Advanced Programming 2017
Assignment 3

Tom Jager
dgr418@alumni.ku.dk

Tobias Ludwig
fqj315@alumni.ku.dk

October 4, 2017

1 Warm Up

1.1 Likes

`likes(G,X,Y)` is a simple predicate that is used to determine whether person `Y` is inside person `X`'s friend list. `likes` uses the helper functions `equal` and `elem`. `equal` is a very simple predicate that takes two arguments and passes only if they are the same.

```
equal(X,X).
```

`elem` takes an element and a list and checks if the first element of the list is the same as the given element. If it isn't it discards the head of the list and performs `elem` on the rest of the list.

```
elem(E, [E|_]).
elem(E, [_|T]) :-
  elem(E, T).
```

`likes` is implemented with two clauses. The first checks if the first element of the graph is `X` and if so binds variable `L` to be their friends list through the use of `equal`. It then checks if `Y` is inside list `L` through the use of equal. If both checks pass, it returns true. If it is not true the seconds clause is performed which is a simple recursive call. Here `likes` is performed again but without the first element of `G`, allowing us to iterate through the graph.

1.2 Dislikes

`dislikes(G, X, Y)` is a predicate that takes a graph and two people `X` and `Y` and returns true if `Y` is not in `X`'s friends list but `X` is still aware of `Y`. This rule uses the helper rules `different`, `member` and `not_in`.

`different(G, X, Y)` is a goal that returns true if `X` and `Y` are different people. It does this by using `select1` to remove person `X` from the graph. It then checks if person `Y` is still part of the graph. If it is not then `X` and `Y` were the same people and `different` returns false, otherwise it returns true.

```
different(G, X, Y) :-
  member(G, X, _),
  member(G, Y, _),
  select1(person(X, _), G, R),
  elem(person(Y, _), R).
```

`member(G,X,XFs)` is a goal that takes a person `X` and their list of friends, `XFs` and checks if the `person` constructed from that, is in the graph. It can be used to retrieve someone's list of friends.

```
member(G, X, XFs) :-
  elem(person(X, XFs), G).
```

`not_in(G,X,L)` is a goal that checks if person `X` is in a list of people `L`. It does this by iterating through the list of people and ensuring that is is `different` to each member.

```
not_in(_, _, []).
not_in(G, X, [H|T]) :-
  different(G, X, H),
  not_in(G, X, T).
```

`dislikes(G,X,Y)` is true only if `Y` likes `X`, is `different` to `X`, and is `not_in` `X`'s friends list. `X`'s friends list is extracted by using `member`.

```
dislikes(G, X, Y) :-
  likes(G, Y, X),
  different(G, X, Y),
  member(G, X, XFs),
  not_in(G, Y, XFs).
```

1.3 Testing

`Likes` and `dislikes` are tested through the use of the query`?- g1(G), warmup_test(G)` in `tests.pl`. This checks to ensure that the right people like and dislike the correct people in test graph `G` and that erroneous tests do not succeed.

```
warmup_test(G) :-
    warm_test(G),
    \+(warm_err(G)).
```

```
warm_test(G) :-
    likes(G,kara,barry),
    likes(G,kara,clark),
    dislikes(G,kara,oliver).
```

```
warm_err(G) :-
    dislikes(G,kara,clark);
    likes(G,oliver,bruce).
```

2 Local Relations

2.1 Popular

`popular(G, X)` is a predicate that determines if a person `X` is liked by everyone in their friends list. It involves the use of the helper goal `all_like(G, YFs, Y)`. This helper predicate determines if each person in a friends list `YFs` likes person `Y`. `popular` simply checks if a person `X` is a member of the graph and uses it to extract their friends list. It then performs `all_like` on `X` and their friends list.

```
popular(G, X) :-
    member(G, X, XFs),
    all_like(G, XFs, X).
```

```
all_like(_, [], _).
all_like(G, [H|T], Y) :-
    likes(G, H, Y),
    all_like(G, T, Y).
```

2.2 Outcast

The predicate to determine if everyone in someone's friends list dislikes them, is called `outcast`. `outcast(G, X)` performs in a similar way to `popular` where `member` is used to check that person `X` is part of graph `G` and extract their friends list. Then `all_dislike` is used to check if each member of the friends list "dislikes" `X`.

```
outcast(G, X) :-
    member(G, X, XFs),
    all_dislike(G, XFs, X).
```

```
all_dislike(_, [], _).
all_dislike(G, [H|T], Y) :-
    dislikes(G, H, Y),
    all_dislike(G, T, Y).
```

2.3 Friendly

`friendly(G, X)` is used to determine if `X`'s friends list contains every person who likes `X`. This is done with two clauses, `find_friends` and `likes_all`. `find_friends(G, G, X, [], Fs` is used to extract a list of people who like `X`. Its arguments are two instances of a graph, the person `X`, an empty accumulator which holds the people who like `X` and the resultant list of people. It does this by extracting each person `Y` and their friends list from the graph by using `equal`. It then uses `elem` to check if `X` is a member of their friends list. If it is, then `find_friends` is called again with the tail of the graph and with `Y` added to the accumulator. If not, `not_in` is used to check that they are not in the friends list and`find_friends` is called on the tail but with the accumulator unchanged. Once all elements in the graph have been checked, `find_friends` copies the accumulator to the result.

`likes_all(G, X, Fs)` is a predicate similar to `all_likes` except each member of the friends list is checked to see if `X` likes them. Finally, `friendly` used `find_friends` to extract the people who like `X` and then use `likes_all` to check if he likes all of them.

```
find_friends(_, [], _, Acc, Acc).
find_friends(G, [H|T], X, Acc, Res) :-
    equal(H, person(Y, YFs)),
    elem(X, YFs),
    find_friends(G, T, X, [Y|Acc], Res).
find_friends(G, [H|T], X, Acc, Res) :-
    equal(H, person(_, YFs)),
    not_in(G, X, YFs),
    find_friends(G, T, X, Acc, Res).
```

```
friendly(G, X) :-
    find_friends(G, G, X, [], Fs),
    likes_all(G, X, Fs).

likes_all(_, _, []).
likes_all(G, X, [H|T]) :-
    likes(G, X, H),
    likes_all(G, X, T).
```

2.4 Hostile

`hostile(G, X)` behaves in an opposite way to `friendly`, it checks whether a person `X` dislikes all people who are friends with them. Just like `friendly`, it uses `find_friends` to extract the list of people who like `X` and then `dislikes_all` to check if `X` dislikes all of those people.

```
hostile(G, X) :-
  find_friends(G, G, X, [], Fs),
  dislikes_all(G, X, Fs).
```

```
dislikes_all(_, _, []).
dislikes_all(G, X, [H|T]) :-
  dislikes(G, X, H),
  dislikes_all(G, X, T).
```

2.5 Testing

All local predicates are tested through the use of the query `-? g1(G1), g2(G2), local_test(G1,G2).`. This checks that all local predicates pass on correct inputs for both the superhero graph and the alias graph. It also checks that erroneous inputs fail on the graphs. The test predicates can be seen here. All tests pass.

```
local_test(G1,G2) :-
  pop_test(G1,G2),
  \+(pop_err(G1,G2)),
  out_test(G1,G2),
  \+(out_err(G1,G2)),
  friendly_test(G1,G2),
  \+(friendly_err(G1,G2)),
  host_test(G1,G2),
  \+(host_err(G1,G2)),
```

```
pop_test(G1,G2) :-
  popular(G1, kara),
  popular(G2, supergirl).

pop_err(G1,G2) :-
  popular(G1, bruce);
  popular(G1, clark);
  popular(G2, green_arrow).
```

```
out_test(G1,G2) :-
  outcast(G1, bruce),
  outcast(G1, oliver),
  outcast(G2, batman).

out_err(G1,G2) :-
  outcast(G1, kara);
  outcast(G1, clark);
  outcast(G2, flash).
```

```
friendly_test(G1,G2) :-
  friendly(G1, barry),
  friendly(G2, flash).

friendly_err(G1,G2) :-
  friendly(G1, clark);
  friendly(G1, kara).
```

```
host_test(G1,G2) :-
  hostile(G1, oliver),
  hostile(G1, bruce),
  hostile(G2, green_arrow).

host_err(G1,G2) :-
  hostile(G1, kara);
  hostile(G1, barry);
  hostile(G2, superman).
```

3 Global Relations

The global predicates try to find paths between two people, `admires` succeeds if there is one, `indifferent` would fail in that case.

If we want to implement this as a search algorithm (like Dijkstra) we need a way to mark nodes visited in order not to end up in cycles. This can be implemented in Prolog by carrying an agenda of people yet to check. This agenda is passed as an additional variable `Todo` and after checking one person this is extracted. Therefore we need a wrapper function which calls the recursive predicates with a list of all names (c.f. helper function `names/2`) and passes it to the recursive version of the predicates, respectively.

Also note in order for admires and indifferent and X = Y to be mutually exclusive we ensured that X and Y are different in admires and indifferent. Otherwise, e.g., Oliver would admire himself.

3.1 admires

A person A admires another one C if either A likes C directly or there is a person B that A likes and in turn admires C. This is pretty straight-forwardly translated into Prolog by either trying to call `likes` or recursively call `admires1`. For this, we need to pass a list `Todo1` (see above) which is the previous `Todo` with the current person (more exact, the person's name) extracted, i.e. `select1(X, Todo, Todo1)` does the job. Once there is no-one in the agenda anymore the predicate fails.

```
admires(G, X, Y) :-
  different(G, X, Y),
  names(G, Todo),
  admires1(G, Todo, X, Y).

admires1(G, _, X, Y) :-
  likes(G, X, Y).
admires1(G, Todo, X, Y) :-
  elem(Z, Todo),
  likes(G, X, Z),
```

```
  select1(Z, Todo, Todo1),
  admires1(G, Todo1, Z, Y).
```

3.2 indifferent

Indifferent is similar, but the other way around. A person X must not like another Y (c.f. helper predicate `not_likes/3`, calls `not_in` for checking if someone is not in the list), and neither may his friends like Y.

```
indifferent(G, X, Y) :-
  different(G, X, Y),
  names(G, Todo),
  indifferent(G, Todo, X, Y).

indifferent(_, [], _, _).
indifferent(G, Todo, X, Y) :-
  not_likes(G, X, Y),                    % X does not like Y himself
  friends(G, X, XFs),                    % and X's friends are also ...
  select1(X, Todo, Todo1),               % (exclude this person from todos)
  filter(G, XFs, Todo1, FsTodo),         % (filter the friends who are in todos)
  all_indifferent(G, Todo1, FsTodo, Y). % ... all indifferent to Y
```

In order to check a list of friends we first filter it for the ones that are still in `Todos`. For that we have a helper predicate `filter` which takes two lists and "returns" (unifies its third argument with) the intersection of both. This is easily implemented by taking the first list and checking for each element if it also occurrs (`elem`) in the second. In that case we include it in an accumulator. (Accumulators are a clever design pattern that allows us to build up a list in an intuitive way and in the base case copy it into the final result.) In the other case, if an element from the first list is `not_in` the second (note that negation is again relative to the graph!), we just do not copy the element into the accumulator.

```
filter(G, L1, L2, Intersect) :-
  filter1(G, L1, L2, [], Intersect).

filter1(_, [], _, Acc, Acc).
filter1(G, [H|T], L2, Acc, Res) :-
  elem(H, L2),
  filter1(G, T, L2, [H|Acc], Res).
filter1(G, [H|T], L2, Acc, Res) :-
  not_in(G, H, L2),
  filter1(G, T, L2, Acc, Res).
```

`all_indifferent` then takes a list of the filtered names, iterates over it and checks if each element is indifferent. It is important that for each person we pass the current agenda when checking indifference of all its friends recursively. If this agenda is finally empty, `indifferent` reaches its base case, there is nobody left to check and the original person X is indeed indifferent from the target Y.

```
all_indifferent(_, _, [], _).
all_indifferent(G, Todo, [H|T], Y) :-
  indifferent(G, Todo, H, Y),
  all_indifferent(G, Todo, T, Y).
```

3.3 Testing

Testing for the global predicates are performed by the query `-? g1(G1), g2(G2), global_test(G1,G2).` which test the goals on a variety of inputs and ensure that erroneous inputs fail. All tests pass, and they can be seen below.

```
global_test(G1,G2) :-
  adm_test(G1,G2),
  not(adm_err(G1,G2)),
  indif_test(G1,G2),
  not(indif_err(G1,G2)).

adm_test(G1,G2) :-
  admires(G1, bruce, kara),
  admires(G1, clark, barry),
  admires(G1, oliver, clark),
  admires(G2, supergirl, green_arrow).

adm_err(G1,G2) :-
  admires(G1, kara, bruce);
  admires(G1, oliver, bruce);
  admires(G2, superman, batman).

indif_test(G1,G2) :-
  indifferent(G1, kara, bruce),
  indifferent(G1, oliver, bruce),
  indifferent(G2, superman, batman).
```

```
indif_err(G1,G2) :-
  indifferent(G1, bruce, kara);
  indifferent(G1, clarke, barry);
  indifferent(G1, oliver, clark);
  indifferent(G2, supergirl, green_arrow).
```

4 Whole world properties

The `same_world` predicate is trying to match two worlds and, if possible, tries to find a mapping between the members of both. In other words, we have an constraint satisfaction problem at hand, consisting of many identity constraints between people of the graphs. Solving this directly using backtracking is a non-trivial problem because it requires carefull checking of the mapping at each point, that is still to be created!

Complexity theory however tells us, that checking a given solution is always easier than the primary problem. So we apply a so-called generate-test approach in which we alternately generate an arbitrary mapping and then test if it fulfills the constraints.

```
same_world(G1, G2, Map) :-
  same_length(G1, G2), % quick sanity check first
  names(G1, N1),       % get the names for convenience
  names(G2, N2),
  % stupid generate and test approach follows ...
  generate_mapping(N1, N2, Map),
  test_mapping(G1, G2, Map).
```

Generating all combinations of people (from world 1 and world 2) is natural in Prolog. Again `select1` can help here, because if called with a free variable at the first and a list at the second position, it will extract every possible element by backtracking and bind it to the free variable. Thus, we construct pairs by fixing one element from the first list and selecting one from the second:

```
generate_mapping([], _, []).
generate_mapping([H|T], N2, [(H,N)|Map]) :-
  select1(N, N2, N2R),
  generate_mapping(T, N2R, Map).
```

That being done, we have a mapping of names that we can check - if it works, we are lucky, if it doesn't Prolog will generate a new one and if it tried all we are sure there is none. Should the mapping work we automatically have it as an output.

Testing is now done by checking if for each person in world 1 the corresponding person in world two has the same friends, meaning they have to correspond in number and with regard to the mapping. Thus we need two additional helper predicates, one checking if there are equally many friends (`same_length` applied to two lists) and another `same_friends` which is given the mapping and reviews the identities. The former is easy, it just desroys both lists simultaneously and succeeds if that takes the same time. Checking for identity between two lists is also trivial, it involves a lookup in the mapping to get the complementary name of the person and checks if this is `elem` of the other list of friends. This way we ensure that the order of friends in the adjacency list does not matter but at the same time there cannot be more people in F2 than in F1, because their length is the same.

```
test_mapping([], _, _).
test_mapping([H|T], G2, Map) :-
  equal(H, person(N1, F1)),
  elem((N1,N2), Map),   % lookup the corresponding N2
  friends(G2, N2, F2),  % and his friends;
  same_length(F1, F2),  % necessary, not only sanity check because
  same_friends(F1, F2, Map), % this checks only if F1 subset_of F2
  test_mapping(T, G2, Map).

same_friends([], _, _).
same_friends([N1|T], F2, Map) :-
  elem((N1,N2), Map),            % lookup in mapping
  elem(N2, F2),                  % check if N2 is in F2
  same_friends(T, F2, Map).

same_length([], []).
same_length([_|T1], [_|T2]) :-
  same_length(T1, T2).
```

4.1 Testing

The `same_world` predicate can be tested simply by running the query `-? g1(G1), g2(G2), a1(A1), whole_test(G1,G2,A1).`. This query checks that G1 and G2 are the same world and if true results in a list of each person paired with their pair on the other world. This is checking the result list against an array of each superhero's real name paired with their alias. We also note that the predicate is symmetric. The test passes and can be seen here.

```
a1([(kara,supergirl),(bruce,batman),(barry,flash),(clark,superman),(oliver,green_arrow)]).

whole_test(G1,G2,A1) :-
  same_world(G1,G2,A),
  same_world(G2,G1,A),
  equal(A,A1).
```

5 Conclusion

The testing we did might not be exhaustive (since only example based), but together with the Online-TA we are confident we haven't missed any major bugs. The Online-TA is also testing edge cases such as empty graphs which requires some predicates to ensure its arguments are actually members of the graph. This was irritating at first but could easily be debugged. We may also mention that there are some queries spitting out the same solution multiple times. For instance, `g1(G), admires(G, X, kara).` does so, which in this case is due to the way the algorithm constructs the chain to super-popular kara. That we can live with.