Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #11, 2021

Cluj-Napoca

Agenda

- More built in predicates (and relationship to graphs)
  - findall and more
- Constructs and relationship with graphs
  - For all is true
  - Application – same graph
- Search for a path in graphs
  - Dfs
  - Bfs (and other bfs – next time)

Built-in predicates
their utility in graph searching

- `findall` – selects all items (and only those items) with a certain property from a knowledgebase
- Knowing the predicate to use it appropriately
- Knowing HOW is implemented
  - to utilize the strategy in graph problems

findall

(example from Clocksin and Mellish)

- Suppose we have a knowledge base of some drinks lovers:

```
//likes/2
//likes(person,drink_person_likes).

likes(bill,wine).
likes(dick,beer).
likes(harry,beer).
likes(john,beer).
likes(peter,wine).
likes(tom,beer).

?-findall(X,likes(X,beer),L).
Yes, L=[dick,harry,john,tom].
```

`findall` – implementation

```
//findall/3
//findall(collected_var,collector_predicate, collector_var).

findall(X,G,_):-                     // uses an auxiliary knowledge base  (akb)
        asserta(found(end)),         //  marker for the bottom of the akb
        G,                           //= call(G),  G is a predicate; its exe instantiates
                                     //some argument X.
        asserta(found(X)),           //puts G's instantiated argument on top of akb
        fail.                        //request for backtrack to  G.
findall(_,_,L):-                     //reach here when G fails on backtrack
        collect_found([],L).         //starts collecting from the akb, initializing the partial
                                     //result to empty list.
collect_found(P,L):-
        get_next(X),!, //asks for one element in akb; X hidden; known AFTER call
        collect_found([X|P],L). //adds it in front and go recurse
collect_found(L,L).

get_next(X):-
        retract(found(X)),!, //extracts from top of the akb
        X=/=end.                     //succeeds if a genuine data; fails when get to end.
```

findall – implementation

```
//findall/3
//findall(collected_var,collector_predicate, collector_var).

findall(X,G,_):-
        asserta(found(end)),
        G,
        asserta(found(X)),
        fail.
findall(_,_,L):-
        collect_found([],L).

collect_found(P,L):-
        get_next(X),!,
        collect_found([X|P],L).
collect_found(L,L).

get_next(X):-
        retract(found(X)),!,
        X=/=end.
```

`findall` – execution

```
findall(X,G,_):-
        asserta(found(end)),
        G,
        asserta(found(X)),
        fail.
findall(_,_,L):-
        collect_found([],L).

collect_found(P,L):-
        get_next(X),!,
        collect_found([X|P],L).//created bottom-up↑   collected top-down↓
collect_found(L,L).
get_next(X):-
        retract(found(X)),!,
        X=/=end.
```

//initial kb

```
likes(bill,wine).
likes(dick,beer).
likes(harry,beer).
likes(john,beer).
likes(peter,wine).
likes(tom,beer).
```

//akb

```
found(tom).
found(john).
found(harry).
found(dick).
found(end).
```

`findall` – related

- General call: `?-findall(X,G,L)`

  where X is a variable

  G a predicate and X occurs in G

  L would collect all X's so that G's execution succeeds on X
- Creates the list of X's that satisfy G, in the order of occurrence in the original knowledge base
- Associate predicates:
- `setof(X,G,S)`
- Creates the set of terms (standard order, without duplicates; represented also as list) of X's that satisfy G. If none, fails.
- Differences to `findall`:
  - Order: standard vs as found in the knowledge base
  - Duplicates: no vs found in the knowledge base
  - No term: fails vs empty list
- `bagof(X,G,S)`
  - Same as `setof` BUT list NOT ordered + may have duplicates
  - So same as `findall` but fails if no term matches

```
findall(X,G,L):-bagof(X,G,L),!.
findall(_,_,[]).
```

For all is true technique

- Construct to check if `for_all Predicate1 is_true Predicate2`
- Aims to identify/verify whether in all cases when P1 is true, P2 is true as well. Thus, checks P1=>P2.
- Template looks like:

```
for_all_is_true(X,Y):-
        X,
        not(Y),!,
        fail.
for_all_is_true(_,_).
```

```
for_all_is_true(X,Y):-
      X,          //calls X, a predicate, whose execution may instantiate hidden args
      not(Y),!,   //calls Y in the same context. If succeeds, its negation fails,
                  //and backtracks to X who is executed again, with a new instance of args
      fail.       //the first context of X where Y fails, not(Y)  succeeds, and ! is
                  //irreversible crossed and the predicate fails overall.
for_all_is_true(_,_). //if we reached here, in all contexts when X holds true, Y
                  //succeeds as well, therefore, X=>Y
```

For all is true in graph context

- Given graphs G1 and G2 with neighbor and edge representations respectively, do they represent the same graph?

G1

```
neighbor(Node,List_of_Neighbors)
```

G2

```
edge(Node1,Node2)
```

- G1 and G2 same means G1⇔G2 which is (G1=>G2)+(G2=>G1)

(G1=>G2)

```
for_all edges_in_G1
is_true edge_in_G2
```

(G2=>G1)

```
for_all edges_in_G2
is_true edge_in_G1
```

- Define the edges in the 2 representations:

An edge in G1:

```
is_edge_1(X,Y):-     //to find all edges in G1, call is nondeterminist (X,Y, free)
        neighbor(X,L),//finds first pair first (and next on backtrack), instantiates both X and L.
        member(Y,L).//nondeterm call on member; instantiate Y, one at a time, eventually all.
```

An edge in G2:

```
is_edge_2(X,Y):-
        edge(X,Y);
        edge(Y,X).
```

Same graph?

- (G1=>G2)

```
for_all edges_in_G1
is_true edge_in_G2
```

(G2=>G1)

```
for_all edges_in_G2
is_true edge_in_G1
```

- Denote the `for_all_is_true` as `eq1` and `eq2` depending on the implication (G1=>G2) respectively (G2=>G1) we verify

```
eq1:-
      is_edge_1(X,Y),        //for each edge in the first representation
      not(is_edge_2(X,Y)),!, //there is one edge in the other graph
      fail.                  //otherwise we reach here, hence, fail.
eq1. //when we get here, G1=>G2 shown
eq2:-
      is_edge_2(X,Y),
      not(is_edge_1(X,Y)),!,
      fail.
eq2.
same_graph:-eq1,eq2.
```

Change representation of a graph

- Given graph **G1** a weighted, edge representation and we need to get **G2** with neighbor representation

G1

```
edge(a,b,15).
```

G2

```
neighbor(a,[p(b,15),…]).
```

```
gen_graph2:-
        is_edge(X,_,_),
        not(neighbor(X,L)),
        findall(Z,succ(X,Z),L),
        assertz(neighbor(X,L)),
        fail.
gen_graph2.

succ(X,Z):-
        is_edge(X,Y,W),
        Z=p(Y,W).
```

- Note:

`findall(Z,succ(X,Z),L)` same as `findall(p(Y,W), is_edge(X,Y,W),L).`

- Given G2, generate G1. Homework.

Depth first search

- Assume undirected, unweighted graph, edge representation
- 2 stage process: (1) make the trail placing in a queue the vertices in order of visitation (being in the additional knowledge base `vert` (hence a dynamic predicate) is as if we have the vertex colored grey). (2) when done, collect them.

```
df_search(X,_):-
        assertz(vert(X)),//place start/current node in Q
        is_edge(X,Y), //take first/next neighbor of current
        not(vert(Y)), //if already visited, backtrack to take another; if not, continue from Y
        df_serach(Y,_).
df_search(_,L):-
        assertz(vert(end)),
        collect([],L).//similar to collect in findall, but on vert akb.
```

- It is covering just the connected component of the starting vertex (is similar to dfs_visit in Cormen). Need a wrapper to re-run it from all connected components.

Breadth first search

- Assume undirected, unweighted graph, edge representation
- queue made by the akb named `node` (hence dynamic predicate)

```
bf_search(X,_):-
        assertz(node(X)),//add at the end of the Q the current node
        node(Y),//reads from front of Q (first time is first=X=ONLY one in Q) current Y (gets Y instantiated)
        is_edge(Y,Z), //take first/next neighbor of current Y (gets Z instantiated); if none, backtrack and take next from Q
        not(node(Z)), //if Z already visited, backtrack to take another;
        assertz(node(Z)),// if not, put Z in Q and
        fail.   // backtrack anyway to another neighbor of Y
bf_search(_,L):-
        assertz(node(end)),
        collect([],L). //similar to collect in findall, but on node akb.
```

- There is an issue with this implementation for some languages/versions (due to an optimization). Oral explanation. Fix it! Homework.