Analyzing InstaHub

Andrzej Filinski (andrzej@di.ku.dk), Ken Friis Larsen (kflarsen@di.ku.dk), and Oleksandr Shturmov (oleks@oleks.info)

Last revision: September 23, 2020

Note: This assignment also includes a collection of simple warm-up exercises, as described on Absalon. For those, you are *only* asked to submit your working code, but not a separate design/implementation document, assessment, or evidence of testing. If you need to communicate anything extra about your solutions to the warm-up exercises, place your remarks as comments in the source code.

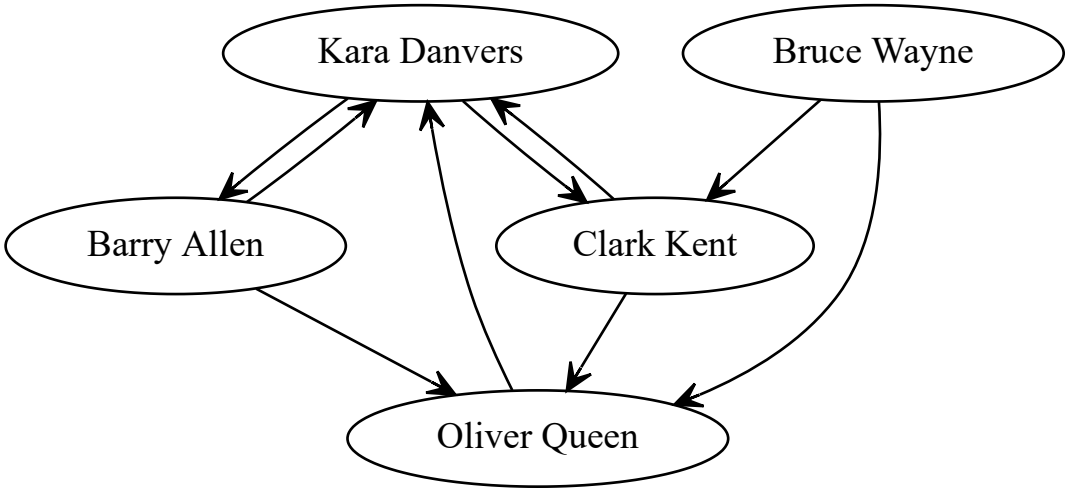
Background

The objective of this assignment is to gain hands-on programming experience with Prolog, and in particular with declarative characterizations of both positive and negative knowledge.

This assignment is about analyzing the social relations in a fictional social network called “InstaHub”.

An InstaHub *follower graph* consists of a set of people, where each person has a set of *other* people whose updates that person follows. The graph is represented as a list of Prolog terms, each of the form `person(X, [X1, ..., Xn])`, where `X` is an atom representing a person, and `X1`,...,`Xn` (where $n \geq 0$) are a *subscription list* of the people whom `X` follows, in no particular order, without duplicates, and not including `X` itself. The graph contains only one person by any given name X, and all members of the network have a `person`-record with a (possibly empty) subscription list.

For example, consider the following follower graph G_1, where each node represents a person on InstaHub, and a directed edge from X to Y denotes that X is following Y.



This graph can be represented with (for instance) the Prolog term:

```
[person(kara, [barry, clark]),
 person(bruce, [clark, oliver]),
 person(barry, [kara, oliver]),
 person(clark, [oliver, kara]),
 person(oliver, [kara])]
```

For easy experimentation, you can declare a Prolog fact corresponding to this graph:

```
g1([person(kara, [barry, clark]), person(bruce, [clark, oliver]),
...]).
```

Then the query `?- g1(_G), some_pred(_G, X, Y).` will instantiate the variable `_G` to the above representation of G_1 for the second subgoal. Note that, by default, SWI Prolog will repeatedly print the (largish) value of `_G` with every answer it finds, but you can override that by executing the query:

```
?- set_prolog_flag(toplevel_print_anon, false).
```

Alternatively, you may put the above line in your `.swiplrc` (Unix) or `swipl.ini` (Windows), or invoke SWI Prolog as

```
swipl -g 'set_prolog_flag(toplevel_print_anon, false)'
```

Do not put any such customizations in the code you actually hand in, as it will cause it to be flagged as using impure Prolog features.

(GNU Prolog already suppresses reporting of answer variables starting with an underscore in the standard setup.)

Predicates

In the following, you may assume that `G` and `H` are fully instantiated (i.e., ground), well-formed Prolog representations of follower graphs.

Caution: The following formal specifications of various suggestively named predicates do not necessarily match the common real-world meanings of the names; for example, there's *a priori* nothing preventing someone from being considered both "friendly" and "hostile". Just implement the predicates as specified, without worrying about whether the specification makes intuitive sense in all cases.

Level 0: pairwise connections

a. We say that *X follows Y* in the graph, if *Y* is on *X*'s subscription list (but not necessarily vice versa). For example, in G_1, Bruce follows Clark.

   Write a predicate `follows(G, X, Y)` that is true whenever `X` follows `Y` in the graph `G`. For example `?- g1(_G), follows(_G, bruce, clark).` should succeed, while the same query with the two names swapped should fail. A query where `X` and/or `Y` are uninstantiated variables should succeed repeatedly with all relevant pairs (and then fail).

b. In general, many people in the network may not even be aware of each other, so *not* being on someone's subscription list is not in itself significant. However, we say that *X ignores Y* if *Y* follows *X*, but *X* does not follow *Y* back. Thus, in G_1, we would say that Clark ignores Bruce, while he neither follows nor ignores Barry.

   Write a predicate `ignores(G, X, Y)` that succeeds whenever `X` ignores `Y` in the graph `G`. *Hint:* start by defining an auxiliary predicate `different(G, X, Y)` that succeeds whenever `X` and `Y` are different members of the network `G`, where you exploit that `G` lists each member exactly once.

Level 1: local connections

c. We say that a network member *X* is *popular* if everyone whom *X* follows, follows him/her back. For example in G_1, Kara is popular, while Clark is not.

   Write a predicate `popular(G, X)` that succeeds whenever `X` is popular in `G`.

d. Conversely, a network member *X* is an *outcast* if everybody whom *X* follows, ignores (i.e., fails to follow back) him/her. For example, in G_1 Bruce is an outcast.

   Write a predicate `outcast(G, X)` that succeeds whenever `X` is an outcast in `G`.

e. A network member *X* is said to be *friendly* if *X* follows back everyone who follows him/her. For example, in G_1, Barry is friendly.

   Write a predicate `friendly(G, X)` that succeeds whenever `X` is friendly in `G`.

f. Conversely, a network member *X* is *hostile* if *X* ignores everyone who follows him/her. For example, in G_1, Oliver is hostile.

   Write a predicate `hostile(G, X)` that succeeds whenever `X` is hostile in `G`.

Note that the first two predicates above consider how others behave towards `X`, while the last two look at how `X` behaves towards others.

Level 2: global connections

g. We say that *X* is *aware of* some other person *Y* if *X* follows someone who either is *Y*, or is themselves aware of *Y* – i.e., if there is a chain of followings from *X* to *Y*. For example, in G_1, Bruce is aware of Kara (via both Clark and Oliver), while the opposite is not true.

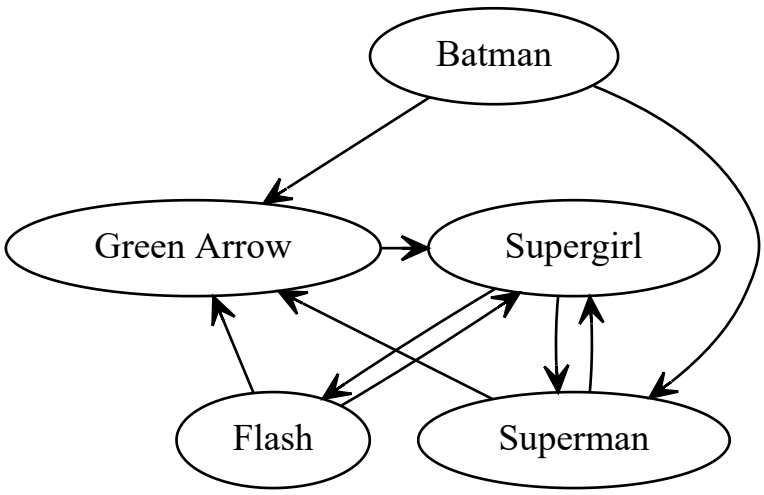
   Write a predicate `aware(G, X, Y)` that succeeds whenever `X` is aware of `Y` in `G`. Note that the graph `G` may contain cycles, which you'll need to deal with in order to avoid going into an infinite recursion.

h. We say that *X* is *ignorant of* some other person *Y* if *X* is *not* aware of *Y*, in the sense above. Thus, for any network members *X* and *Y*, exactly one of the following should hold: (1) *X* is aware of *Y*, (2) *X* is ignorant of *Y*, or (3) *X* and *Y* are the same person.

   Write a predicate `ignorant(G, X, Y)` that succeeds whenever `X` is ignorant of `Y` in `G`. *Hint:* start by constructing (e.g., by a flooding algorithm) a list containing `X` and all people of whom `X` is aware, and then check that `Y` is *not* on that list. (You may want to first code up a function for constructing such a list in your favorite language (Haskell), and then convert your solution to Prolog, once it works.)

Level 3: whole-graph properties

i. Sometimes two follower graphs may describe identical connection patterns, except for the *identities* of the people involved, and the orders in which they are listed. We say that two graphs *G* and *H* describe the *same world* if there is a one-to-one correspondence between the people in one and the other, such that everyone is following exactly the corresponding sets of people in the two graphs. For example, consider the following follower graph G_2:



   which can be represented with the following Prolog term:

```
[person(batman, [green_arrow, superman]),
 person(green_arrow, [supergirl]),
 person(supergirl, [flash, superman]),
 person(flash, [green_arrow, supergirl]),
 person(superman, [green_arrow, supergirl])]
```

Write a predicate `same_world(G, H, K)` that succeeds whenever `G` and `H` describe the same social graph, and `K` is the correspondence *key*, i.e., a list of pairs `p(X,Y)` that says, for each person `X` in `G`, who the corresponding person `Y` is in `H`. (`K` should enumerate the `X`s in the same order as in `G`.) In the example above, with `G` and `H` taken as the Prolog representations of G_1 and G_2 above, the call should thus succeed with

```
K = [p(kara,supergirl), p(bruce,batman), p(barry,flash),
p(clark,superman), p(oliver,green_arrow)]
```

j. **Optional!** Write a predicate `different_world(G, H)` that succeeds *if and only if* `same_world(G, H, _)` would fail.

Restrictions

Your predicates should work for both checking and enumerating solutions. The predicates should eventually fail (rather than loop forever) if the predicate does not hold of its arguments, or after finding all the solutions.

It is not a requirement that every solution be found exactly once, but you will probably find it easier to argue/test that your program is correct (i.e., allows neither too few nor too many solutions) if it only finds the same solution multiple times because it is a solution for different reasons. Keep in mind that the sample graph G_1 is not necessarily sufficient for properly testing all the predicates you are asked to implement.

For this assignment, efficiency is *not* a major concern; that is, you need not worry about whether your programs would scale to significantly larger graphs than in the examples.

Important: Your program must consist of pure facts and rules only; do not use *any* built-in Prolog predicates or control operators – including in particular, but not limited to: equality/inequality ("`=`" and "`\=`"), arithmetic ("`is`"), disjunction ("`;`"), cuts ("`!`"), or negation-as-failure ("`\+`"). If you need any standard utility predicates, such as `select/3` (*Hint*: extremely useful for many parts of this assignment!), include their definitions in your program, but be sure to name them something different from the built-in ones.

Exception: For your *testing*, you are allowed to use all of Prolog and whatever standard predicates or extensions SWI Prolog offers, such as `setof/3`. You may, but are not required to, use the `plunit` test framework to organize your tests.

Hand-in instructions

You should hand in two things:

1. A short report, `report.pdf`, explaining your high-level design/implementation choices (in particular, which algorithms/approaches do you use for the various predicates?), and containing an assessment of your implementation, including what this assessment is based upon. In the report, you should also:

   a. Demonstrate that your predicates work on suitable examples.

   b. Tell us how we can run your code, and reproduce your test results.

   Finally, the report should contain a listing of your implementation and tests in an appendix.

2. A ZIP archive `code.zip`, mirroring the structure of the handout, and containing your source code and tests. Your implementation of the InstaHub predicates above (as well as any auxiliary predicates used in their definitions) should be in the file `code/part2/src/instahub.pl`; your tests should be in `code/part2/tests/instatest.pl`.

To keep your TA happy, follow these simple rules:

1. Clean up your code before you submit.

2. SWI Prolog should not yield any errors or warnings for your code.

3. You should comment your code properly, especially if you doubt the correctness of something, or if you think there is a more elegant way to write a certain piece of code. A good comment should explain your ideas and provide insight that the code otherwise does not. As always, try to write short specifications for any auxiliary functions you define, explaining in particular the meanings of all parameters.

4. Adhere to the restrictions set in the assignment text. **Only use pure Prolog in your solution.**

End of assignment text.