Inference control and search

Lecture 4

Inference control and search

(in PROLOG)

- Negation as Failure
- Inference control
- Cut to control backtracking
- Meta programming (hints)
- search

SLDNF = SLD-resolution + NF

NF (Negation as Finite Failure):
an atom $\neg A$ succeeds if the derivation tree corresponding to the goal A is finite and its leaves are all failure leaves.

Consistent (with the various characterizations of negation), but incomplete.

Conditions for completeness are very stringent:

- instantiated variables in negated atoms
- constraints on the program structure

Answer Set Programming extends SLDNF, computing the answer on the basis of the stable model semantics

Negation as Failure in PROLOG

In prolog negation is realized as **failure** in the search.

If `X` is a list of atoms, `not(X)` is true if the interpreter cannot find a proof for `X`.

```
student(bill).
student(joe).
married(joe).

unmarried_student(X) :- not(married(X)), student(X).
? unmarried_student(bill/joe).
```

Termination of negation

The negation mechanism of Prolog is neither correct nor complete: it depends on the ordering of evaluation of clauses

The termination of `not(X)` depends on the termination of `X`:
- If `X` terminates, `not(X)` terminates.
- If `X` has infinite solutions, `not(X)` terminates if a success node is found before an infinite branch is encountered.

```
married(gino, remberta).
married(X,Y):-married(Y,X).

? not(married(remberta, gino)).
```

Example

```
p(s(X)):-p(X).
q(a).
```

The query `not(p(X),q(X))` does not terminate.

Instead, `not(q(X),p(X))` terminates.

Using negation with completely instantiated atoms does not generate incorrect behaviors, in case the atoms terminate. If negation is used with variables we must take into account the execution mechanism of prolog

Problems with negation

```
unmarried_student(X):-not(married(X)),student(X).
student(bill).
married(joe).
?- unmarried_student(X).

unmarried_student1(X) :- student(X), not(married(X)).
student(bill).
student(joe).
married(joe).
?- unmarried_student1(X).
```

Sufficient condition for correctness:
variables in atoms must be instantiated before they are executed

Inference control in PROLOG programs

- Program specification
- Clause ordering
- Conjunct ordering
- Cut to control backtracking

PROLOG program specification

```
descendant(X,Y):-son(X,Y). % 1
descendant(X,Y):-son(Z,Y),descendant(X,Z).

descendant(X,Y):-son(X,Y). % 2
descendant(X,Y):-son(X,Z),descendant(Z,Y).

descendant(X,Y):-son(X,Y). % 2
descendant(X,Y):-descendant(X,Z),descendant(Z,Y).
```

Explicit Inference Control

"What is the income of the US president' wife?"

```
income(S,I),married(S,P),job(P,uspresident).
```

"Do I have an american cousin?"

```
american(Y),cousin(daniele,Y).
```

♢ **meta-reasoning** to decide which order of the atoms is more effective

Given a predicate returning the number of instances of a relation, one can sort the conjuncts to optimize the computation of the answer.

Efficiency of PROLOG implementations

- **chronological backtracking**
- **dependency directed backtracking**
- **intelligent backtracking** memorizing the previous derivation to avoid re-discovering them

Cut

Generic clause with cut:

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n.$$

If the current goal G unifies with A and $B_1, \ldots, B_k$ are successfully proven,

- any other proof achievable by unifying G with another clause is eliminated from the search tree;
- any alternative proof of $B_1, \ldots, B_k$, is also eliminated from the search tree.

Cut: example

```
merge1([X|Xs],[Y|Ys],[X|Zs]):-
        X<Y,!,
        merge1(Xs,[Y|Ys],Zs).
merge1([X|Xs],[Y|Ys],[X,Y|Zs]):-
    X==Y,!,
    merge1(Xs,Ys,Zs).
merge1([X|Xs],[Y|Ys],[Y|Zs]):-
    X>Y,!,
    merge1([X|Xs],Ys,Zs).
merge1([],X,X):- !.
merge1(Y,[],Y):- !.
```

Green and red cuts: discarding valid solutions

A cut is **green**, when it does not affect the program solutions, **red** otherwise.

```
descendant(X,Y):-son(X,Y). % 1
descendant(X,Y):-son(Z,Y),!,descendant(X,Z).
```

Stops the search at the first son of Y: Z. Since X may not be a descendant of the first son Z, the search would fail, even if X is the descendant of (another) son of Y.

Controlling backtracking

```
cousin(massimo,X),!,american(X)

member(a,C), !, q(a).

member1(X,[X|Xs]) :- !.
member1(X,[Y|Ys]) :- member1(X,Ys).

member1(a,C).
```

Meta-logical predicates

♢ Needed to deal with a program element as data.

♢ Access the program's internal representation.

♢ Enable meta-programming.

♢ Meta-programming makes it possible to **explicitly** control execution

Collecting solutions

It is often useful to compute all the solutions of a program.

`findall(Term,Goal,List).`

`List` is the list, obtained computing `Goal` and taking the bindings of `Term`.

`findall(X,grandfather(giovanni,X),Grandchildren).`

Variables as programs

- `call(X)` executes the goal `X` which must be instantiated, (simply written `X`).

Es: Disjunction

```
X ; Y :- X.
X ; Y :- Y.
```

Negation as failure

```
not(G):- G, !, fail.
not(G).
```

`fail` always fails.

Type meta-logical predicates

- `var(Term)` succeeds if `Term` is a variable
- `nonvar(Term)` succeeds if `Term` is not a variable

```
grandfather(X,Z):- nonvar(X),
                   parent(X,Y),
                   parent(Y,Z).
grandfather(X,Z):- nonvar(Z),
                   parent(Y,Z),
                   parent(X,Y).
```

Equality and Unification

- `X == Y` succeeds if `X` and `Y` are the same constant, the same variable, or structures with the same functor and recursively == (negated \ ==)
- `X = Y` succeeds if `X` and `Y` unify

Search in PROLOG

- Manage state transitions:
  applicable(Action,State) and
  apply(Action,State,NewState);
- Avoid repeated states; identical states;
  sameState(S1,S2);
- Initial state and goal state
  Predicates **initState(S)** and **finalState(S)**;
- Depth first search using PROLOG backtracking.

PROLOG program structure

```
solution(ActionList):-init(I),
   solution(I,[],ActionList).
solution(State,VisitedStates,[]) :-
   final(State).
solution(State,VisitedStates, [action|Rest]) :-
   applicable(Action,State),
   apply(Action,State,NewState),
   not visited(NewState,VisitedStates),
   solution(NewState,[State|VisitedStates],Rest).
```

Avoiding repeated states

```
visited(State,[VisitedState|OthVisitedStates]) :-
   sameState(State,VisitedState).
visited(State,[VisitedState|OthVisitedStates]) :-
   visited(State,OthVisitedStates).

sameState(S1,S2):-...
```

(depends on the state representation)

Crossing the river: LPC

A man has a wolf, a sheep and a cabbage; he is on a river bench with a boat, whose maximum load for a single trip is the man, plus only one of his 3 goods. The man wants to cross the river with his goods, but he must avoid that (when he is far away) the wolf eats the sheep or the sheep eats the cabbage. what should he do?

LPC: state space S

Let $S = D \times D \times D \times D$ where $D = \{a, b\}$, a and b are the 2 river benches
$< U, L, P, C > \in S$ represents the position of the man, the wolf and the cabbage.

Moreover, in D $\overline{a} = b$ and $\overline{b} = a$.

Initial State: $< a, a, a, a >$
Goal state $< b, b, b, b >$

LPC: operators

Operator	Condition	From state	To state
carryNothing	$L \neq P, P \neq C$	$< U, L, P, C >$	$< \overline{U}, L, P, C >$
carryWolf	$U = L, P \neq C$	$< U, L, P, C >$	$< \overline{U}, \overline{L}, P, C >$
carrySheep	$U = P$	$< U, L, P, C >$	$< \overline{U}, L, \overline{P}, C >$
carryCabbage	$U = C, L \neq P$	$< U, L, P, C >$	$< \overline{U}, L, P, \overline{C} >$

LPC: a solution

$$< a, a, a, a > -carrySheep\rightarrow < b, a, b, a >$$
$$< b, a, b, a > -carryNothing\rightarrow < a, a, b, a >$$
$$< a, a, b, a > -carryWolf\rightarrow < b, b, b, a >$$
$$< b, b, b, a > -carrySheep\rightarrow < a, b, a, a >$$
$$< a, b, a, a > -carryCabbage\rightarrow < b, b, a, b >$$
$$< b, b, a, b > -carryNothing\rightarrow < a, b, a, b >$$
$$< a, b, a, b > -carrySheep\rightarrow < b, b, b, b >$$

Crossing the river in PROLOG

```
init(pos(0,0,0,0)).
final(pos(1,1,1,1)).
applicable(carryNothing,pos(U,L,P,C)):-
   L =/= P,P =/= C.
applicable(carryWolf,pos(UeL,UeL,P,C)):-
   P =/= C.
applicable(carrySheep,pos(UeP,L,UeP,C)).
applicable(carryCabbage,pos(UeC,L,P,UeC)):-
   L =/= P.
```

Crossing the river in PROLOG (cnt'ed)

```
apply(carryNothing,pos(U,L,P,C),pos(NewU,L,P,C)):-
   NewU is 1-U.
apply(carryWolf,pos(U,L,P,C),pos(NewU,NewL,P,C)):-
   NewU is 1-U, NewL is 1-L.
apply(carrySheep,pos(U,L,P,C),pos(NewU,L,NewP,C)):-
   NewU is 1-U, NewP is 1-P.
apply(carryCabbage,pos(U,L,P,C),pos(NewU,L,P,NewC)):-
   NewU is 1-U, NewC is 1-C.
```

N-queens

```
queens(N,Qs) :- range(1,N,Ns),
                permutation(Ns,Qs),
                safe(Qs).
safe([Q|Qs]) :- safe(Qs), not attack(Q,Qs).
safe([]).

attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|Ys]) :- X is Y+N ; X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).

range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).
```

Anticipate the test

```
queens1(N,Qs) :- range(1,N,Ns), queens1(Ns,[],Qs).

queens1(UnplacedQs,SafeQs,Qs) :-
        select(Q,UnplacedQs,UnplacedQs1),
        not attack(Q,SafeQs),
        queens1(UnplacedQs1,[Q|SafeQs],Qs).
queens1([],Qs,Qs).
```

Esercises (i)

- Define in PROLOG the relation `onlychild(X)`, exploiting the family defined before.
- Define the predicate `notMember(X,L)`, true if `X` does not occur in the list `L`. Provide a definition using NAF and one without it, and compare them.
- Define union using the negation of `member`. Build the search tree for the goal

  ```
  ?  union([a,b],[b,c],Z).
  ```

Esercises (ii)

- Apply the cut operator to the program `insert` of assignment 2, and draw the search tree for the goal `insert(4, [3,5,7], X)`.
- Define a predicate `sortm(X,Y,Z)`, to order lists `X` and `Y` and return the result in `Z`, through merge-sort.
- Add cuts to the program `sortm` if needed to obtain a single solution.