Introduction to Artificial Intelligence (CS-487)
Assignment #4

Due on Jan 6, 2023

Professor I. Tsamardinos

University of Crete
Department of Computer Science

Nikolaos Kougioulis (ID 1285)

December 28, 2022

Many problems in the field of Artificial Intelligence are reduced to finding a path on a graph. We can represent a directed graph in Prolog (like the one in Figure 1a) using the following facts:

```
connect(a, b).
connect(b, c).
connect(b, d).
connect(e, d).
   etc ...
```

A simple way to represent a non-directed graph (like the one in Figure 2) would be to double the number of facts to state all the reverse connections (e.g. `connect(b, a)`). A better way is to define the following two rules:

```
interconnected(X, Y):- connect(X, Y).
interconnected(X, Y):- connect(Y, X).
```

where the predicate `interconnected/2` represents the bi-directional relationship between the nodes.

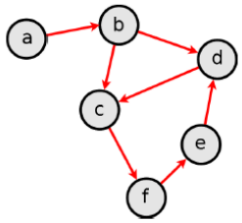

Figure 1: A directed graph.

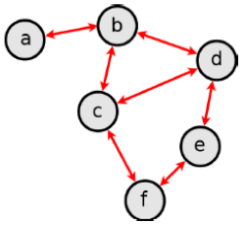

Figure 2: A non-directed graph.

Exercise 1

Could we use just one rule like the following (instead of two rules)?

```
interconnected(X, Y):- interconnected(Y, X).
```

Explain your answer. Are there situations where the above rule would work and other situations that would be problematic?

Solution:

Using just the above symmetric rule,
`interconnected(X, Y):- interconnected(Y, X).`

instead of two (`interconnected(X, Y):- connect(X, Y), connect(Y, X).`), is mostly problematic and cumbersome compared to the rule used at the beginning of the text.

Consider a graph with two nodes, a and b. If `connect(a, b).` and `interconnected(a, b).` exist in our Knowledge Base, then by performing the query `?- interconnected(a,b)` we obviously obtain `true`, as

```
1 connect(a, b).
2
3 interconnected(a, b).
4 interconnected(X, Y):- interconnected(Y, X).
5
```

Figure 3: Exercise 1: A knowledge base of rules in the SWISH environment of SWI-Prolog.[a]

[a] https://swish.swi-prolog.org/

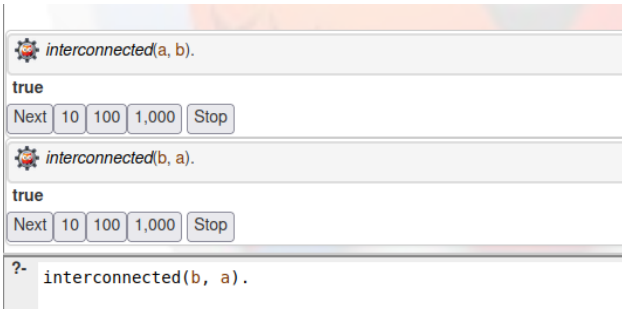

Figure 4: Successfully querying `?- interconnected(a, b).` and `?- interconnected(b, a).`

```
1 connect(a, b).
2 connect(b, a).
3
4 interconnected(X, Y):- interconnected(Y, X).
5
6
```

Figure 5: Exercise 1: Altered Knowledge Base.

Figure 6: Exercise 1: Example of unsuccessful KB Query.

it already exists in our Knowledge Base. Querying `?- interconnected(b,a)` we also obtain `true`, as it is derived using Modus Ponens from the `interconnected` rule (Figures 3 & 4).

Now consider the case our KB contains only the rules `connect(a,b).`, `connect(b,a).`, and `interconnected(X, Y):- interconnected(Y, X).` (Figures 5 & 6). From the first two rules, it is evident that if a is connected to b and b is connected to a means `interconnected(a,b)` and `interconnected(b,a)` yield `true`. This however is not entailed and logically derived from our KB, so such queries would fail.

```
connect(a, b).
connect(b, c).
connect(b, d).
connect(e, d).
connect(d, c).
connect(c, f).
connect(f, e).
interconnected(X, Y):- connect(X, Y); connect(Y, X).

interconnected(a,b).
interconnected(b,d).
interconnected(e,d).
interconnected(d,c).
interconnected(c,f).
interconnected(f,e).

:- style_check(-singleton).

%exists_path/2
exists_path(StartNode, EndNode).
exists_path(StartNode, EndNode) :- interconnected(StartNode, NextNode), exists_path(NextNode, EndNode).
```

Figure 7: KB and rules in Exercise 2.

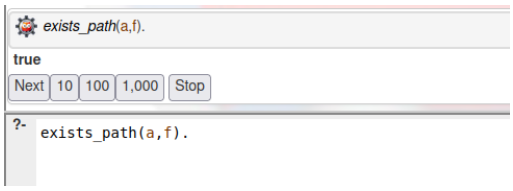

Figure 8: `exists_path` query in Exercise 2.

Exercise 2

Define a predicate `exists_path/2` that implements a simple Depth-First Search. This predicate just confirms the existence of a connection between two nodes. For example, the following question would answer "true":

```
?- exists_path(a, f).
true
```

Solution

The solution is based on the recursive definition of the path from Graph Theory as follows:

```
%exists_path/2
exists_path(StartNode, EndNode).
exists_path(StartNode, EndNode) :- interconnected(StartNode, NextNode),
                                   exists_path(NextNode, EndNode).

```

```
connect(a, b).
connect(b, c).
connect(b, d).
connect(e, d).
connect(d, c).
connect(c, f).
connect(f, e).
interconnected(X, Y):- connect(X, Y); connect(Y, X).

interconnected(a,b).
interconnected(b,d).
interconnected(e,d).
interconnected(d,c).
interconnected(c,f).
interconnected(f,e).

:- style_check(-singleton).

%exists_path/2
exists_path(StartNode, EndNode).
exists_path(StartNode, EndNode) :- interconnected(StartNode, NextNode), exists_path(NextNode, EndNode).

path(StartNode, EndNode, Route) :- path(StartNode, EndNode, [], Route).

path(StartNode, StartNode, _, [StartNode]).

path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes]) :-
    \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is a member of VisitedNodes list
    dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are different
    interconnected(StartNode, Node),
    path(Node, EndNode, [StartNode|VisitedNodes], Nodes).


```

Figure 9: KB and rules in Exercise 3.

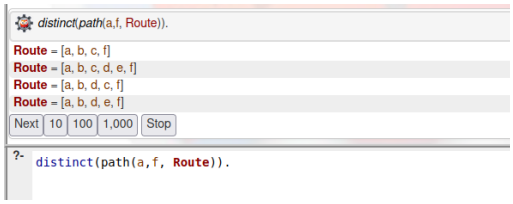

Figure 10: `path(StartNode, EndNode, Route)` query in Exercise 3. Query returns multiple routes as answers.

Exercise 3

Define a `predicate_path/3` which will return a list of nodes (i.e. the path) starting from the initial node to the final node. For example, the following question would result in multiple answers: `?- path(a, f, Route).`

```
Route=[a, b, c, f];
Route=[a, b, d, e, f];
Route=[a, b, d, c, f];
...
```

Solution

To construct the predicates for this exercise we are using both the `member`, and `dif` predicates as a helping hand:

```
%path/3
path(StartNode, EndNode, Route) :- path(StartNode, EndNode, [], Route).

path(StartNode, StartNode, _, [StartNode]).

path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes]) :-
    \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is member of VisitedNodes list
    dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are different
    interconnected(StartNode, Node),
    path(Node, EndNode, [StartNode|VisitedNodes], Nodes).

```

```
connect(a, b, 3).

connect(b, d, 4).

connect(b, c, 12).

connect(d, c, 7).

connect(d, e, 5).

connect(c, f, 11).

connect(e, f, 15).

interconnected(X, Y, C):- connect(X, Y, C).
interconnected(X, Y, C):- connect(Y, X, C).

interconnected(a, b, 3).
interconnected(b, d, 4).
interconnected(b, c, 12).
interconnected(d, c, 7).
interconnected(d, e, 5).
interconnected(c, f, 11).
interconnected(e, f, 15).

%cost_path/4
cost_path(StartNode, EndNode, Route, Cost) :-
    cost_path(StartNode, EndNode, [], Route, Cost).

cost_path(StartNode, StartNode, _, [StartNode], 0).

cost_path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes], Cost) :-
    interconnected(StartNode, Node, C),
    \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is member of VisitedNodes list
    dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are different
    cost_path(Node, EndNode, [StartNode|VisitedNodes], Nodes, RCost),
    Cost is C+RCost.
```

Figure 12: KB and rules for Exercise 4.

```
distinct(cost_path(a,f,Route,Cost)).
Cost = 25,
Route = [a, b, d, c, f]
Cost = 27,
Route = [a, b, d, e, f]
Cost = 26,
Route = [a, b, c, f]
Next 10 100 1,000 Stop
?- distinct(cost_path(a,f,Route,Cost)).
```

Figure 13: `cost_path(StartNode, EndNode, Route, Cost)` query for Exercise 4. Query returns multiple paths and their associated costs as answers.

Exercise 4

Consider the weighted graph shown in Figure 3 where each edge has a specific cost. Modify the facts so that they include the cost information as shown below:

```
connect(a, b, 3).
connect(b, c, 12).
connect(b, d, 4).
...
```

and the rules:

```
interconnected(X, Y, C):- connect(X, Y, C).
interconnected(X, Y, C):- connect(Y, X, C).
```

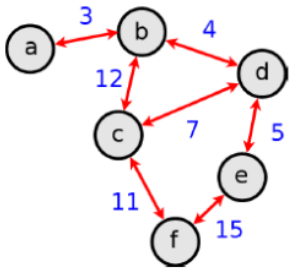

Figure 11: A weighted graph.

Define a predicate `cost_path/4`, which will return a list of nodes starting from the initial node to the final node (like the previous task 3) and the total cost of the path. The total cost of the route should be calculated at the same time as finding it. For example, the following question would result in multiple answers:

```
?- cost_path(a, f, Route, Cost).
Route=[a, b, d, c, f],
Cost=25;
Route=[a, b, c, f],
Cost=26;
...
```

Solution

The solution in this exercise is very similar to the previous one, using a list to keep track of the visited nodes and using the `member` and `dif` predicates, autoloaded from `swi-prolog`.

```
%cost_path/4
```

```
cost_path(StartNode, EndNode, Route, Cost) :-
    cost_path(StartNode, EndNode, [], Route, Cost).

cost_path(StartNode, StartNode, _, [StartNode], 0).

cost_path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes], Cost) :-
    interconnected(StartNode, Node, C),
    \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is member of VisitedNodes list
    dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are different
    cost_path(Node, EndNode, [StartNode|VisitedNodes], Nodes, RCost),
    Cost is C+RCost.

```

Complete Prolog code listing of Assignment # 4 is available on the Appendix, located at the next page.

Appendix

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%   Homework 4  %%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% CS-487 Fall Semester 2022-2023  %%%%
%%%% Department of Computer Science  %%%%
%%%%       University of Crete       %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Nikolaos-Modestos Kougioulis (1285) %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

connect(a, b).

connect(b, c).

connect(b, d).

connect(e, d).

connect(d, c).

connect(c, f).

connect(f, e).

%interconnected(X, Y):- connect(X, Y).
%interconnected(X, Y):- connect(Y, X).
interconnected(X, Y):- connect(X, Y); connect(Y, X).

interconnected(a,b).
interconnected(b,d).
interconnected(e,d).
interconnected(d,c).
interconnected(c,f).
interconnected(f,e).

:- style_check(-singleton).

%exists_path/2
%exists_path(StartNode, EndNode).
%exists_path(StartNode, EndNode) :- connect(StartNode, NextNode), exists_path(NextNode,
    EndNode).

%exists_path/2 with interconnect
exists_path(StartNode, EndNode).
exists_path(StartNode, EndNode) :- interconnected(StartNode, NextNode), exists_path(NextNode
    , EndNode).

%path/3
path(StartNode, EndNode, Route) :- path(StartNode, EndNode, [], Route).

path(StartNode, StartNode, _, [StartNode]).

path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes]) :-
    \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is member of
    VisitedNodes list
    dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are
    different
    interconnected(StartNode, Node),
    path(Node, EndNode, [StartNode|VisitedNodes], Nodes).
```

```
56
57 connect(a, b, 3).
58
59 connect(b, d, 4).
60
61 connect(b, c, 12).
62
63 connect(d, c, 7).
64
65 connect(d, e, 5).
66
67 connect(c, f, 11).
68
69 connect(e, f, 15).
70
71 interconnected(X, Y, C):- connect(X, Y, C).
72 interconnected(X, Y, C):- connect(Y, X, C).
73
74 interconnected(a, b, 3).
75 interconnected(b, d, 4).
76 interconnected(b, c, 12).
77 interconnected(d, c, 7).
78 interconnected(d, e, 5).
79 interconnected(c, f, 11).
80 interconnected(e, f, 15).
81
82 %cost_path/4
83 cost_path(StartNode, EndNode, Route, Cost) :-
84     cost_path(StartNode, EndNode, [], Route, Cost).
85
86 cost_path(StartNode, StartNode, _, [StartNode], 0).
87
88 cost_path(StartNode, EndNode, VisitedNodes, [StartNode|Nodes], Cost) :-
89     interconnected(StartNode, Node, C),
90     \+ member(StartNode, VisitedNodes), %member/2 is true if StartNode is member of
       VisitedNodes list
91     dif(StartNode, EndNode), %dif/2 predicate is true if-f StartNode and EndNode are
       different
92     cost_path(Node, EndNode, [StartNode|VisitedNodes], Nodes, RCost),
93     Cost is C+RCost.
```

Listing 1: Complete Code Listing