PROJECT DESCRIPTION

==================

This (mid-size) project is part of the course titled "Advanced Problem Solving Techniques", which aims primarily at students in the Cognitive Systems program. The project work is to be done (mostly) independently during the lecture-free period, in groups of ONE to THREE students each. At the end of the lecture-free period, the developed implementation and achieved (performance) results are to be presented in a talk, based on electronic slides in PDF format.

The topic of the project is to develop an ASP encoding for elevator control. Instances are given by facts such as those in instances/instance-04_5-2.lp:

```
agent(elevator(1)).
agent(elevator(2)).
floor(1). floor(2). floor(3). floor(4). floor(5).
init(at(elevator(1),1)).
init(at(elevator(2),2)).
init(request(deliver(2),3)).
init(request(call(up),3)).
init(request(call(down),4)).
init(request(call(down),5)).
```

These facts express that there are two elevators within the same building, which has five floors. Atoms over the predicate init/1 provide the starting floors of the two elevators along with requests they have to serve. Here, the second elevator has to bring people to the third floor. In addition, calls for some elevator (each with a direction that does not matter in our scenario) are issued from the third, fourth, and fifth floor.

A delivery request is handled, once the respective elevator serves the floor of the request, and call requests are completed when any elevator serves the floor. Note that several requests concerning the same floor may be served simultaneously by a single elevator. For example, a plan to serve all requests along with corresponding states is given by atoms in a stable models such as those stored as facts in instances/instance-04_5-2.lp.sol:

```
holds(at(elevator(1),1),0).
holds(at(elevator(2),2),0).
holds(request(deliver(2),3),0).
holds(request(call(up),3),0).
holds(request(call(down),4),0).
holds(request(call(down),5),0).

do(elevator(1),move(1),1).
do(elevator(2),move(1),1).
holds(at(elevator(1),2),1).
holds(at(elevator(2),3),1).
holds(request(deliver(2),3),1).
holds(request(call(up),3),1).
holds(request(call(down),4),1).
holds(request(call(down),5),1).

do(elevator(1),move(1),2).
do(elevator(2),serve,2).
holds(at(elevator(1),3),2).
holds(at(elevator(2),3),2).
holds(request(call(down),4),2).
holds(request(call(down),5),2).

do(elevator(1),move(1),3).
do(elevator(2),move(1),3).
holds(at(elevator(1),4),3).
holds(at(elevator(2),4),3).
holds(request(call(down),4),3).
holds(request(call(down),5),3).

do(elevator(1),move(1),4).
do(elevator(2),serve,4).
holds(at(elevator(1),5),4).
holds(at(elevator(2),4),4).
holds(request(call(down),5),4).

do(elevator(1),serve,5).
holds(at(elevator(1),5),5).
holds(at(elevator(2),4),5).
```

Atoms over holds/2 for the initial state, indicated by the last argument 0, are simply obtained by "copying" the contents of the facts over init/1 given above. That is, a state is given by the current floors of elevators along with yet open requests. Atoms over do/3, first included with 1 in the last argument, provide actions performed by the elevators, and there are three possible values for the second argument: move(1), move(-1), and serve. They express whether an elevator moves up, down, or serves the floor where it currently is. Clearly, elevators can only move up or down if there is a next floor in that direction, and serving is only possible if there is an open request that can be handled by the respective elevator. In particular, take into account that serving a floor is admissible only if there is a yet unhandled delivery request for the respective elevator or some call request that has not been completed before.

The actions lead to successor states with the same integer as the last argument in atoms over holds/2 as included in do/3. For example, both elevators are one floor higher than before in the state obtained by performing move(1) as first action by both of them, while all four requests are still open. The second elevator then serves two requests concerning the third floor, so that these two requests are no longer open in the state referred to by 2. Otherwise, the elevators keep moving to floors with yet open requests, the second elevator serves the fourth floor as its fourth and final action, and the first elevator serves the fifth floor as its final action. Finally, all requests are handled and the plan is complete. Note that an elevator does not have to do an action at each point in time, e.g., the second elevator performs one action less than the first one, but an elevator can do at most one action per time point.

FRAMEWORK AND REQUIREMENTS

========================

Optimal plans such as the above shall be computed by clingo (version 5.3.0), once a respective ASP encoding has been implemented in elevator.lp. An optimal plan complies with two objectives in the following order of preference:

1. The number of time points, i.e., states traversed while executing the plan, is minimal. This essentially means that all requests shall be handled as soon as possible.

2. The total number of moves (actions indicated by "move(1)" and "move(-1)") shall be minimized as secondary criterion.

The ASP encoding must be an incremental program (clingo's built-in "incmode" is included in elevator.lp) with a base, a step(t), and a check(t) subprogram as its three parts. The format of (optimal) plans must be exactly as described above to enable checking (by YETI) and visualization (see below) of solutions. Please keep the #show directives included in elevator.lp as they are, to make sure that the relevant atoms of a stable model are contained in the output. During the encoding development, you may of course add further #show statements on your own to inspect stable models, but please remove your own statements when you upload your encoding in YETI for (mandatory) correctness checking.

For further information about clingo's "incmode", you may, e.g., consult the following resources:

- Lecture slides on Multi-shot Solving:
  http://www.cs.uni-potsdam.de/~torsten/Potassco/Slides/msolving.pdf

- Article on clingo's scripting interface (part 1,2 and 3):
  https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/rweb/KaminskiSW17.pdf

- Examples downloaded with the clingo sources (examples/clingo subfolder):
  https://github.com/potassco/clingo/releases/tag/v5.3.0

  * Especially consider the Python re-implementation of the built-in "incmode" contained in examples/clingo/iclingo/incmode-py.lp

  * A typical example that makes use of the "incmode" is shipped in examples/clingo/solitaire/solitaire.lp

The provided archive with this README also includes a prototypical visualizer and an instance generator. Both are briefly described below, their usage is optional, and possibly they may be helpful while working on the project.

Instances are provided in the instances subfolder within this archive. (The .lp files include instance data, and corresponding .lp.sol files respective optimal example plans, which may be visualized. For some of the larger instances, the computation of optimal plans is time-consuming, and .lp.sol files are not yet available for a few of the instances.) Please ignore the #const directives in the files, which are included for the visualizer only. However, in case you are yet unexperienced with incremental programs, consider to develop a "standard" one-shot encoding first, assuming some fixed maximum number of states/actions, before adapting the encoding to be incremental.

Getting started with the visualizer

===================================

1. Download clingo version 5.3.0 sources from:

   https://github.com/potassco/clingo/releases/tag/v5.3.0

   Build and install the Python Module (clingo.{so,dll,dylib}) as described in
   the downloaded INSTALL.md file.

2. Install any Python components (e.g., python-qt4 under Linux) needed to
   launch the visualizer. Try

```
$ python visualizer/Elevator.py -h
```

   to see whether the visualizer is ready to run.

3. Launch the visualizer for some instance, e.g., using the following command:

```
$ python visualizer/Elevator.py \
  -i instances/instance-04_5-2.lp \
  -e instances/instance-04_5-2.lp.sol
```

4. Try the buttons and tabs in the Elevator window to trace the example plan
   given by facts in the "encoding" file specified via the -e option.

[5. In principle, the visualizer could be used to calculate stable models
    representing solutions, if an (incremental) encoding like the one to be
    developed in elevator.lp is provided. However, it is recommended to
    calculate plans offline, and you may use getPlan.sh for this purpose. Try

```
    $ ./getPlan.sh
```

to see how the script is run.]

Getting started with the instance generator
==

1. To see the available options, launch the instance generator as follows:

```
   $ python generator/InstanceGenerator.py -h
```