Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #9

CS@TUCN

Agenda

- Built in predicates – review
  - `functor` and `arg`
  - `=..`
- Utilization on complex tasks
  - Substructure change (partial review and completes now)
  - Towers of Hanoi
- Quiz analysis
- Graphs
  - Representations
  - Search for a path
    - One way
      - V1
      - V2
      - V3 – nonmonotonic reasoning!

Built in predicates

- `functor` and `arg`
- `=..`

Substituting expressions

- Given a (complex) expression, it is requested to replace a given subexpression with another one.

- `subst(OldExpr,NewExpr,OldSubExpr,NewSubExpr).`
- Example: in a large arithmetic expression replace (7-2*x) whereever it occurs with another one, say (2*y+3/z)
- 2 solutions
  - =..
  - functor + arg

```
//subst/4
//subst(OldExpr,NewExpr,OldSubExpr,NewSubExpr).

//subst_args/4
//subst_args(ListOldExpr,ListNewExpr,OldSubExpr,NewSubExpr).
```

Substituting expressions =..

```
subst(Old,New,Old,New):-!.  //in case the whole expression is represented
//by only the subexpression to be replaced, the old one is replaced by the new one
subst(Val,Val,_,_):-                        //in case the expression is an atomic value
            atomic(Val),!.                  //it remains unchanged
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
      Val=..[F|Args], //extract from the old expression its functor and the list
                      //of arguments
      subst_args(Args,NewArgs,OldSubExpr,NewSubExpr),
//take the old list of arguments and process it with substitution
      NewVal=..[F|NewArgs]. //create the new expression with the same
//functor and the new list of arguments, updated by substitutions
subst_args([],[],_,_).
subst_args([Arg|Args],[NArg|NArgs],Old,New):-
      subst(Arg,NArg,Old,New),  //as Arg is an expression, go back to the
//other predicate
      subst_args(Args,NArgs,Old,New). //continue with the rest of
//arguments, tail of list
```

Substituting expressions =.. Just code, comments removed

```
subst(Old,New,Old,New):-!.
subst(Val,Val,_,_):-
            atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
      Val=..[F|Args],
      subst_args(Args,NewArgs,OldSubExpr,NewSubExpr),
      NewVal=..[F|NewArgs].

subst_args([],[],_,_).
subst_args([Arg|Args],[NArg|NArgs],Old,New):-
      subst(Arg,NArg,Old,New),
      subst_args(Args,NArgs,Old,New).
```

Qs:

1. Order of processing the structure?
2. Where the execution ends?

Substituting expressions functor+arg

```
subst(Old,New,Old,New):-!.                //same as before
subst(Val,Val,_,_):-            //same as before
                 atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
        functor(Val,F,N),  //extract from the old expression Val its functor F
                          // and the number of arguments N
        functor(NewVal,F,N), //create a new expression NewVal from the
                          // known functor F  and N  arguments, free variables at the time
        subst_args(N,Val,NewVal,OldSubExpr,NewSubExpr).
//for each of the arguments, 1, 2, …, N, process it one at a time
subst_args(0,_,_,_,_):-!.//arg 0 needs no change
subst_args(N,Val,NewVal,Old,New):-
        arg(N,Val,OldArg),//extract the Nth arg of the old expression
        arg(N,NewVal,NewArg),//set the Nth arg of the new expression; a var at the time
        subst(OldArg,NewArg,Old,New),   //as OldArg is an expression, go back
//to the other predicate  and make the same processing
        N1 is N-1,     //process next the previous argument
        subst_args(N1,Val,NewVal,Old,New). //continue with the rest of
//arguments, tail of list
```

Substituting expressions functor+arg Just code, comments removed

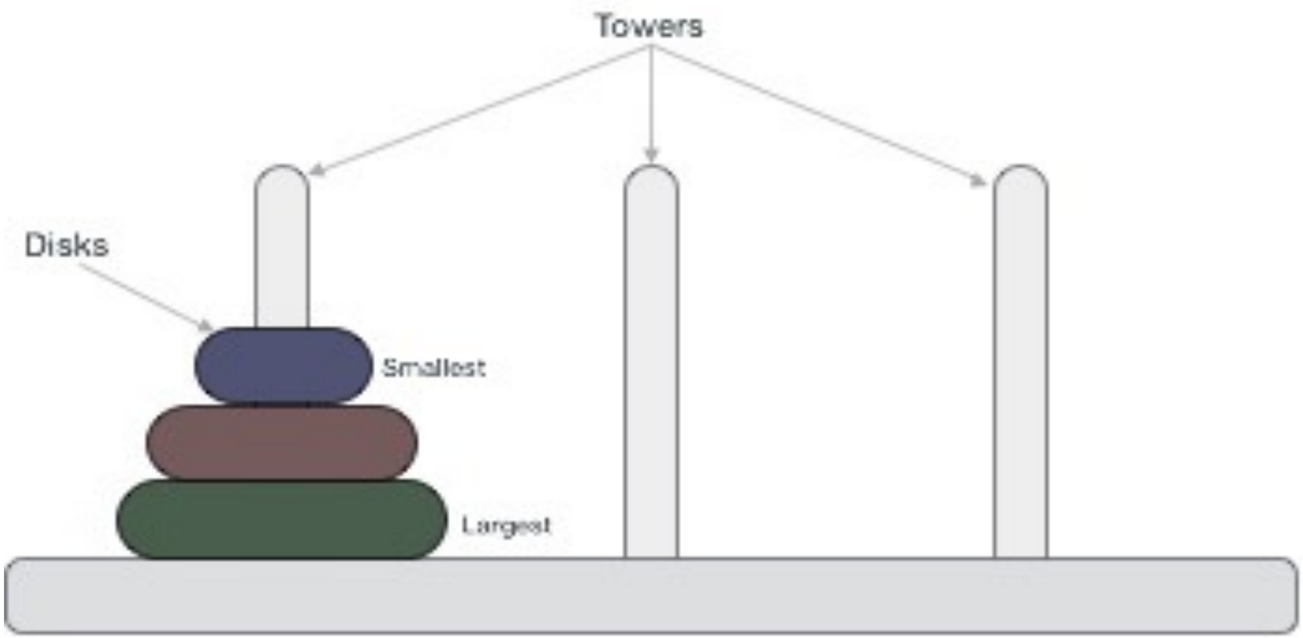
```
subst(Old,New,Old,New):-!.
subst(Val,Val,_,_):-
              atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
       functor(Val,F,N),
       functor(NewVal,F,N),
       subst_args(N,Val,NewVal,OldSubExpr,NewSubExpr).

subst_args(0,_,_,_,_):-!.
subst_args(N,Val,NewVal,Old,New):-
       arg(N,Val,OldArg),
       arg(N,NewVal,NewArg),
       subst(OldArg,NewArg,Old,New),
       N1 is N-1,
       subst_args(N1,Val,NewVal,Old,New).
```

Qs:

1. Order of processing the structure?
2. Where the execution ends?

Towers of Hanoi – the problem



x y z

Picture from
https://www.tutorialspoint.com/data_structures_algorithms/tower_of_hanoi.htm

Towers of Hanoi - formulation

- Given n disks and 3 poles
- Move the disks from X to Z
- At any time the constraints are:
  - One at a time
  - Just the top disk is accessible
  - on any pole, any disk has the diameter smaller than the diameter of the disk on top of which it stays

Towers of Hanoi - rephrasing

Start with n disks on X and end with n disks on Z

$X \xrightarrow{(n,Y)} Z$ means $X \xrightarrow{(n-1,Z)} Y$

$X \rightarrow Z$

$Y \xrightarrow{(n-1,X)} Z$

Means:

Move n discs from X to Z using Y as intermediate by means of:

- Move (n-1) discs from X to Y using Z as intermediate
- move one single disc from X to Z
- Move (n-1) discs from Y to Z using X as intermediate

Towers of Hanoi - rephrasing

Start with n disks on X and end with n disks on Z

$X \xrightarrow{(n,Y)} Z$ means $X \xrightarrow{(n-1,Z)} Y$

$X \rightarrow Z$

$Y \xrightarrow{(n-1,X)} Z$

BUT this is almost Prolog code:

```
move_discs(N,X,Y,Z):- N1 is N-1,
        move_discs(N1,X,Z,Y),
        move_disc(X, Z),
        move_discs(N1,Y,X,Z).
```

Towers of Hanoi - realization

- Disc representation (as stack of facts in the knowledge base):

x(1).
⋮
x(n-1).
x(n). //n is NOT a variable, is a value.

- How is it realized?

```
init_pole(_,[]):-!.
init_pole(P,[N|L]):-
      D=..[P,N]     //P name of pole, as x in example; N size
      asserta(D),   //put on top
      init_pole(P,L).
```

- Where L is a list of integers (diameters of discs) in decreasing order

```
init_list(1,[1]):-!.
init_list(N,[N|Rest]):- N1 is N-1,
      init_list(N1,Rest).
```

Towers of Hanoi – moving one disk

- Move one disc from Input Pole to Output Pole:

```
move_disc(IP,OP):-
        DI=..[IP,Size],    //IP name of input pole, Size diam of disk
        retract(DI),       //extract from top
        DO=..[OP,Size],    //IP name of output pole, Size diam of disk
        asserta(DO).        //put on top
```

- If input has:

```
x(3).
x(4).          y(6).
x(5).          y(7).
```

And call `move_disc(x,y).` goes to:

```
               y(3).
x(4).          y(6).
x(5).          y(7).
```

Towers of Hanoi – complete code

```
move_discs(0,_,_,_):-!.
move_discs(N,X,Y,Z):- N1 is N-1,
            move_discs(N1,X,Z,Y),
            move_disc(X, Z),
            move_discs(N1,Y,X,Z).

move_disc(IP,OP):-
      DI=..[IP,Size],
      retract(DI),
      DO=..[OP,Size],
      asserta(DO).

init_pole(_,[]):-!.
init_pole(P,[N|L]):-
      D=..[P,N],
      asserta(D),
      init_pole(P,L).
```

Towers of Hanoi – complete code with initial call

```
move_discs(0,_,_,_):-!.
move_discs(N,X,Y,Z):- N1 is N-1,
                move_discs(N1,X,Z,Y),
                move_disc(X, Z),
                move_discs(N1,Y,X,Z).
move_disc(IP,OP):-
        DI=..[IP,Size],
        retract(DI),
        DO=..[OP,Size],
        asserta(DO).
init_pole(_,[]):-!.
init_pole(P,[N|L]):-
        D=..[P,N],
        asserta(D),
        init_pole(P,L).
```

// INITIAL call

```
hanoi(N):-init_list(N,List),
        init_pole(input,List),
        move_discs(N,input,int,out).
```

Graphs - Representations

- Traditional representations:
  - Adjacency matrix – not efficient here
  - Adjacency lists – 2 way of doing this
- Neighbor list:
  - Knowledge base with facts of nodes, list of neighbors
- Edge list
  - Knowledge base with edge pairs

Graphs - Representations

- Neighbor list – Knowledge base with facts of nodes, list of neighbors

```
//neighb/2
//neighb(Vertex,List_of_neighb).
neighb(a,[b,c]).
```

- Directed graphs? What changes?
- Isolated nodes?

```
neighb(z,[]).
```

Graphs - Representations

- Edge list - Knowledge base with edge pairs

```
//edge/2
//edge(vertex1,vertex2).
edge(a,b).
edge(a,c).
```

- UNDirected graphs? Add a predicate to ask searching both ways.

```
//is_edge/2
//is_edge(vertex1,vertex2).
is_edge(X,Y):-
      edge(X,Y);  //check one way
      edge(Y,X).  //and the other way
```

- Isolated nodes?

```
edge(z,nil).  //not a built-in way; just assume an edge to some dummy vertex
```

Search for a path in a graph

- Example from Clocksin & Mellish book
- Finding the way out from the labyrinth
- Labyrinth =
  - a set of rooms
  - With doors linking them
  - Unique entrance
  - One objective = way out = reach a given room
  - You **know** you reach it just **AFTER** you entered the room! (that is, you cannot define the problem in terms of "go to room x" as you don't know what x would be beforehand. You have this ability just after you reach there).

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!