AP Assignment 3: Analysing FacedIn

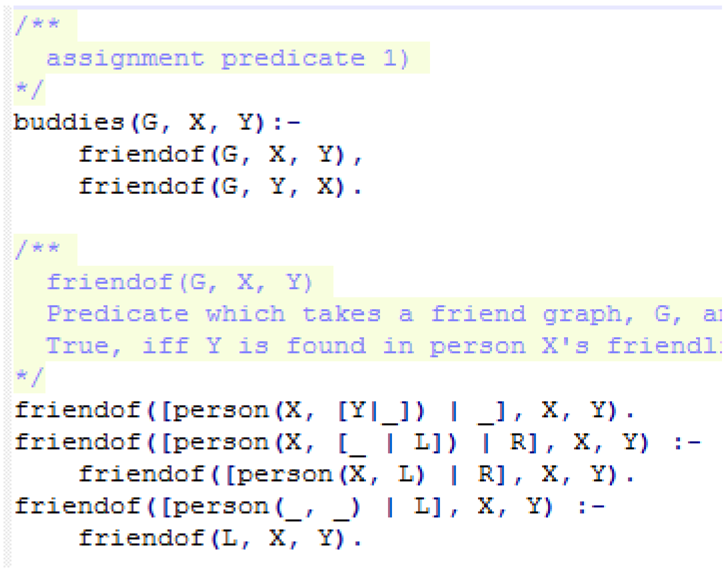
By Simon Gustav Cordes

This report explains the code in my solution to the assignment as well as showing test results and how to run these test.

Code Explanation

This section examines and explains the code in my solution to the assignment

Buddies



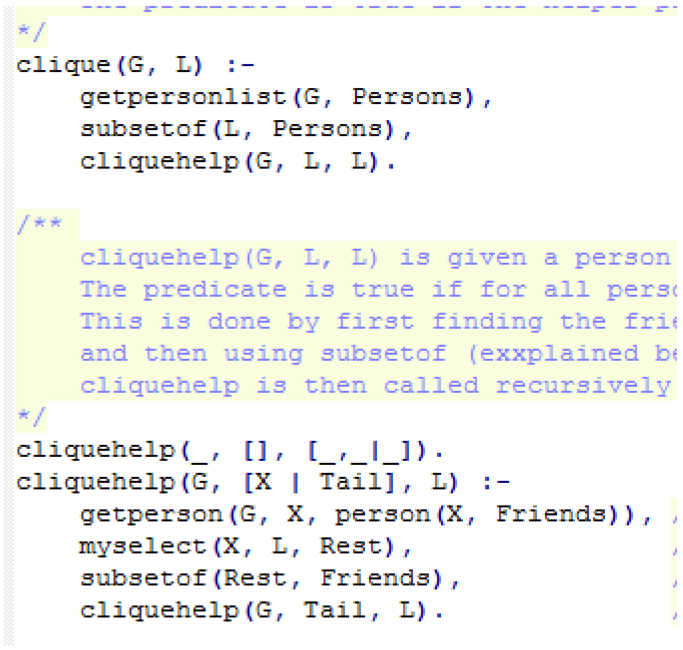
```
/**
  assignment predicate 1)
*/
buddies(G, X, Y):-
    friendof(G, X, Y),
    friendof(G, Y, X).

/**
  friendof(G, X, Y)
  Predicate which takes a friend graph, G, a
  True, iff Y is found in person X's friendl
*/
friendof([person(X, [Y|_]) | _], X, Y).
friendof([person(X, [_ | L]) | R], X, Y) :-
    friendof([person(X, L) | R], X, Y).
friendof([person(_, _) | L], X, Y) :-
    friendof(L, X, Y).
```

The buddies(G, X, Y) predicate is true whenever the two predicates friendsof(G, X, Y) and friendof(G, Y, X) is true. G is a friend graph and X and Y is names of persons in the graph.

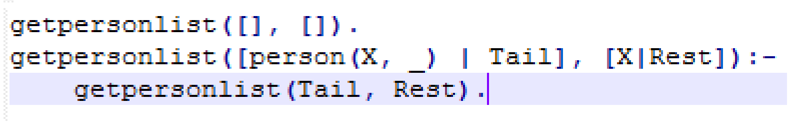
Friendof is a predicate which is true if a person Y (where Y is the name of the person) is in some person X's (X is the name of the other person) friends list in some friend graph, G. In essence we see if X is found in G. If X is found in G we see if Y is found in the friends list of X. Return true, if so.

We see in buddies that if Y is in X's friends list and X is in Y's friends list that they are buddies.

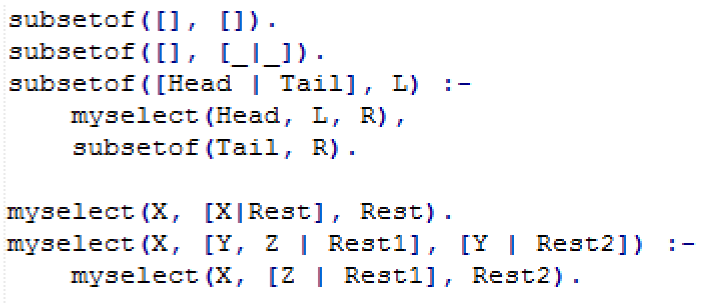
Clique



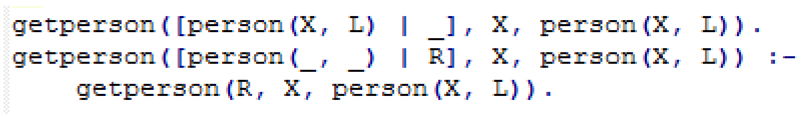
```
*/
clique(G, L) :-
    getpersonlist(G, Persons),
    subsetof(L, Persons),
    cliquehelp(G, L, L).

/**
    cliquehelp(G, L, L) is given a person
    The predicate is true if for all perso
    This is done by first finding the frie
    and then using subsetof (exxplained be
    cliquehelp is then called recursively
*/
cliquehelp(_, [], [_,_|_]).
cliquehelp(G, [X | Tail], L) :-
    getperson(G, X, person(X, Friends)),
    myselect(X, L, Rest),
    subsetof(Rest, Friends),
    cliquehelp(G, Tail, L).
```



```
getpersonlist([], []).
getpersonlist([person(X, _) | Tail], [X|Rest]):-
    getpersonlist(Tail, Rest).
```



```
subsetof([], []).
subsetof([], [_|_]).
subsetof([Head | Tail], L) :-
    myselect(Head, L, R),
    subsetof(Tail, R).

myselect(X, [X|Rest], Rest).
myselect(X, [Y, Z | Rest1], [Y | Rest2]) :-
    myselect(X, [Z | Rest1], Rest2).
```



```
getperson([person(X, L) | _], X, person(X, L)).
getperson([person(_, _) | R], X, person(X, L)) :-
    getperson(R, X, person(X, L)).
```

The predicate Clique is true when L represents a clique in the friend graph G.

Here we first use getpersonlist to get a list of the names, Persons, of the people in G, and then see if L is a subset of Persons (We assume that G is a well-formed graph, thus a clique list with duplicates will be caught here). This is done with the subsetof(L1, L2) predicate. We have the base case the if L1 and L2 are empty lists the predicate is true. The same applies if L1 is empty. If the lists are nonempty we remove the first element of L1 from L2 and call subset of on the tail of L1 and the remainder of L2. This yields true if L1 is a subset of L2.

If L is a subset of Persons we call cliquehelp which gets the person X from G (to be exact we get the friends list, use myselect to remove X from L, then see if the remainder of this list is a subset of the friends of X. If so, we see call cliquehelp recursively with the rest of the list L.

Admirer

```
admirer(G, X) :-
    getpersonlist(G, L),
    myselect(X, L, Rest),
    pathToAll(G, X, Rest).

pathToAll(_, _, []).
pathToAll(G, X, [Head | Tail]) :-
    pathToPerson(G, X, Head),
    pathToAll(G, X, Tail).

pathToPerson(G, X, Y) :-
    friendof(G, X, Y).
pathToPerson(G, X, Y) :-
    friendof(G, X, Z),
    getperson(G, X, P),
    myselect(P, G, Rest),
    pathToPerson(Rest, Z, Y).
```

Admirer is true when there is a path from X to everyone else in G. We assume that G is a well-formed graph

We first get the list, L ,of the names of people in the graph G. We then remove X from this list and use pathToAll to see if there is a path from X to all other people in the list.

pathToAll(G, X, L) uses pathToPerson to see if there is a path from X in to some person in G.

the predicate pathToPerson(G, X, Y) is of course true if Y is in X's friends list. If the case differs we find some friend of X, Z, and uge getperson and myselect to remove X from G (so we don't go visit the same node twice). We then use pathToPerson to see if there is a path from Z to Y in the rest of the graph.

Logically it makes sense that this is correct since at some point we will reach Y if there is a path. If there is no path then we will never run in circles as there is a limited number of paths around the graph visiting every node at most once.

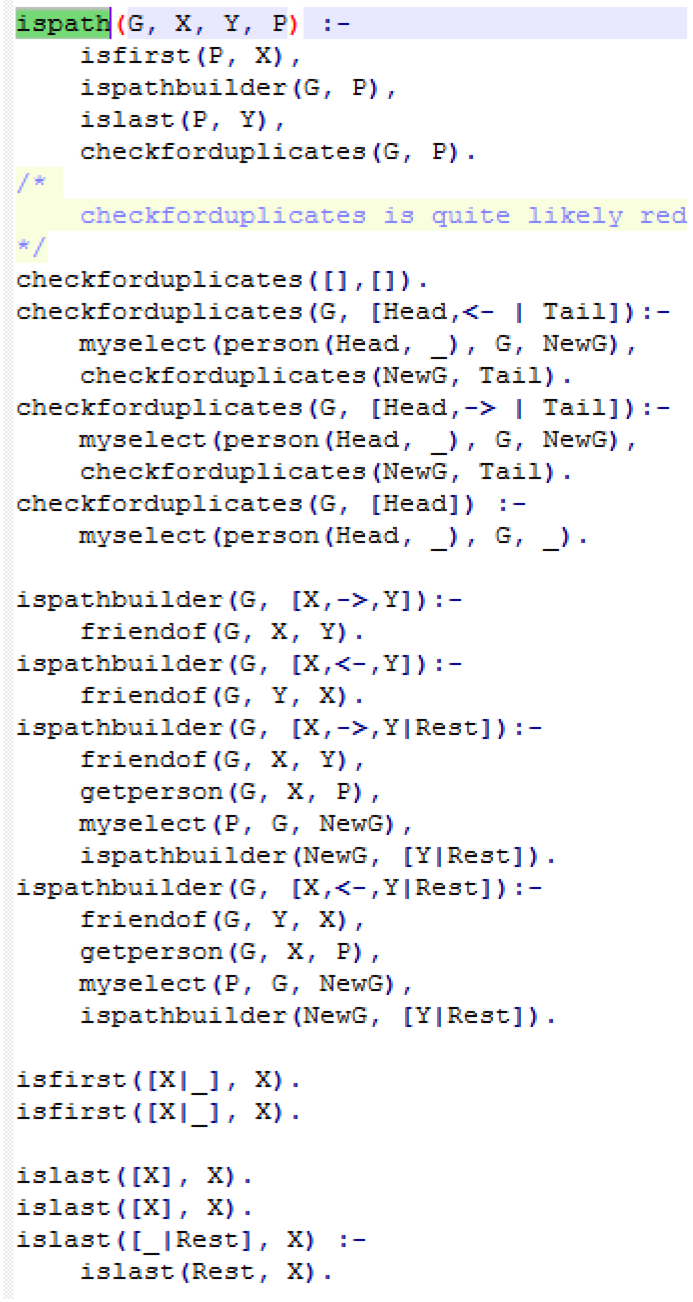
Idol

```
idol(G, X) :-
    getpersonlist(G, L),
    myselect(X, L, Rest),
    pathFromAll(G, X, Rest).

pathFromAll(_, _, []).
pathFromAll(G, X, [Head| Tail]) :-
    pathToPerson(G, Head, X),
    pathFromAll(G, X, Tail).
```

Idol works opposite of Admirer. Idol uses the predicate pathFromAll instead which checks if there is a path from every other person to X.

This is also correct since there is a limited number of paths visiting all nodes at most once, every person has a limited friends list and there is a limited number of different starting points.

Ispath



```
ispath(G, X, Y, P) :-
    isfirst(P, X),
    ispathbuilder(G, P),
    islast(P, Y),
    checkforduplicates(G, P).
/*
    checkforduplicates is quite likely red
*/
checkforduplicates([],[]).
checkforduplicates(G, [Head,<- | Tail]):-
    myselect(person(Head, _), G, NewG),
    checkforduplicates(NewG, Tail).
checkforduplicates(G, [Head,-> | Tail]):-
    myselect(person(Head, _), G, NewG),
    checkforduplicates(NewG, Tail).
checkforduplicates(G, [Head]) :-
    myselect(person(Head, _), G, _).

ispathbuilder(G, [X,->,Y]):-
    friendof(G, X, Y).
ispathbuilder(G, [X,<-,Y]):-
    friendof(G, Y, X).
ispathbuilder(G, [X,->,Y|Rest]):-
    friendof(G, X, Y),
    getperson(G, X, P),
    myselect(P, G, NewG),
    ispathbuilder(NewG, [Y|Rest]).
ispathbuilder(G, [X,<-,Y|Rest]):-
    friendof(G, Y, X),
    getperson(G, X, P),
    myselect(P, G, NewG),
    ispathbuilder(NewG, [Y|Rest]).

isfirst([X|_], X).
isfirst([X|_], X).

islast([X], X).
islast([X], X).
islast([_|Rest], X) :-
    islast(Rest, X).
```

Ispath(G, X, Y, P) attemps to find a path to from X to Y.

For this, I have designed two predicates, isfirst(L, X) which is true if X is first in L and islast(L, X) which is true if X is last in L.

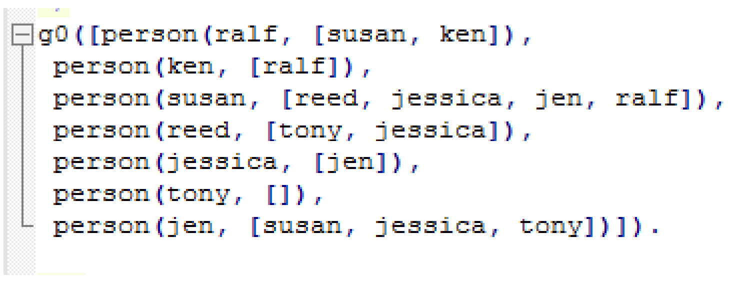
The predicate ispath first assures that X is first in P, it then attemps to build a path in G. When this is done, it then assures if Y is past in P. At first I had islast before ispathbuilder which made the predicate loop forever (not good).

Ispathbuilder(G, P) where P is some list [X,A_i,Y|Rest] sees if X or Y is in the others friend list, then removes X from G and calls ispathbuilder with the new graph and the rest of P. This will not loop forever since there is a limited number of paths around G when we only visit each vertex at most once.

I have an additional check, checkforduplicates", which checks if there are duplicates in P, this is, though, as far as I can see redundant.

Tests

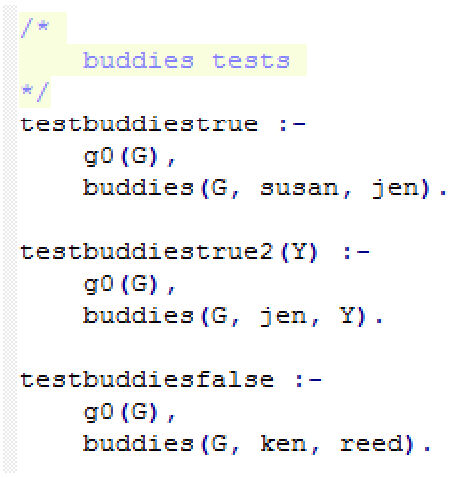
We run the tests on the graph given in the exercise. In particular, this:



```
g0([person(ralf, [susan, ken]),
 person(ken, [ralf]),
 person(susan, [reed, jessica, jen, ralf]),
 person(reed, [tony, jessica]),
 person(jessica, [jen]),
 person(tony, []),
 person(jen, [susan, jessica, tony])]).
```

Buddies tests

The predicate buddies was tested with three cases, which are the following:



```
/*
    buddies tests
*/
testbuddiestrue :-
    g0(G),
    buddies(G, susan, jen).

testbuddiestrue2(Y) :-
    g0(G),
    buddies(G, jen, Y).

testbuddiesfalse :-
    g0(G),
    buddies(G, ken, reed).
```

Results

The output from the tests is the following:

```
15 ?- testbuddiestrue.
true ;
false.

16 ?- testbuddiestrue2(Y).
Y = susan ;
Y = jessica ;
false.

17 ?- testbuddiesfalse.
false.
```

Conclusion

These results show that the buddies predicate is correct and does not run forever.

Clique tests

The predicate clique was tested in three different cases with the following tests:

```
/*
    clique tests
*/

testcliquetrue :-
    g0(G),
    clique(G, [susan, jen]).

testcliquetrue2(L) :-
    g0(G),
    clique(G, L).

testcliquefalse :-
    g0(G),
    clique(G, [ralf, susan, ken]).
```

Results

The output from the tests is as follows:

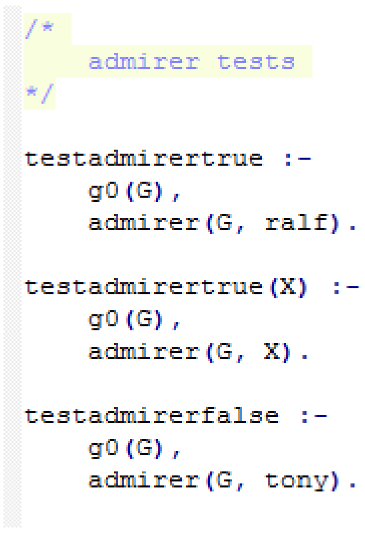
```
18 ?- testcliquetrue.
true ;
false.

19 ?- testcliquetrue2(L).
L = [ralf, ken] ;
L = [ralf, susan] ;
L = [ken, ralf] ;
L = [susan, ralf] ;
L = [susan, jen] ;
L = [jessica, jen] ;
L = [jen, susan] ;
L = [jen, jessica] ;
false.

20 ?- testcliquefalse.
false.
```

Conclusion

These test results also show that clique is working as intended and that we do not have a situation where the program runs forever.

Admirer tests

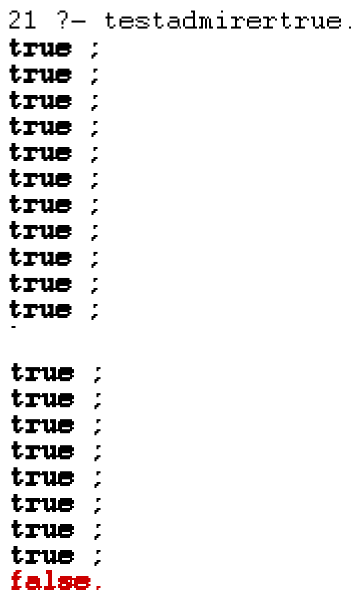
The admirer predicate was tested with the following three test cases:



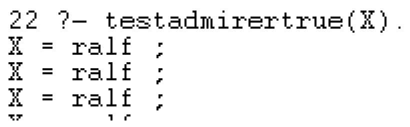
```
/*
    admirer tests
*/

testadmirertrue :-
    g0(G),
    admirer(G, ralf).

testadmirertrue(X) :-
    g0(G),
    admirer(G, X).

testadmirerfalse :-
    g0(G),
    admirer(G, tony).
```

Results

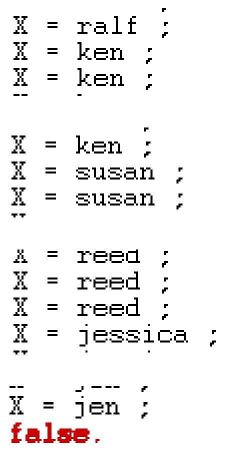
The output from the tests was as follows:



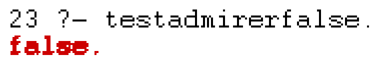
```
21 ?- testadmirertrue.
true ;
true ;
true ;
true ;
true ;
true ;
true ;
true ;
true ;
true ;
true ;

true ;
true ;
true ;
true ;
true ;
true ;
true ;
true ;
false.
```



```
22 ?- testadmirertrue(X).
X = ralf ;
X = ralf ;
X = ralf ;
```



```
X = ralf ;
X = ken ;
X = ken ;

X = ken ;
X = susan ;
X = susan ;

X = reed ;
X = reed ;
X = reed ;
X = jessica ;

X = jen ;
false.
```



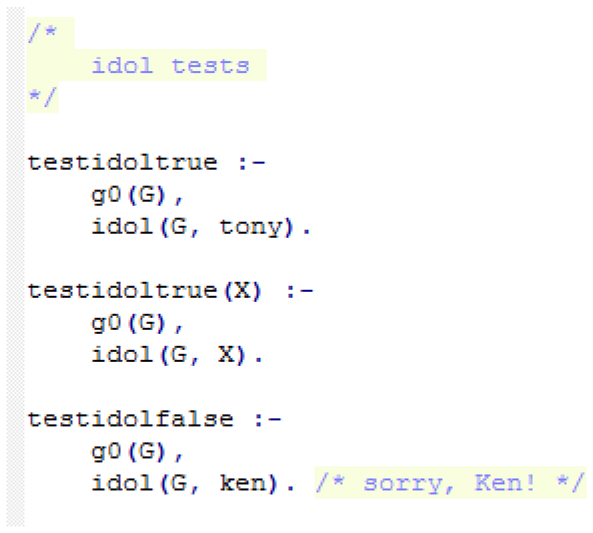
```
23 ?- testadmirerfalse.
false.
```

Conclusion

Even though we get the same result many times, the program doesn't run forever. One reason could be that there are several ways around the graph from each member. But, in conclusion, the predicate works as intended. In general, at least.

Idol tests

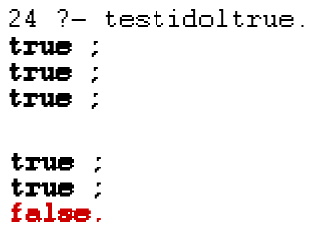
The idol predicate was tested with the following three test cases:



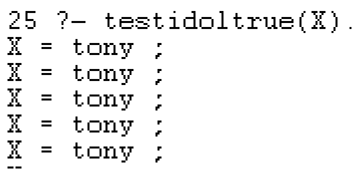
```
/*
    idol tests
*/

testidoltrue :-
    g0(G),
    idol(G, tony).

testidoltrue(X) :-
    g0(G),
    idol(G, X).

testidolfalse :-
    g0(G),
    idol(G, ken). /* sorry, Ken! */
```

Results

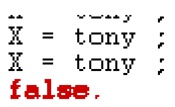
The results of the tests is as follows



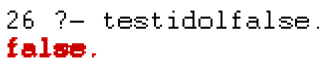
```
24 ?- testidoltrue.
true ;
true ;
true ;

true ;
true ;
false.
```



```
25 ?- testidoltrue(X).
X = tony ;
X = tony ;
X = tony ;
X = tony ;
X = tony ;
```



```
X = tony ;
X = tony ;
false.
```



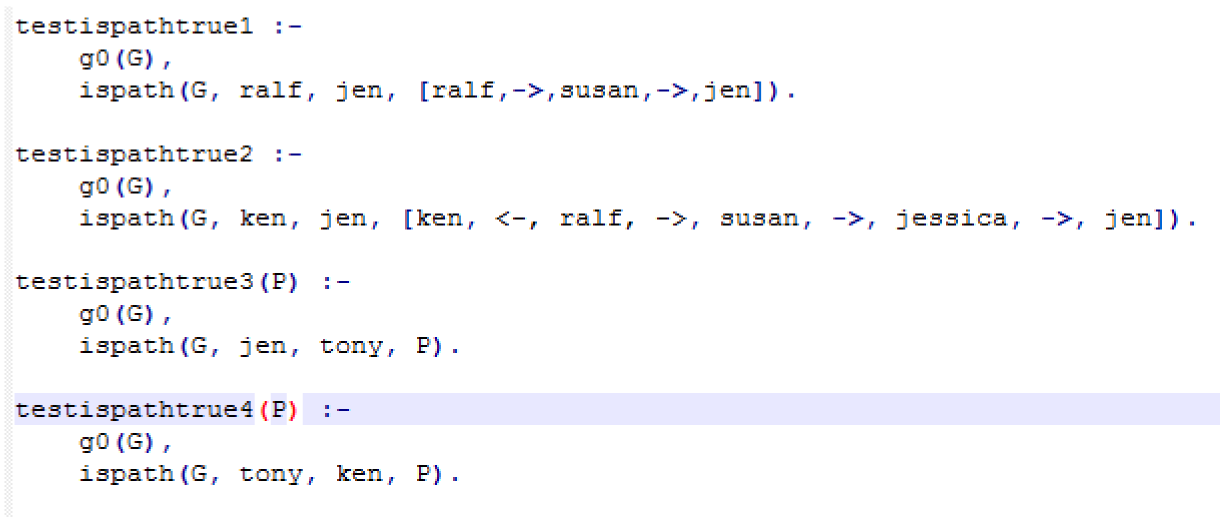
```
26 ?- testidolfalse.
false.
```

Conclusion

The results show that, even though we get the same output several times, the program doesn’t go on infinitely. It also returns the correct result every time and stops when there are no more paths to explore.

Ispath Tests

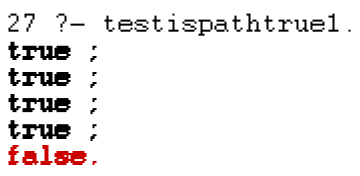
The ispath predicate was tested with the following tests:



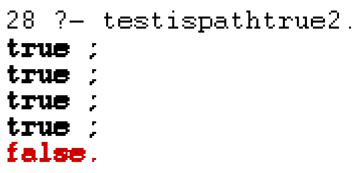
```
testispathtrue1 :-
    g0(G),
    ispath(G, ralf, jen, [ralf,->,susan,->,jen]).

testispathtrue2 :-
    g0(G),
    ispath(G, ken, jen, [ken, <-, ralf, ->, susan, ->, jessica, ->, jen]).

testispathtrue3(P) :-
    g0(G),
    ispath(G, jen, tony, P).

testispathtrue4(P) :-
    g0(G),
    ispath(G, tony, ken, P).
```

Results

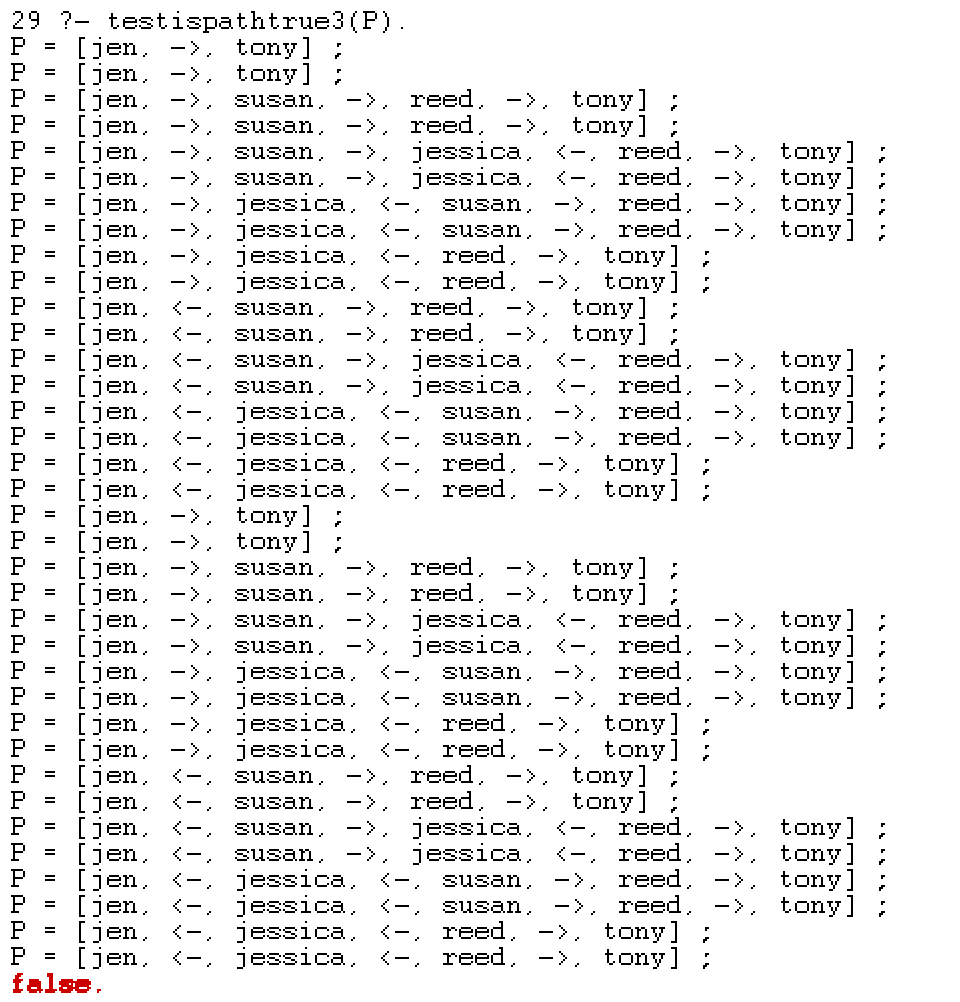
The results were as follows:



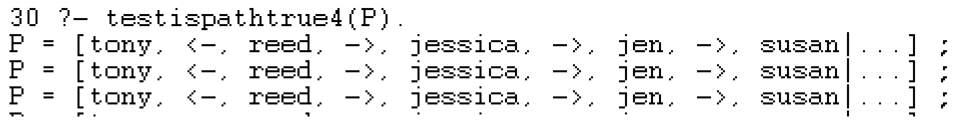
```
27 ?- testispathtrue1.
true ;
true ;
true ;
true ;
false.
```



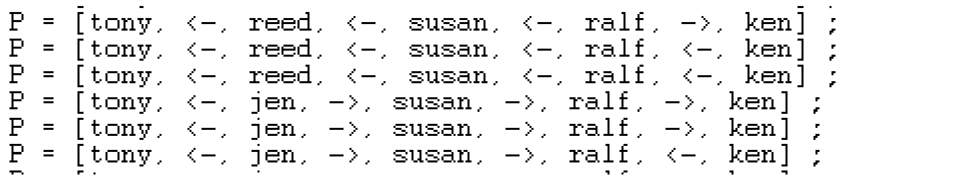
```
28 ?- testispathtrue2.
true ;
true ;
true ;
true ;
false.
```



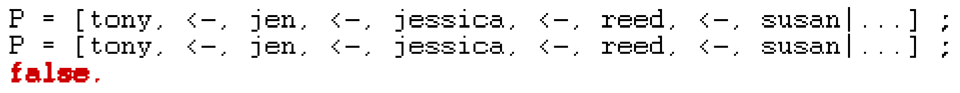
```
29 ?- testispathtrue3(P).
P = [jen, ->, tony] ;
P = [jen, ->, tony] ;
P = [jen, ->, susan, ->, reed, ->, tony] ;
P = [jen, ->, susan, ->, reed, ->, tony] ;
P = [jen, ->, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, ->, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, jessica, <-, reed, ->, tony] ;
P = [jen, <-, jessica, <-, reed, ->, tony] ;
P = [jen, ->, tony] ;
P = [jen, ->, tony] ;
P = [jen, ->, susan, ->, reed, ->, tony] ;
P = [jen, ->, susan, ->, reed, ->, tony] ;
P = [jen, ->, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, ->, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, ->, jessica, <-, reed, ->, tony] ;
P = [jen, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, susan, ->, jessica, <-, reed, ->, tony] ;
P = [jen, <-, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, jessica, <-, susan, ->, reed, ->, tony] ;
P = [jen, <-, jessica, <-, reed, ->, tony] ;
P = [jen, <-, jessica, <-, reed, ->, tony] ;
false.
```



```
30 ?- testispathtrue4(P).
P = [tony, <-, reed, ->, jessica, ->, jen, ->, susan|...] ;
P = [tony, <-, reed, ->, jessica, ->, jen, ->, susan|...] ;
P = [tony, <-, reed, ->, jessica, ->, jen, ->, susan|...] ;
```



```
P = [tony, <-, reed, <-, susan, <-, ralf, ->, ken] ;
P = [tony, <-, reed, <-, susan, <-, ralf, <-, ken] ;
P = [tony, <-, reed, <-, susan, <-, ralf, <-, ken] ;
P = [tony, <-, jen, ->, susan, ->, ralf, ->, ken] ;
P = [tony, <-, jen, ->, susan, ->, ralf, ->, ken] ;
P = [tony, <-, jen, ->, susan, ->, ralf, <-, ken] ;
```



```
P = [tony, <-, jen, <-, jessica, <-, reed, <-, susan|...] ;
P = [tony, <-, jen, <-, jessica, <-, reed, <-, susan|...] ;
false.
```

Conclusion

The predicate at least gives the correct result. It could probably (certainly) be tweaked to return far fewer results, especially in the first cases where we give it an exact solution.

At the very least the predicate does not run forever, and we get correct results.

How to run the code:

The code can be run by consulting the facedin.pl file in your favorite prolog environment, and the tests can be run by consulting the tests.pl module and running the tests as in the examples above. I have not supplied a “run all” option as some tests yield false on purpose.