Works For*).

The second task means to calculate the total deficit of the whole university. We assume all persons in the database are students in case they are not (directly or indirectly) employed by the university. Every student has to pay a constant fee of *100*. The total deficit of the university is the sum of the expenses for the employed persons minus the income from the students fees.

3.2 Problem Encoding

The information in the database tables is given by the following ASP predicates:

`person(N,S)`, `assigned(N,G)`, and `worksFor(N1,N2)`.

The answer set of your ASP program should contain as output the binary predicate `groupMinus(G,C)` for every group `G` representing that the people in the group with groupId `G` receive a salary of amount `C` in total. Additionally, there should also be a unary predicate `universityMinus(C)` denoting that the university pays an amount of `C` in total to all people in all groups.

A person can be an assistant of multiple people. You have to count their salary evenly distributed to all groups the person works for in `groupMinus`. (The input is chosen in a way such that the division works properly.) A person that works for the same group "multiple times" should only count once with their salary for this group. The salary of such a person should also count only once for `universityMinus` no matter how often they work for a group and for how many groups they work. Furthermore, people work only for the university if they work for at least one group. This means that the database may also contain persons and assistance information for persons not related to the university. Their salaries do not count for the universities expenses (nor for the expenses of a group). However, each of those people have to pay a fee of 100 (independent of their salary) that is subtracted from the minus of the whole university.

Important

- We assume that every person is uniquely identified by their name and names are given as constants (no special treatment of string constants is required for this task).
- The attributes of `worksFor` refer to the persons' name in table `person`.

3.2.1 Example

Consider a database instance as follows:

Person		Assigned		Works For	
PersonName : Str	Salary : Int	PersonName : Str	GroupId : Str	Assistant : Str	Superior : Str
a	100	c	g1	b	a
b	150	a	g2	b	c
c	1000			d	b
d	50			f	e
e	266				
f	133				

This database can also be specified in terms of the following ASP facts:

`person(a,100). person(b,150). person(c,1000). person(d,50). person(e,266). person(f,133).`

`assigned(c,g1). assigned(a,g2).`

`worksFor(b,a). worksFor(b,c). worksFor(d,b). worksFor(f,e).`

The single solution for this problem is an answer set containing the following atoms:

`universityMinus(1100). groupMinus(g1,1100). groupMinus(g2,200).`

Note that `e` and `f` do not affect the result as they are not part of one of the university's groups. Furthermore, the salary of `b` (and of `d`) is counted to 50% in each of the two groups, and once for the overall university expenses.

4 Writing Tests

Construct 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 have to design *at least five test cases* for each of the three exercises and save them in separate files; name these files `exercise[1|2|3]_n.lp`, where n is a number ($1 \leq n \leq 5$), e.g., `exercise2_3.lp`. If you do not hand in at least five test cases, you will not receive any points for the corresponding exercise.

It might be useful for you to restrict the output of Clingo to some specific predicates. You can put lines like

```
#show maps/2.
```

in your test cases to print only those atoms over the `maps` predicate with arity 2.[1] **However, make sure that you use `#show` only in your test-case files and not in your actual program!**

You can find the encoding of all examples discussed here in TUWEL. They might help you to design your own test cases.

4.1 Hints and final remarks

- All three exercises can (but do not have to) be solved within few lines of code (~ 10 lines if written succinctly).
- You are not allowed to use strong negation (only default negation)—it will not help you anyway.
- The order of predicates in the output of Clingo is irrelevant.
- Your files `exercise[1|2|3].lp` should be completely self-contained.
- You are allowed to introduce any number of additional predicates within your program.
- Make sure that your program terminates for small inputs in a few seconds.
- Your encoding must not output the same combination of output-predicates when asking the solver for multiple models. This can happen if you use auxiliary predicates internally that make the models be different, although they look equal when only showing the output-predicates (e.g., by using `#show`).
- Use Clingo v5.4.0 (or later) for testing your encodings.
- The ASP Core-2 specification can be found at

  `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf`,

  and the Clingo Guide, available at

  `http://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/clingo_guide.pdf`,

  might be useful.

5 Submission Information

Submit your solution by uploading a ZIP file with the files shown below to the TUWEL system.

Make sure that the ZIP file contains at least the following files:

- `exercise1_n.lp`, `exercise2_n.lp`, `exercise3_n.lp`, where n is an integer ($1 \leq n \leq 5$),
- `exercise1.lp`,
- `exercise2.lp`,
- `exercise3.lp`.

Make sure that your ZIP file does not contain subdirectories, and do not forget to add test files, otherwise you cannot get the full score.

[1] Note that "`#show`" is a Clingo specific keyword and is not defined by the ASP Core-2 standard.