AP assignment 3: Analyzing Instahub

sortraev, wlc376

2020-09-30

0 Design and Implementation

In this section, I present and explain a sample of design choices made in my implementation.

The handed-in code in `code/part2/src` (see appendix A) is supplied with simple code comments where appropriate.

0.1 Names and persons

Each node in an Instahub graph `G` is a `person(X, X_adj)` functor, where `X` is the name of this particular person, and `X_adj` is `X`'s adjacency list in `G`.

Quickly, a great amount of confusion ensues as the need for distinction between *names* and *persons* arises. For example, say a given predicate requires iterating the list of nodes in the (sub-)graph; does it explicitly need to iterate the *persons* in the graph, or simply the names of those persons?

If only names are necessary, then surely carrying the full list of persons is redundant complexity; however, there might be cases where discarding this information early is destructive.

In my implementation, I choose to discard the `person` wrappers and concern myself only with names wherever possible; this has the proposed benefit of smaller complexity. Whenever the need arises to look into a person's adjacency list, this can be looked up in the full graph, which is typically stringed along in predicates due to its necessity in testing list non-membership and inequality given the restrictions on third-party predicates (which I will assume the reader is familiar with and thus not go into detail with).

0.2 Friendliness, hostility, and incoming adjacency

`popular(G, X)` and `outcast(G, X)` each require looking up the adjacency list of the person in `G` corresponding to the queried name `X`; this is quite straight-forwardly handled by iteration of `G`.

However, the `friendly(G, X)` and `hostile(G, X)` predicates are somewhat more complicated, as these require looking up the incoming adjacency list; that is, the list of names of persons in `G` who follow `X`. Below is a snippet of my solution to computing incoming adjacency:

```
% inAdj(G, G, X, X_inAdj) extracts the list of X's incoming adjacents in the
% graph G if X is the name of a person in G. I don't know how to make the
% predicate work without first asserting X as a member of G.
inAdj(G, X, X_inAdj) :-
  elem(person(X, _), G),
  inAdj_(G, G, X, X_inAdj).

inAdj_(_, [], _, []).
inAdj_(G, [person(_, Y_adj) | T], X, X_inAdj) :-
  notElem(G, X, Y_adj),     % required, since we might have Y = X.
  inAdj_(G, T, X, X_inAdj).
inAdj_(G, [person(Y, Y_adj) | T], X, [Y | X_inAdj]) :-
  elem(X, Y_adj),
  inAdj_(G, T, X, X_inAdj).
```

0.3 Awareness, ignorance, and graph search

In implementing the predicates `aware/3` and `ignorant/3`, I have taken two largely different strategies. Below is a snippet of my `aware/3`:

```
% My `aware/3` simply encodes the transitive closure of the `following`
% relation. To avoid infinite recursion, I keep track of previously visited
% persons.
aware(G, X, Y) :-
  different(G, X, Y),
  reachable(G, X, Y, []).

reachable(G, X, Y, _Visited) :-
  follows(G, X, Y).
reachable(G, X, Y, Visited) :-
  notElem(G, X, Visited),
  follows(G, X, Z),
  reachable(G, Z, Y, [X | Visited]).
```

My initial thought was to simply encode the transitiuve closure of the *following* relation; `X` is aware of `Y` if `X` follows `Y` or follows some `Z` who is aware of `Y`. To avoid infinite recursion on cyclic graphs, I use a `Visited` list to stop recursing on known persons in the graph.

This implementation was largely straight-forward and trivial. Beginning work on `ignorant(G, X, Y)`, I expected the same strategy to work; use the transitive closure of the `notFollows` relation (as I called it) to compute ignorance. This, of course, did not work, since the *ignorance* relation is not transitive. How embarassing.

My next thought was to use my recently finished implementation of `aware/3` along with the hint given in the assignment text, which was to flood the graph in order to build a list of reachable nodes, which `Y` could then be as a non-member of. However, I ran into a lot of problems trying to follow this strategy, none of which I have unfortunately documented.

Finally, I settled on a solution which more or less implements a breadth-first search of the network graph, starting with `X` as the singular known, but yet unprocessed person, recursing until all reachable persons have been processed. Below is a snippet of the finished implementation of the predicate, which should document itself in code comments:

```
ignorant(G, X, Y) :-
  different(G, X, Y),
  reachables(G, [X], [], Reachable_from_X),
  notElem(G, Y, Reachable_from_X).

% reachables maintains a list of nodes which at this point are known to be
% reachable from X but which are to be recursively processed for reachability.
reachables(G, [X | Todo], Visited, Final) :-
  notElem(G, X, Visited),
  adj(G, X, X_adj),
  concat(Todo, X_adj, Todo_), % BFS; swap Todo and X_adj to get DFS.
  reachables(G, Todo_, [X | Visited], Final).

reachables(G, [X | Todo], Visited, Final) :-
  elem(X, Visited),
  reachables(G, Todo, Visited, Final).

% no more unprocessed nodes? copy visited nodes to final!
reachables(_G, [], Visited, Visited).
```

0.4 Implementation limitations

I unfortunately did not manage to get a good attempt at solving the level 3 predicates `same_world/3` or `different_world/2`. As such, they are left commented-out in the code hand-in, and I will not ignore them in my testing.

1 Testing

In this section, I discuss my validation test plan and report the results of testing.

1.1 Reproduction of tests

To reproduce my tests, navigate to `code/` of the code hand-in and run `\$ make` or `\$make test` using the supplied `Makefile`.

I use `plunit` for testing. All of my tests can be viewed in `code/part2/tests/instatest.pl` or appendix B to this report.

1.2 Testing goals

The goal of the test plan is to attain full edge case coverage with unit testing of each implemented predicate and helper predicate.

1.3 Test plan

1.3.1 Test plan: strategy

I want to test correct evaluation of those of the predicates which I managed to implement (as mentioned in section 0.4, everything but level 3 predicates).

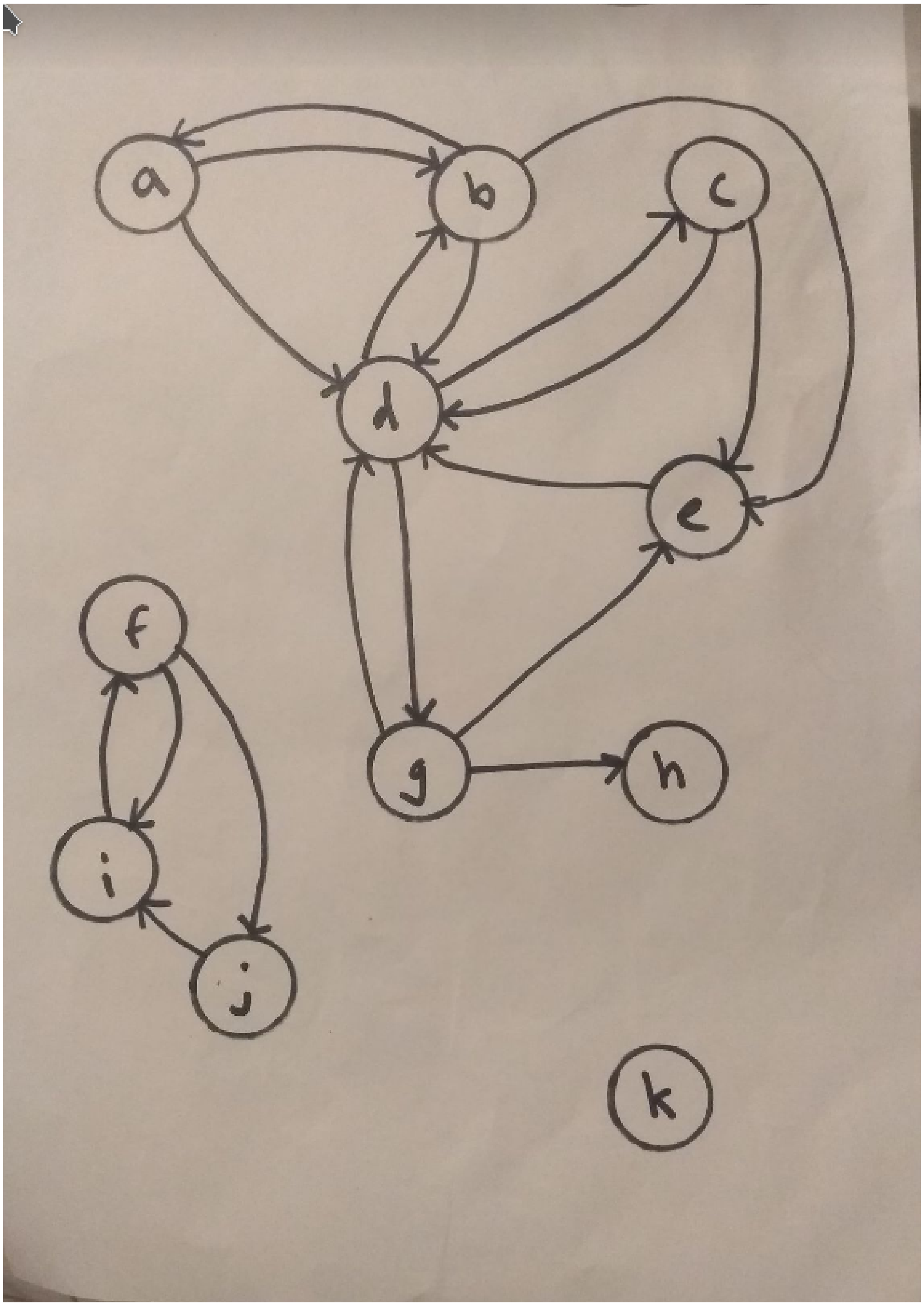
I argue that this should be adequate to simultaneously obtain coverage of all implemented helper predicates.

1.3.2 Testing: graph used

In testing my predicates, I want to use a test graph containing all the types of inter-person relationships covered by the implemented predicates.

Additionally, I want a test graph containing multiple, disjoint sub-graphs, as this might make for interesting edge cases.

I argue that a graph satisfying these specifications is sufficient for validation testing and, with pen and paper, construct a small and simple example of such a graph, pictured below:



The Prolog encoding of this graph can be found at the top of `code/tests/instatest.pl`.

1.3.3 Testing: test suite

In devising my test suite, I attempt to cover as many edge cases as possible; incoming and outgoing relations,

For lack of time, I won't go further detail with my test suite in this report. Each test has a (somewhat) fitting name explaining (to some degree) what that test asserts, so I will instead refer the extraordinarily curious reader to view the test cases in the test source code (as explained in 1.1).

1.4 Validation testing results

All of my own validation tests pass successfully.

All of OnlineTA's validation tests pass successfully, **save for those tests involving level 3 predicates `same_world/3` and `different_world/3`** (which were not expected to pass; see section 0.4).

1.5 Evaluation

My final test suite includes 42 tests, and I manage to cover a great number of interesting test cases. I succumb to the deadline and unfortunately do not rigorously seek out possibly uncovered edge cases; as such, I am only

Based on validation test results alone, I am almost convinced that my implementation is sound; I would have liked to have rigorously sought out possibly uncovered edge cases, but once again, I succumb to the deadline.

Aside from validation test results, I am generally satisfied with my implementation and what it has taught me of programming in Prolog and of declarative programming in general, since my only prior experience has been in Verilog, which, curiously, was an *entirely* different experience.

Appendix

A code/part2/src/instahub.pl

```
%%% instahub

/*
 * level 0
 */
follows(G, X, Y) :-
  adj(G, X, X_adj),
  elem(Y, X_adj).

ignores(G, X, Y) :-
  follows(G, Y, X),
  notFollows(G, X, Y).

notFollows(G, X, Y) :-
  adj(G, X, X_adj),
  notElem(G, Y, X_adj).


/*
 * level 1
 */
popular(G, X) :-
  adj(G, X, X_adj),
  allFollow(G, X_adj, X).

outcast(G, X) :-
  adj(G, X, X_adj),
  noneFollow(G, X_adj, X).


% friendly and hostile both require X to actually be a member of G. This
% specification is not immediately discernable from the assignment text, but can
% be inferred from onlineTA's complaints when the `elem`s are commented out.
friendly(G, X) :-
  elem(person(X, _), G),
  inAdj(G, G, X, X_inAdj),
  followsAll(G, X_inAdj, X).

hostile(G, X) :-
  elem(person(X, _), G),
  inAdj(G, G, X, X_inAdj),
  followsNone(G, X_inAdj, X).


/*
 * level 2
 */
% My `aware/3` simply encodes the transitive closure of the `following`
% relation. To avoid infinite recursion, I keep track of previously visited
% persons.
aware(G, X, Y) :-
  different(G, X, Y),
  reachable(G, X, Y, []).
```

```
reachable(G, X, Y, _Visited) :-
  follows(G, X, Y).
reachable(G, X, Y, Visited) :-
  notElem(G, X, Visited),
  follows(G, X, Z),
  reachable(G, Z, Y, [X | Visited]).



% ignorant uses a slightly different tactic than aware; first, `reachables`
% performs a breadth-first search to find all nodes reachable from X; then, Y is
% asserted to not be a member of the set of nodes reachable from X.
ignorant(G, X, Y) :-
  different(G, X, Y),
  reachables(G, [X], [], Reachable_from_X),
  notElem(G, Y, Reachable_from_X).


% reachables maintains a list of nodes which at this point are known to be
% reachable from X but which are to be recursively processed for reachability.
reachables(G, [X | Todo], Visited, Final) :-
  notElem(G, X, Visited),
  adj(G, X, X_adj),
  concat(Todo, X_adj, Todo_), % BFS; flip Todo and X_adj to get DFS.
  reachables(G, Todo_, [X | Visited], Final).

reachables(G, [X | Todo], Visited, Final) :-
  elem(X, Visited),
  reachables(G, Todo, Visited, Final).

%% no more unprocessed nodes? copy visited nodes to final!
reachables(_G, [], Visited, Visited).


/*
 * level 3 - nope! not attempted.
 */
% same_world(G, H, K)
% different_world(G, H)


/*
 * level 1 specific helper predicates
 */
allFollow(_, [], _).
allFollow(G, [Y | T], X) :-
  follows(G, Y, X),
  allFollow(G, T, X).

noneFollow(_, [], _).
noneFollow(G, [Y | T], X) :-
  notFollows(G, Y, X),
  noneFollow(G, T, X).

followsAll(_, [], _).
followsAll(G, [Y | T], X) :-
  follows(G, X, Y),
  followsAll(G, T, X).

followsNone(_, [], _).
followsNone(G, [Y | T], X) :-
  notFollows(G, X, Y),
  followsNone(G, T, X).

```

```prolog

/*
 * helper predicates
 */
elem(X, [X | _]).
elem(X, [_ | T]) :- elem(X, T).

% notElem(G, X, Gsubset) succeeds if X is *not* in Gsubset. Needs G to determine
% if X is different from other persons.
notElem(_, _, []).
notElem(G, X, [Y | T]) :-
  different(G, X, Y),
  notElem(G, X, T).

% different(G, X, Y) succeeds if X and Y are persons of the graph G and X != Y.
different(G, X, Y) :-
  removeFirst(G, person(X, _), G_minus_X),
  elem(person(Y, _), G_minus_X).

% removeFirst(X, G, G_minus_X) removes first occurence of X in G, storing
% resulting resulting graph in G_minus_X.
removeFirst([X | T], X, T).
removeFirst([Y | T], X, [Y | T_minus_X]) :-
  removeFirst(T, X, T_minus_X).


% adj(G, X, X_adj) extracts X's adjacency list in the graph G if X is the name
% of a person in G.
adj([person(X, X_adj) | _], X, X_adj).
adj([_ | T], X, X_adj) :-
  adj(T, X, X_adj).


% inAdj(G, G, X, X_inAdj) extracts the list of X's incoming adjacents in the
% graph G if X is the name of a person in G.
inAdj(_, [], _, []).
inAdj(G, [person(_, Y_adj) | T], X, X_inAdj) :-
  notElem(G, X, Y_adj),
  inAdj(G, T, X, X_inAdj).
inAdj(G, [person(Y, Y_adj) | T], X, [Y | X_inAdj]) :-
  elem(X, Y_adj),
  inAdj(G, T, X, X_inAdj).

concat([], SomeList, SomeList).
concat([Head | L1], L2, [Head | L3]) :-
  concat(L1, L2, L3).

/*
 * sample network (the one used for testing).
 */
% g2([person(a, [b, d]),
%     person(b, [a, d, e]),
%     person(c, [d, e]),
%     person(d, [b, c, g]),
%     person(e, [d]),
%     person(f, [i, j]),
%     person(g, [d, e, h]),
%     person(h, []),
%     person(i, [f]),
%     person(j, [i]),
%     person(k, [])]).
```

B code/part2/tests/instatest.pl

```
%%% instahub tests.

% see report for a pretty drawing of this graph.
g1([person(a, [b, d]),
    person(b, [a, d, e]),
    person(c, [d, e]),
    person(d, [b, c, g]),
    person(e, [d]),
    person(f, [i, j]),
    person(g, [d, e, h]),
    person(h, []),
    person(i, [f]),
    person(j, [i]),
    person(k, [])]).
........ .............. ..................
:- begin_tests(level0_tests).

% follows/3
test("multiple incoming follows",  set(X = [a, b, c, e, g])) :-
  g1(G), follows(G, X, d).
test("multiple outgoing follows", set(X = [d, b])) :-
  g1(G), follows(G, a, X).
test("no outgoing follows.",      set(X = [])) :-
  g1(G), follows(G, h, X).
test("no incoming follows.",      set(X = [])) :-
  g1(G), follows(G, X, k).
test("following of non-existing person ", set(Y = [])) :-
  g1(G), follows(G, Y, anders).

% ignores/3
test("multiple incoming ignores", set(X = [e, h])) :-
  g1(G), ignores(G, X, g).
test("multiple outgoing ignores", set(X = [b, c, g])) :-
  g1(G), ignores(G, e, X).
test("following of non-existing person", set(Y = [])) :-
  g1(G), ignores(G, Y, anders).

:- end_tests(level0_tests).



:- begin_tests(level1_tests).
% popular/2

test("all populars", set(X = [d, h, i, k])) :-
  g1(G), popular(G, X).
test("all outcasts", set(X = [e, h, j, k])) :-
  g1(G), outcast(G, X).
test("all friendlies", set(X = [a, b, c, f, g, k])) :-
  g1(G), friendly(G, X).
test("all hostiles", set(X = [e, h, j, k])) :-
  g1(G), hostile(G, X).


test("popular and outcast", set(X = [h, k])) :-
  g1(G), popular(G, X), outcast(G, X).
test("popular and friendly", set(X = [k])) :-
  g1(G), popular(G, X), friendly(G, X).
test("popular and hostile", set(X = [h, k])) :-
```

```
  g1(G), popular(G, X), hostile(G, X).
test("outcast and friendly", set(X = [k])) :-
  g1(G), outcast(G, X), friendly(G, X).
test("outcast and hostile", set(X = [e, h, j, k])) :-
  g1(G), outcast(G, X), hostile(G, X).
test("friendly and hostile", set(X = [k])) :-
  g1(G), friendly(G, X), hostile(G, X).

test("popular, outcast, friendly, and hostile", set(X = [k])) :-
  g1(G), popular(G, X), outcast(G, X), friendly(G, X), hostile(G, X).

test("popularity of non-existing person", fail) :-
  g1(G), popular(G, anders).
test("isolarity of non-existing person", fail) :-
  g1(G), outcast(G, anders).
test("friendliness of non-existing person", fail) :-
  g1(G), friendly(G, anders).
test("hostility of non-existing person", fail) :-
  g1(G), hostile(G, anders).


:- end_tests(level1_tests).



:- begin_tests(level2_tests).
/*
 * aware tests
 */
test("irreflexitivity of awareness", set(X = [])) :-
  g1(G), aware(G, X, X).
test("transitive awareness", nondet) :-
  g1(G), aware(G, a, d), aware(G, d, h), aware(G, a, h).
test("awareness of non-existing person", set(X = [])) :-
  g1(G), aware(G, X, anders).

test("multiple incoming awarenesses", set(X = [e, c, a, b, g])) :-
  g1(G), aware(G, X, d).
test("multiple outgoing awarenesses", set(X = [b, c, d, e, g, h])) :-
  g1(G), aware(G, a, X).

test("awareness in a disconnected graph 1", set(X = [i, j])) :-
  g1(G), aware(G, f, X).
test("awareness in a disconnected graph 2", set(X = [])) :-
  g1(G), aware(G, f, X), aware(G, d, X).
test("awareness in a disconnected graph 3", set(X = [])) :-
  g1(G), aware(G, X, a), aware(G, X, k).
test("awareness in a disconnected graph 4", set(X = [])) :-
  g1(G), aware(G, X, f), aware(G, X, k).



/*
 * ignorant tests
 */
test("irreflexitivity of ignorance", set(X = [])) :-
  g1(G), ignorant(G, X, X).
test("symmetric ignorance", set( Y = [f, i, j, k])) :-
  g1(G), ignorant(G, a, Y), ignorant(G, Y, a).
test("transitive ignorance", set(Y = [k])) :-
  g1(G), ignorant(G, a, Y), ignorant(G, Y, f), ignorant(G, a, f).


test("multiple incoming ignorances", set(X = [f, i, h, j, k])) :-
  g1(G), ignorant(G, X, d).
```

```
test("multiple outgoing ignorances", set(X = [f, i, j, k])) :-
  g1(G), ignorant(G, e, X).


test("ignorance in a disconnected graph 1", set(X = [a, b, e, c, d, g, h, k])) :-
  g1(G), ignorant(G, i, X).
test("ignorance in a disconnected graph 2", set(X = [a, b, e, c, d, g, h, k])) :-
  g1(G), ignorant(G, f, X).
test("ignorance in a disconnected graph 3", set(X = [k])) :-
  g1(G), ignorant(G, f, X), ignorant(G, d, X).
test("ignorance in a disconnected graph 4", set(X = [h])) :-
  g1(G), ignorant(G, X, a), ignorant(G, X, k), ignorant(G, X, f).
test("ignorance of non-existing person", set(X = [])) :-
  g1(G), ignorant(G, anders, X).


:- end_tests(level2_tests).
```