Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #12,
CS@TUCN

Agenda

- Graphs isomorphism
  - with metaprogramming (review for comparison)
  - With nonmonotonic reasoning
  - Various implementations (compare & contrast)
- Hamiltonian cycle
  - What is it
  - Class of the problem
  - How (not to) solve it
  - How to address the issue (next lecture)
- BFS using Queues

Graphs isomorphism problem statement

- G1=(V1,E1)
- G2=(V2,E2)
- G1 iso G2 iff

  i) $|V1|=|V2|$

  ii) $\exists\ \varphi$: V1-> V2 a bijection so that

  iii) $\forall\ x, y \in V1\ [(x, y) \in E1 \Leftrightarrow (\varphi(x), \varphi(y)) \in E2)$

```
Ex: graph1([ n(a,[b,c,d]),          graph2([n(1,[2,4]),
             n(b,[a,c]),                    n(2,[1,3,4]),
             n(c,[a,b,d]),                  n(3,[2,4]),
             n(d,[a,c]),                    n(4,[1,2,3]
             ]).                             ]).
```

- Are the 2 graphs isomorphs?

Graphs isomorphism approach (v1)

```
?-graph1(G1),graph2(G2),iso_graph(G1,G2).
iso_graph(L1,L2):-eq_perm(L1,L2,eq_neighb).

eq_perm([H1|T1],L2,EQ):-  //is the perm predicate
      delete(H2,L2,T2),    //with an equivalence predicate
      P=..[EQ,H1,H2],      //which is created here
      call(P),             //and called here
      eq_perm(T1,T2,EQ).
eq_perm([],[],_).

eq_neighb(n(N1,L1),n(N2,L2)):-  //2 neighb pairs are equivalent
            eq_node(N1,N2),     //if the nodes are equivalent
            eq_perm(L1,L2,eq_node).//and the neighb lists are
                                   //equivalent

delete(H,[H|T],T).
delete(X,[H|T],[H|R]):-
      delete(X,T,R).
```

Nodes equivalence
(implements φ function)
V1 – default logic with side effects

- p a dynamic predicate, used for the nomonotonic reasoning
- implements φ function (to form `p` as akb)

```
eq_node(N1,N2):-       // if nodes N1 and N2 already form a pair in the akb
      p(N1,N2).        // the evaluation continues
eq_node(N1,_):-        // if N1 forms a pair with some OTHER node
      p(N1,_),!,       // we get an inconsistency, so should NOT allow
      fail.            // (N1,N2) form a pair, so, fail to backtrack
eq_node(_,N2):-        // symmetric on N2
      p(_,N2),!,
      fail.
eq_node(N1,N2):-       // if you reach here, the is no inconsistency
      asserta(p(N1,N2)).//hence pair N1, N2 is a legitimate one.
eq_node(N1,N2):-       //if at a later moment, although pair N1, N2 is a
      retract(p(N1,N2)),!,//legitimate one, the reasoning cannot
      fail.            //conclude, so remove it from the akb, and fail to
               //backtrack and resume execution WITHOUT the pair in the
akb
```

Graphs isomorphism approach (v1)

```
?-graph1(G1),graph2(G2),iso_graph(G1,G2).
iso_graph(L1,L2):-eq_perm(L1,L2,eq_neighb).
eq_perm([H1|T1],L2,EQ):-
              delete(H2,L2,T2),
              P=..[EQ,H1,H2],
              call(P),
              eq_perm(T1,T2,EQ).
eq_perm([],[],_).
eq_neighb(n(N1,L1),n(N2,L2)):-
              eq_node(N1,N2),
              eq_perm(L1,L2,eq_node).
delete(H,[H|T],T).
delete(X,[H|T],[H|R]):-delete(X,T,R).
eq_node(N1,N2):-p(N1,N2).
eq_node(N1,_):-p(N1,_),!,fail.
eq_node(_,N2):-p(_,N2),!,fail.
eq_node(N1,N2):-asserta(p(N1,N2)).
eq_node(N1,N2):-retract(p(N1,N2)),!,fail.
```

Nodes equivalence
(implements φ function)
V2 – with arguments (lists)

```
?-graph1(G1),graph2(G2),iso_graph(G1,G2).
```

//instead of the akb `p` in v1, here we store the pairs in the arg list

// arg 4 = partial list, empty initially (arg to model the akb)

//arg 5 = final list; a copy of the partial list at the end of the execution (arg with the status of the akb at the end; the bijection)

```
iso_graph(L1,L2):-eq_perm(L1,L2,eq_neighb,[],Lout).
eq_perm([H1|T1],L2,EQ,LI,LO):-//same as in v1
      delete(H2,L2,T2),
      P=..[EQ,H1,H2,LI,Lint],  //just that with args
      call(P),
      eq_perm(T1,T2,EQ,Lint,LO).
eq_perm([],[],_,L,L).

eq_neighb(n(n1,L1),n(N2,L2),LI,LO):-
            eq_node(N1,N2,LI,Lint),
            eq_perm(L1,L2,eq_node,Lint,LO).
```

Nodes equivalence
(implements φ function)
V2 – with arguments (lists) – contd.

- p is a pair of nodes in the list argument
- list evolves nomonotonic (during reasoning) to allow for pairs addition/removal
- models φ function

```
eq_node(N1,N2,LI,LI):-
      member(p(N1,N2),LI),!.
eq_node(N1,N2,LI,[p(N1,N2)|LI]):-
      not(member(p(N1,_),LI)),
      not(member(p(_,N2),LI)).
```

- Those 2 clauses do the same as those 5 in v1
- How? Find 1to1 correspondence!

Nodes equivalence
(φ function v2 VS v1)

```
eq_node(N1,N2,LI,LI):-
        member(p(N1,N2),LI),!.

eq_node(N1,N2,LI,[p(N1,N2)|LI]):-

        not(member(p(N1,_),LI)),

        not(member(p(_,N2),LI)).
```

```
eq_node(N1,N2):-
        p(N1,N2).
eq_node(N1,_):-
        p(N1,_),!,
        fail.
eq_node(_,N2):-
        p(_,N2),!,
        fail.
eq_node(N1,N2):-
        asserta(p(N1,N2)).
eq_node(N1,N2):-
        retract(p(N1,N2)),!,
        fail.
```

Nodes equivalence
(implements φ function)
V3 – with arguments (incomplete lists)

- Same as v2 just that lists are incomplete
- So, it must adjust the behavior of the search predicate to handle appropriately the "not found" case
- Just the equivalence changes (and the `eq_perm` predicate needs just 1 list arg):

```
eq_node(N1,N2,[p(N1,N2)|_]):-!.
eq_node(N1,_,[p(N1,_)|_]):-
                  !,fail.
eq_node(_,N2,[p(_,N2)|_]):-
                  !,fail.
eq_node(N1,N2,[_|T]):-
                  eq_node(N1,N2,T).
```

- Compare v1 with v3. Why is 1 (from v1) clause missing (in v3)?
- Compare v2 with w3.

Nodes equivalence
(φ function v3 VS v1)

```
eq_node(N1,N2,[p(N1,N2)|_]):-!.
eq_node(N1,_,[p(N1,_)|_]):-
                        !,fail.
eq_node(_,N2,[p(_,N2)|_]):-
                        !,fail.
eq_node(N1,N2,[_|T]):-

       eq_node(N1,N2,T).
```

```
eq_node(N1,N2):-
       p(N1,N2).
eq_node(N1,_):-
       p(N1,_),!,
       fail.
eq_node(_,N2):-
       p(_,N2),!,
       fail.
eq_node(N1,N2):-
       asserta(p(N1,N2)).
eq_node(N1,N2):-
       retract(p(N1,N2)),!,
       fail.
```

Nodes equivalence
(φ function v3 VS v2)

```
eq_node(N1,N2,[p(N1,N2)|_]):-!.
eq_node(N1,_,[p(N1,_)|_]):-
                        !,fail.
eq_node(_,N2,[p(_,N2)|_]):-
                        !,fail.
eq_node(N1,N2,[_|T]):-

       eq_node(N1,N2,T).
```

```
eq_node(N1,N2,LI,LI):-
       member(p(N1,N2),LI),!.

eq_node(N1,N2,LI,[p(N1,N2)|LI]):-

       not(member(p(N1,_),LI)),

       not(member(p(_,N2),LI)).
```

Hamiltonian cycle

- Consider the undirected weighted graph:

```
edge(a,b,7).
edge(a,c,1).
edge(a,d,8).
edge(b,c,2).
edge(b,f,5).
edge(b,e,10).
edge(c,d,3).
edge(d,f,4).
edge(d,e,8).
edge(f,e,3).

is_edge(X,Y,W):-
        edge(X,Y,W);
        edge(Y,X,W).
```

Hamiltonian cycle (NP complete/NPhard problem)

```
hamilton(N,X,Cycle,Cost):-
      N1 is N-1,
      try(N1,X,X,[X],Cycle,0,Cost).

?-hamilton(6,a,Cycle,Cost).

try(N,X,Y,Pway,Cycle,Pcost,Cost):-
      is_edge(Y,Z,W),
      not(member(Z,Pway)),
      N1 is N-1, N1>0,
      NewPcost is Pcost+W,
      try(N1,X,Z,[Z|Pway],Cycle,NewPcost,Cost).
try(0,X,Y,Pway,[X|Pway],Pcost,Cost):-
      is_edge(Y,X,W),
      Cost is Pcost+W.
```

Hamiltonian cycle (NP complete/NPhard problem)

- Should not be addressed straight away unless the space is VERY small (how small? Size ~20? TBD orally)
- What should be done? One approach next lecture!

Bfs – queues explicit

- Bfs with 3 arguments representing:
  - Candidate list = queue = list of items we reached, yet not finalized to process = grey nodes. Implemented as incomplete lists. Why? Try to find out. Oral discussion!
  - Expanded list = list of items reached and processed = black nodes. Implemented as regular (complete) lists.
  - Result = nodes in order of bfs processing. Copy of Expanded list when processing ends.

```
//bf_search/3
//bf_search(queue_cand,list_expand,result).
?-bf_search([a|_],[],Rez).

bf_search(Cand,Exp,Exp):-//when Q is empty, end exe, the Exp becomes Rez
            var(Cand),!.
bf_search([X|Cand],Exp,Rez):-//else, take first from Q
      expand(X,Cand,Exp),        // and expand (put all white neighbors in Q)
      bf_search(Cand,[X|Exp],Rez).// and continue by moving it in expanded =
                                                 //make it black
```

Bfs – queues explicit - contd

- The expansion step – decides which neighbors to add in Q
- 2-stage process
- `descend` – dynamic predicate; potential neighbors stored in additional knowledge base (akb)

```
expand(X,_,Exp):-
        is_edge(X,Z),  //nondeterministically take Z,  first neighbor of x (eventually all of them)
        not(member(Z,Expand)),//should NOT be already processed (=not  a black node)
        assertz(desc(Z)),        //potentially add it in Q
        fail.                    //backtrack to evaluate another neighbor of X
expand(_,Cand,_):-
        assertz(desc(end)),      //mark end of akb
        collect(Cand).

collect(Cand):-
        get_next(X),             //as long as akb not empty
        insert_IL(X,Cand),//take one and if not under processing (not a grey one) add it in Q
        collect(Cand).           //continue. How is possible with the SAME argument?
collect(Cand).                   //end when akb empty
```

Bfs – queues explicit - contd

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

insert_IL(X,[X|_]):-!.
insert_IL(X,[_|L]):-
        insert_IL(X,L).

?-bf_search([a|_],[],Rez).
```
//for the first graph – search for a path

Bfs – complete code

```
bf_search(Cand,Exp,Exp):-
               var(Cand),!.
bf_search([X|Cand],Exp,Rez):-
       expand(X,Cand,Exp),
       bf_search(Cand,[X|Exp],Rez).
expand(X,_,Exp):-
       is_edge(X,Z),
       not(member(Z,Exp)),
       assertz(desc(Z)),
       fail.
expand(_,Cand,_):-
       assertz(desc(end)),
       collect(Cand).
collect(Cand):-
       get_next(X),
       insert_IL(X,Cand),
       collect(Cand).
collect(Cand).
get_next(X):-
       retract(desc(X)),!
       x=/=end.
```

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1
Exp:	[]
Cand:	[a|_]

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1	Step2
Exp:	[]	[a]
Cand:	[a|_]	[b|_]

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1	Step2	Step3
Exp:	[]	[a]	[b,a]
Cand:	[a|_]	[b|_]	[e,c|_]

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1	Step2	Step3	Step4
Exp:	[]	[a]	[b,a]	[e,b,a]
Cand:	[a|_]	[b|_]	[e,c|_]	[c,d,f,g|_]

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1	Step2	Step3	Step4	Step5
Exp:	[]	[a]	[b,a]	[e,b,a]	[c,e,b,a]
Cand:	[a|_]	[b|_]	[e,c|_]	[c,d,f,g|_]	[d,f,g|_]

Bfs – execution check the whiteboard for tracing

`?-bf_search([a|_],[],Rez).`//for the first graph – search for a path

	Step1	Step2	Step3	Step4	Step5	Step6
Exp:	[]	[a]	[b,a]	[e,b,a]	[c,e,b,a]	[d,c,e,b,a]
Cand:	[a|_]	[b|_]	[e,c|_]	[c,d,f,g|_]	[d,f,g|_]	[f,g|_]

	Step7	Step8	DONE!
Exp:	[f,d,c,e,b,a]	[g,f,d,c,e,b,a]	
Cand:	[g|_]	_	

Next time

- B&B solutions to address complexity issues