Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #10,
CS@TUCN

Agenda

- Graphs
  - Search for a path
    - One way
      - V1 (for comparative analysis only)
      - V2 (for comparative analysis only)
      - V3 – nonmonotonic reasoning!
    - Best way
    - All ways
    - Restricted way
- More built in predicates (and relationship to graphs)
  - findall and more

Search for a path in a graph
Labyrinth representation

- Is a graph
  - Rooms = vertices
  - Doors = edges
- Is it directed? Why/why not?
- Enter the labirynth from room a.

```
is_door(a,b).
is_door(b,c).
is_door(b,e).
is_door(c,d).
is_door(d,e).
is_door(e,f).
is_door(e,g).

is_objective (g).
```

Search for a path in a graph Labyrinth way out

- Search the way out = check every possible root. When deadlock, backtrack. Prolog advantage: backtrack is there for free.
- Graph undirected => add necessary predicate

```
is_pass(X,Y):-
        is_door(X,Y);
        is_door(Y,X).
//search/3
//seach(start,objective,path).

search(X,Y,Way):-
        try(X,Y,[X],Way), //try a path from X to Y with the partial path
                          //containing just the starting vertex at this point
        is_objective(Y),!. //why not start with this?
```

Call it with:

```
?-seach (a,X,Way_Out). //is X safe here? Not Y?
```

Search for a path in a graph
Labyrinth way out – main predicate (v1)

```
//try/4
//try(from_vertex,to_vertex,partial_path,final_path)
try(X,X,L,L).
try(X,Y,Thread,Way):-
        is_pass(X,Z),
        not(member(Z,Thread)),
        try(Z,Y,[Z|Thread],Way).
```

- Order of clauses? Why?
- `is_pass(X,Z)`? Why not `is_door(X,Y)`?
- Pattern composition `[Z|Thread]` in the recursive call? Is it correct? Shouldn't it be in the read of the rule? Why?
- Make the analysis of the calling tree. When does try stop? When does try eventually close?
- What answer would we get to the initial query? Discussion on `Way_Out`.

```
?-seach (a,X,Way_Out).
```

- Ask ANY question you have at this point. Is important to understand NOW!

Search for a path in a graph
Labyrinth way out – main predicate (v2)

```
//try/4
//try(from_vertex,to_vertex,partial_path,final_path_ordered)
try(X,X,L,[X]).
try(X,Y,Thread,[X|L]):-
        is_pass(X,Z),
        not(member(Z,Thread)),
        try(Z,Y,[Z|Thread],L).
```

- Initial call?
- Why do we need both `Thread` and `L`? Do we really need both?
- Just use the 4th arg? With

`not(member(Z,L))?`

- Is it possible? What `Thread` and `L` do contain? Make the difference

Understand and NEVER make confusions. Learn on the meaning of pattern composition on:

- The head of the rule
- Recursive call

Search for a path in a graph
Labyrinth way out – main predicate (v3)

```
//try/3
//try(from_vertex,to_vertex,path).
try(X,X,[X]).
try(X,Y,[X|L]):-
        is_pass(X,Z),
        accept(Z), //can Z be part of the thread
        try(Z,Y,L).
//accept/1
//accept(vertex).
accept(X):-
        seen(X),!,
        fail.
accept(X):-
        assert(seen(X)).
accept(X):-
        retract(seen(X)),!
        fail.
```

Search for a path in a graph
Labyrinth way out – main predicate (v3)

- THIS is the nonmonotonic reasoning part
- Is the predicate which states if and WHEN a vertex makes part of the solution
- Reasoning is nonmonotonic (part of default logic) as the assumptions are not nomoton
  - During the reasoning,
    - If an assumption does NOT already take part of the reasoning
    - And does NOT contradict ANY other assumption
    - Is added to the knowledge base
  - If at a latter point
    - If the reasoning cannot be closed (completed), chance is made to attempt some reasoning WITHOUT that piece of knowledge
  - Therefore, quantity of knowledge is not monotonic

```
accept(X):-
        seen(X),!,  //is the contradiction part! The ONLY contradiction is that the vertex is
        fail.       //ALREADY in the solution. If there, don't loop; fail to backtrack!
accept(X):-
        assert(seen(X)). //no contradiction, add it in the solution
accept(X):-
        retract(seen(X)),!//cannot conclude with X in solution, remove and
        fail.                    //backtrack to try WITHOUT it!
```

Labyrinth way out
v3 / v2 comparative analysis

```
Code v3                    //comments v2
accept(X):-
        seen(X),!,         //if member(Z,Thread) succeeds, then
        fail.              //not(member(Z,Thread)) fails and execution in BOTH versions
                           //backtrack to a different neighbor of the current vertex
accept(X):-
        assert(seen(X)). //not(member(Z,Thread)) succeeds, hence Z could be
                           //added to the current attempted solution
accept(X):-
        retract(seen(X)),!,  //is there any point in v2 where we do this?
        fail.                //if so, where?
                             //if not, are solutions similar?
```

//v3 comments
//although X was in the solution
//at some point since there is no final way
//decide to remove it from path

- Q on nonmonotonic reasoning. We'll see it soon again!

Graphs – Find the Best way

```
a_way(V,V,Thread,Th_length):-
        is_objective(V),!
        retract(best(_,_)),!
        asserta(best(Thread,Th_length),
        fail. // cut above. So, where does fail backtracks us? Mistake?
a_way(V1,V2, Thread,Th_length):-
        best(BThread,BTh_length),
        Th_length1 is Th_length +1,
        Th_length1<BTh_length,
        is_pass(V1,V3),
        not(member(V3,Thread)),
        a_way(V3,V2, [V3|Thread],Th_length1).
best_way(V1,V2,Thread):-
        assert(best([],1000)),
        a_way(V1,V2,[V1],1).
best_way(_,_,Thread):-
        retract(best(Thread,_)).
```

Graphs – Find all ways

- Assume neighbor representation:

```
neighb(b,[a,c,e]).
⋮
//a_way/3
//a_way(first_vertex, last_vertex, path).
a_way(V,V,[V]).
a_way(V1,V2,[V1|Rest]):-
        neighb(V1,L),
        ways(L,V2,Rest).

//ways/3
//ways(list_of vertices_to_start_from, last_vertex, path).
ways([V1|_],V2,Way):-
        a_way(V1,V2,Way).
ways([_|Rest],V2,Way):-
        ways(Rest,V2,Way).
```

- Q: How are loops avoided? How/when does this work?

Graphs – Find restricted way

- Assume back the edge representation.

```
is_way_obj(NX,NY,Way):-
        nonvar(NX),                 //meaning? Why needed?
        try(NX,NY,[NX],Way),        //any try from v1, v2, v3
        is_objective(NY).           //why here the test? Not before try? Would be
                                    //better?

is_restricted_way(NX,NY,Restrictions,Way):-
        is_way_obj(NX,NY,Way),
        is_in_order(Restrictions,Way).      //how else could we do?

is_in_ord([NX|TX],[NX|TY]):-!,              //what is this cut cutting?
        is_in_ord(TX,TY).
is_in_ord([NX|TX],[_|TY]):-!,
        is_in_ord([NX|TX],TY).
is_in_ord([],_).
```

Built-in predicates
their utility in graph searching

- `findall` – selects all items (and only those items) with a certain property from a knowledge base
- 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).
likes(bill,beer).
likes(tom,water).

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

findall – implementation

```
//findall/3
//findall(collected_pattern, 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
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).
```

↑ created bottom-up collected top-down ↓

findall

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

  where X is a Template (variable, variable pattern)

  G a predicate (Goal) call (Query), 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

setOf, bagOf

- `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(_,_,[]).
```

- will generate alternative bindings for free variables upon backtracking
  - free variables can be existentially quantified in `G` by using the notation `Variable^Query` => there exists a `Variable` such that `Query` is true.