1986-27

Controlling Attention in Inference Machine

by

Heedong Ko

1.

Introduction

This project is to exploit the possibility of controlling the attention of the inference procedure under the environment of the popular logic programming language, Prolog. In order to manipulate the control structure, it is necessary to manipulate the interpretive procedure. So the project entailed writing interpreters for each control issues considered. Blind and heuristic backward search, blind and heuristic forward search and finally heuristic bidirectional search are considered. In the following, each control strategies are explained and algorithms are presented with prolog implementations, traces of execution and proof tree. Proof trees are included because it is rather confusing and tedious to follow the trace alone. The proof tree will give hints what to look for in the trace. The trace will show the order of nodes visited by the interpreter and reason for failure.

It is important that each interpretive procedure works on a single representation because it would be impossible to combine them if they did not. So the representation of the database is " A <- B,C " meaning that A is implied by B and C. Note the assertion is of the form, " A <- true". I will call the left hand side of implication as the consequent and the right hand side as the antecedent. I may also call the implication, a clause or a rule depending on the context.

2.

Blind backward search

In problem solving point of view, this search strategy is top–down: that is, given a problem, a goal, the problem is divided into subproblems, subgoals, until the solution to the problem is known. The proof tree is formed from the root down to leaves, thereby, top–down.

More precisely, when given a goal, find a clause whose consequent matches with the goal and recursively calls the procedure with its antecedent. Note this is depth first search. That means the procedure is not complete in a sense that even though there is a clear solution to the problem, the procedure can not prove it. This motivated loop detection interpreter in the next section.

Since the trace of this procedure will be amply illustrated in the following sections when comparing with heuristic backward searches, only implementations is given here.

Implementation:

```
bw((G1 , G2)) :- bw(G1) , bw(G2).
bw(G) :- clause((G <- true),_).
bw(G) :- clause((G <- Body),_), bw(Body).
```

3. Loop detection

How can a loop be detected? It can be detected when there is a back edge. Back edge is an edge which goes from a descendant to an ancestor.(Aho et al.) The ancestors in the solution graph are old goals already resolved in the inference process. So they are kept track of in a list and if the present goal occurs in this list, back edge is detected and an alternative is sought.

Implementation:

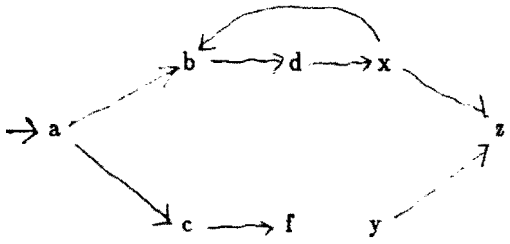
```
% Detects whether first argument occurs in the second argument set.
occursin(G,[G|_]).
occursin(G,[_|T]) :- occursin(G,T).

% Old is the list which holds the ancestors.
% Note back edge is detected by occursin test in the third clause.
% If bw4(B,[G|Old]) fails, alternative clause is sought by clause((G <- B),_).
bw4((G1,G2),Old) :- bw4(G1,Old),
                    bw4(G2,Old).
bw4(G,Old) :- clause((G <- true),_).
bw4(G,Old) :- clause((G <- B),_),
              not(occursin(G,Old)),
```

bw4(B,[G|Old]).

Database:

% This example is a path finding problem. go(N) means it is possible to go
% to node N. If there is an edge between M and N, then assert
% go(N) <- go(M). (note go(N) <- true means N is the start node)
go(a) <- true.
go(b) <- go(x). go(b) <- go(a). go(d) <- go(b). go(x) <- go(d).
go(z) <- go(x). go(c) <- go(a). go(f) <- go(c). go(z) <- go(y).

Pictorial representation of database:



Trace:

** Loop detecting procedure bw4
: bw4(go(z),[])!
C$_{|}$ > bw4(go(z), [])
R$_{|}$ > bw4(go(z), [])
C$_{||}$ > bw4(go(x), [go(z)])
R$_{||}$ > bw4(go(x), [go(z)])
C$_{|||}$ > bw4(go(d), [go(x), go(z)])
R$_{|||}$ > bw4(go(d), [go(x), go(z)])
C$_{||||}$ > bw4(go(b), [go(d), go(x), go(z)])
R$_{||||}$ > bw4(go(b), [go(d), go(x), go(z)])
C$_{|||||}$ > bw4(go(x), [go(b), go(d), go(x), go(z)])
R$_{|||||}$ > bw4(go(x), [go(b), go(d), go(x), go(z)])
F$_{|||||}$ < bw4(go(x), [go(b), go(d), go(x), go(z)]) ** failed because go(x) occurs in the list
R$_{||||||}$ > bw4(go(a), [go(b), go(d), go(x), go(z)])
E$_{||||||}$ < bw4(go(a), [go(b), go(d), go(x), go(z)])
E$_{|||||}$ < bw4(go(b), [go(d), go(x), go(z)])
E$_{|||}$ < bw4(go(d), [go(x), go(z)])
E$_{||}$ < bw4(go(x), [go(z)])
E$_{|}$ < bw4(go(z), [])

** Blide backward search in the previous section.
** Overflow is cause by d–d–x loop in the database.

: bw(go(z))!

STACK OVERFLOW
: :

4.

Heuristic backward search

4.1.

Goal ordering according to the degree of instantiation

When there are more than one goal to be satisfied, it is desirable to try to satisfy the goal whose arguments are most instantiated because intuitively, the number of alternatives possible for a goal whose arguments are all instantiated is fewer than those of the goal with uninstantiated variables.

Algorithm:

```
procedure bw2(Goals);
 if there are more than one goal in Goals
   then begin
     find the goal, G, whose arguments are most instantiated in Goals;
     bw2(G) and bw2(Goals - G)
    end
 else if the Goals is a known fact then succed
     else begin
         find the clause whose consequent matches with Goals;
         if not failed then bw2(Body) (* Body is the antecedent of the found clause *)
     end
```

Implementation

```
% change set notation to regular form which is amenable to
% the representation bw2 expects.
convert([],_) :- fail.
convert([H|T],(H,T1)) :- nonvar(T),convert(T,T1).
convert([H],H).

% Num is the number of instantiated arguments in Args.
num(Args,Num) :- num1(Args,0,Num).
num1([],X,X):-!.
num1([A1| A2],Oldmax,Newmax) :- nonvar(A1),T is Oldmax+1,num1(A2,T,Newmax),!.
num1([A1| A2],Oldmax,Newmax) :- num1(A2,Oldmax,Newmax).

% Choose the goal in the first argument whose arguments are most instantiated,
% Newg, and the rest in Newr.
chsg((G1,G2),Newg,Newr)
 :- G1 =.. [_|Arg],
   num(Arg,N),
   chsg1(G2,N,G1,[],Newg,Newr).
chsg1((G,G1),Max,Oldg,Oldr,Newg,Newr)
 :- G =.. [_|Arg],
```

```
    num(Arg,N1),N1 >= Max,
    chsg1(G1,N1,G,[Oldg|Oldr],Newg,Newr),!.
chsg1((G,G1),Max,Oldg,Oldr,Newg,Newr)
  :- chsg1(G1,Max,Oldg,[G|Oldr],Newg,Newr),!.
chsg1(G,Max,Oldg,Oldr,G,[Oldg|Oldr])
  :- G =.. [_|Arg],
    num(Arg,N1), N1 >= Max.
chsg1(G,Max,Oldg,Oldr,Oldg,[G|Oldr]).

% When there are more than one goals to prove, prove the goal whose
% arguments are most instantiated first
bw2((G1 , G2)) :- chsg((G1,G2),Newg1,Newg2),bw2(Newg1),
            convert(Newg2,X),bw2(X).
bw2(G) :- clause((G <- true),_).
bw2(G) :- clause((G <- Body),_), bw2(Body).
```

Database:

```
% I will use abreviation in proof tree: i.e. p for parent.
father(ares,harmonia) <- true.
mother(aphrodite,harmonia) <- true.
father(zeus,ares) <- true.
mother(hera,ares) <- true.
father(cadmus,semele) <- true.
mother(harmonia,semele) <- true.
god(zeus) <- true.
god(hera) <- true.
god(ares) <- true.
god(aphrodite) <- true.
fairy(harmonia) <- true.
male(X) <- father(X,Y).
female(X) <- mother(X,Y).
parent(X,Y) <- father(X,Y).
parent(X,Y) <- mother(X,Y).
grandparent(X,Y) <- parent(X,Z),parent(Z,Y).
```

Trace:

1. Heuristic backward search with goal ordering

```
: bw2(grandparent(X,harmonia))!
C| > bw2(grandparent(_0, harmonia))
R| > bw2(grandparent(_0, harmonia))
C|| > bw2((parent(_0, _79) , parent(_79, harmonia)))
C||| > bw2(parent(_79, harmonia))          ** note the second goal is tried first!!!
R||| > bw2(parent(_79, harmonia))
C|||| > bw2(father(_79, harmonia))
E|||| < bw2(father(ares, harmonia))
E||| < bw2(parent(ares, harmonia))
C||| > bw2(parent(_0, ares))
R||| > bw2(parent(_0, ares))
C|||| > bw2(father(_0, ares))
E|||| < bw2(father(zeus, ares))
E||| < bw2(parent(zeus, ares))
```

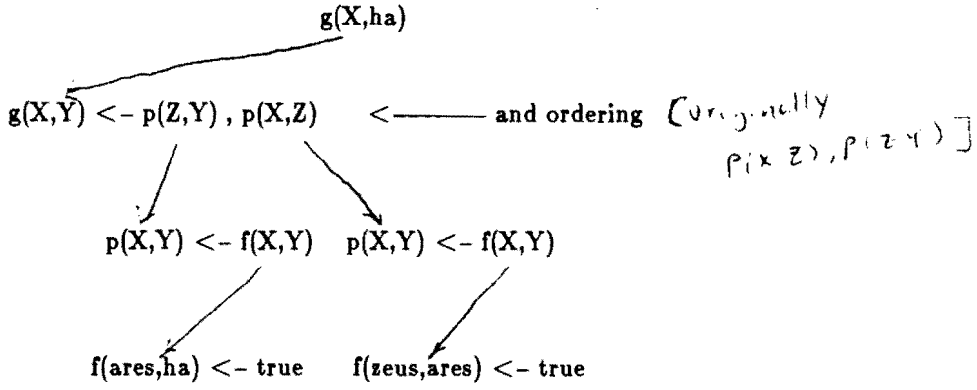
E|| < bw2((parent(zeus, ares) , parent(ares, harmonia)))
E| < bw2(grandparent(zeus, harmonia))

2. Blind backward search

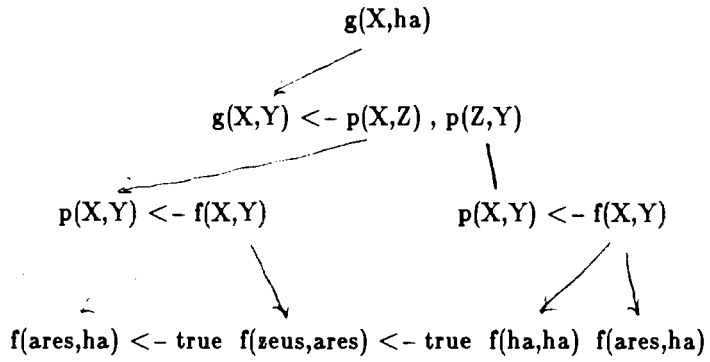
: bw(grandparent(X,harmonia))!
C| > bw(grandparent(_0, harmonia))
R| > bw(grandparent(_0, harmonia))
C|| > bw((parent(_0, _79) , parent(_79, harmonia)))
C||| > bw(parent(_0, _79))
R||| > bw(parent(_0, _79))
C|||| > bw(father(_0, _79))
E|||| < bw(father(ares, harmonia))
E||| < bw(parent(ares, harmonia))
C||| > bw(parent(harmonia, harmonia))
R||| > bw(parent(harmonia, harmonia))
C|||| > bw(father(harmonia, harmonia))
R|||| > bw(father(harmonia, harmonia))
F|||| < bw(father(harmonia, harmonia))
R||||| > bw(mother(harmonia, harmonia))
R||||| > bw(mother(harmonia, harmonia))
F||||| < bw(mother(harmonia, harmonia))
F||| < bw(parent(harmonia, harmonia))
E|||| < bw(father(zeus, ares))
E||| < bw(parent(zeus, ares))
R||| > bw(parent(ares, harmonia))
R||| > bw(parent(ares, harmonia))
C|||| > bw(father(ares, harmonia))
E|||| < bw(father(ares, harmonia))
E||| < bw(parent(ares, harmonia))
E|| < bw((parent(zeus, ares) , parent(ares, harmonia)))
E| < bw(grandparent(zeus, harmonia))

Proof tree

1. Heuristic backward search with goal ordering



2. Blide backward search



4.2. Goal ordering according to the number of corresponding procedures

Another strategy is to order the goals according to the number of procedures: that is, the clauses whose consequent unifies with the goal. Numerous examples were tried but did not save too much effort compared to blind search. This experiment seems to indicate for conjunctive goals, selecting goals according to the number of corresponding procedures is not a good heuristic.

4.3. Procedure ordering

To rescue this situation, another heuristic was tried: the procedure with minimum brancing factor was chosen. The branching factor of a procedure was calculated by summing the number of procedures for each conjuntive goals in the antecedent of the procedure. The difference with above strategy is the different procedures are compared instead of the goals. (In other words, or–ordering instead of and–ordering)

Algorithm:

```
Procedure bw3(Goals)
  if there are more than one goals in Goals then bw3(G) for each G in Goals
  else if Goals is a fact then succeed
      else begin
          find the clause whose consequent is Goals and
          its antecedent branching factor is minimum;
          if non is found then fail else bw3(Antecedent of the clause)
       end;
```

Implementation

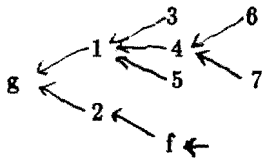
```
% nump creates count database containing the number of procedures for
% each goal in the database.
nump :- clause((G <- Body),_), not(count(G,N)),
      bagof(G,(G <-_),L),length(L,N),assert(count(G,N)),fail.
```

```
% Given conjuctive goals, returns the sum of corresponding counts
% in count database.
sum((B1,B2),P,N) :- count(B1,N1), N2 is N1 + P, sum(B2,N2,N).
sum(B,P,N) :- count(B,N1), N is N1 + P.
sum(_,_,100).
% fm finds the clause whose antecedent has least branching factor
% which it calculated from sum
fm(G) :- clause((G <- B),_),sum(B,0,N),assert(min(B,N)),fm1(G).
fm(_).
fm1(G) :- clause((G <- B),_),
        sum(B,0,N), min(B1,N1), N < N1,
        retract(min(B1,N1)),asserta(min(B,N)),fail.
fm1(_).

% Just like the blind backward chainer except the third clause which
% selects the clause with minimum branching factor instead of
% top to bottom control structure of blind backward chainer.
bw3((G1,G2)) :- bw3(G1),bw(G2).
bw3(G) :- clause((G <- true),_).
bw3(G) :- fm(G),retract(min(Body,_)),bw3(Body).
```

Database:

```
go(g) <- go(1). go(g) <- go(2).
go(1) <- go(3). go(1) <- go(4). go(1) <- go(5).
go(4) <- go(6). go(4) <- go(7).
go(2) <- go(f). go(f) <- true.
```

Pictorial representation of database:



Trace:

1. Heuristic backward search with procedure ordering

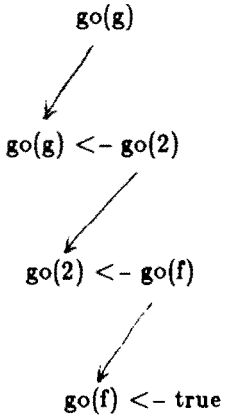
```
: bw3(go(g))!
C'_1 > bw3(go(g))
R'_1 > bw3(go(g))
C''_11 > bw3(go(2))
R''_11 > bw3(go(2))
C'''_111 > bw3(go(f))
E'''_111 < bw3(go(f))
E''_11 < bw3(go(2))
E'_1 < bw3(go(g))
```

2. Blind backward search

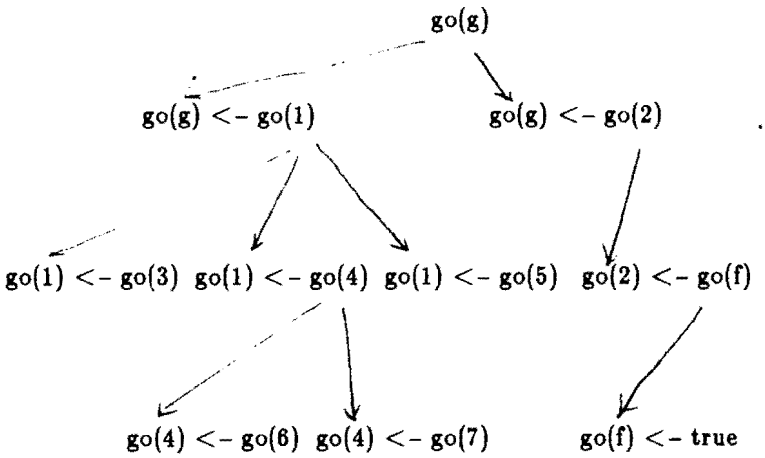
: bw(go(g))!
$C_{|} > bw(go(g))$
$R_{|} > bw(go(g))$
$C_{||} > bw(go(1))$
$R_{||} > bw(go(1))$
$C_{|||} > bw(go(3))$
$R_{|||} > bw(go(3))$
$F_{|||} < bw(go(3))$
$R_{||||} > bw(go(4))$
$R_{||||} > bw(go(4))$
$C_{|||||} > bw(go(6))$
$R_{|||||} > bw(go(6))$
$F_{|||||} < bw(go(6))$
$R_{||||||} > bw(go(7))$
$R_{||||||} > bw(go(7))$
$F_{||||||} < bw(go(7))$
$F_{||||} < bw(go(4))$
$R_{|||||} > bw(go(5))$
$R_{|||||} > bw(go(5))$
$F_{|||||} < bw(go(5))$
$F_{||} < bw(go(1))$
$R_{|||} > bw(go(2))$
$R_{|||} > bw(go(2))$
$C_{||||} > bw(go(f))$
$E_{||||} < bw(go(f))$
$E_{|||} < bw(go(2))$
$E_{|} < bw(go(g))$
:

Proof tree

1. Heuristic backward search with procedure ordering



2. Blind backward search


go(g)
go(g) <- go(1)
go(g) <- go(2)
go(1) <- go(3)
go(1) <- go(4)
go(1) <- go(5)
go(2) <- go(f)
go(4) <- go(6)
go(4) <- go(7)
go(f) <- true

5. Blind Forward search

Synonym for forward chaining is bottom up because in the proof tree, the leaves of the tree are the assertions and the tree is formed bottom up, that is, from assertions to goals. For example, suppose A is an assertion and B <- A. By modus ponens, B could be asserted.

The suggested algorithm exhaustively searches through the universe in breadth first fashion because whenever a new fact is asserted, it is added at the bottom. That is, according Nilsson's terminology, the open list is LIFO. (Note FIFO is for depth first case)

Algorithm:

repeat
  fetch A from known assertions
  For each rules known whose antecedent contains A **do**
    If the antecedent of the rule is satisfiable
      then assert the consequent
    (The antecedent is satisfiable if all of the goals in the antecedent
      are either original or derived assertions.)
until the final goal is in known assertions

Implementation:

```
% member(X,Y) is true if X is a member of Y.
member(X,X).
member(X,(X,_)).
member(X,(_,Y)) :- member(X,Y).

% true if B is a member of Body
superclause(G,B,Body) :- clause((G <- Body),_), member(B,Body).

% final(G) is the ultimate goal to be shown.
% for each known assertions A, call forward with it until either
% no assertions are left to try or the final goal is derived.
fw(G) :-  asserta(final(G)),!,
        clause((A <- true),_),
          forward(A).
% Given a fact, B, find all the rules whose antecedent contains B and assert
% the consequent if the antecedent is satisfiable.

forward(B) :- !,superclause(G,B,Body), satisfiable(Body),
          assert((G <- true)) , final(G).

satisfiable(true):-!.
satisfiable((X,Y)) :- clause((X <- true), ),satisfiable(Y),!.
satisfiable(X) :- clause((X <- true),_).
```

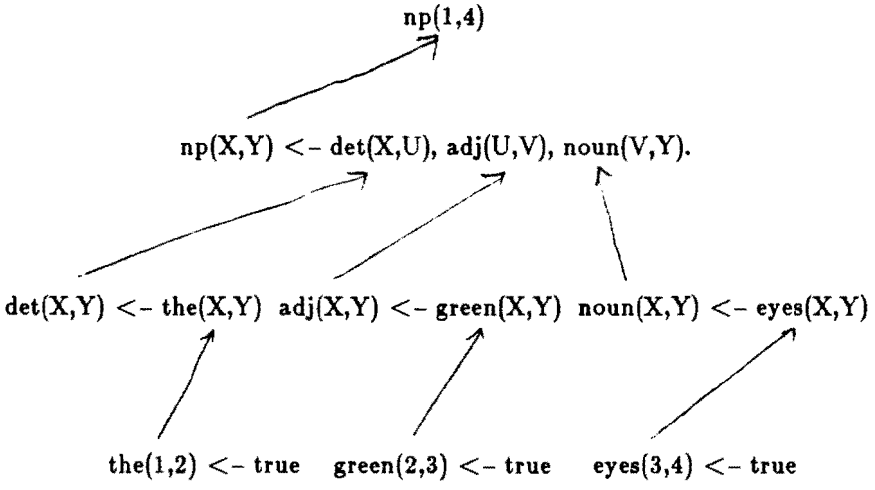
Database:

```
% This example illustrates representing parsing problem by a graph.
% The arcs of the graph are labelled with words occured in the input string
% to be parsed: i.e.
%  1 --the--> 2 --green--> 3 --eyes--> 4
% So if one wants to prove the string,"the green eyes" , is a noun phrase,
% prove np(1,4).
the(1,2) <- true.
green(2,3) <- true.
eyes(3,4) <- true.
det(X,Y) <- the(X,Y).
adj(X,Y) <- green(X,Y).
noun(X,Y) <- eyes(X,Y).
np(X,Y) <- det(X,U),adj(U,V),noun(V,Y).
```

Trace:

$C_{|}^{|} > fw(np(1, 4))$
$C_{||}^{||} > forward(the(1, 2))$
$F_{||}^{||} < forward(the(1, 2))$
$R_{|||}^{|||} > forward(green(2, 3))$
$F_{|||}^{|||} < forward(green(2, 3))$
$R_{||||}^{||||} > forward(eyes(3, 4))$
$F_{||||}^{||||} < forward(eyes(3, 4))$
$R_{|||||}^{|||||} > forward(det(1, 2))$
$E_{|||||}^{|||||} < forward(det(1, 2))$
$E_{|}^{|} < fw(np(1, 4))$
:

Proof tree:



6.

Heuristic Forward search

The purpose of heuristic in searching is to minimize the effort in reaching the final goal state. What is "minimize the effort" in forward proof? Minimize the number of assertions derived in reaching the final goal.

For each assertions in the database, keep track of the number of clauses whose antecedent contains the assertion. Find the assertion whose number is minimum and derive all the assertions possible from it.

Algorithm:

repeat
  find the assertion whose heuristic measure is minimum
  For each rules whose antecedent contains the assertion **do**
    If the antecedent of the rule is satisfiable
      then assert the consequent
until the final goal is in known assertions

Implementation:

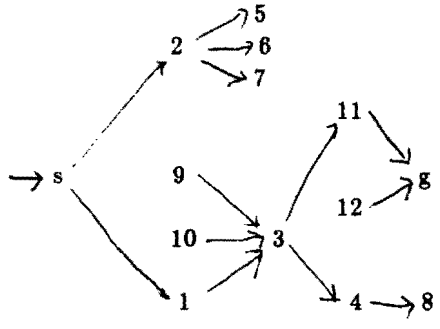
```
% The first argument of countf is the assertion and the second argument
% is the number of clauses whose antecedent contains the assertion.

% initialize countf. If there is no corresponding clauses, set it to
% practically infinity, here 100, because there is no way such fact can
% derive any new assertions.
init :- set,countf(F,0),nb1(F),fail.
init :- retract(countf(F,0)),asserta(countf(F,100)),fail.
init.

% For all the known facts, F, initialize countf to 0.
set :- retractall(countf(_,_)),clause((F <- true),_),assert(countf(F,0)),fail.
set.

% given a fact, F, prepare countf(F,Num) where Num is the number of
% clauses whose antecedent contains F.
nb1(F) :- !,clause((X <- B),_) , member(F,B),
          retract(countf(F,N)),N1 is N +1,
          asserta(countf(F,N1)),fail.
% This procedure is to augment the countf for the new assertions derived
add(F) :- nb1(F).
add(F) :- retract(countf(F,0)),asserta(countf(F,100)),!.
add(_).
% findmin and s2 assert the fact whose corresponding number of clauses is
% minimum among all the facts known at the time of call to findmin in
% min(Fact,Num) as a side effect.
findmin :- countf(F,N),assert(min(F,N)),s2.
s2 :- countf(F,N),min(F1,Min),N < Min,
```

```
    retract(min(F1,Min)),asserta(min(F,N)),fail.
s2.
% after initializing countf and final, repeat the following until the final
% is among the assertions.
% find the fact with minimum number of corresponding clauses.
% increment the number of corresponding clauses by some number, here 1.
% (this is to avoid infinite loop and seems natural way to resist
% backtracking.)
% derive all the assertions derivable from the chosen fact.
fw2(G) :- init,asserta(final(G)),
        repeat,
          findmin,min(Fact,N),retract(countf(Fact,N)),N1 is N+1,
             asserta(countf(Fact,N1)),
          forward2(Fact).

% Given F, derive all the consequents whose antecedent containing
% F are satisfiable.
forward2(F) :- !,superclause(G,F,Body), satisfiable(Body),
             assertz((G <- true)),assert(countf(G,0)),
               add(G),final(G).
```

Database:

```
go(s) <-true.
go(2) <- go(s). go(1) <- go(s).
go(3) <- go(9). go(3) <- go(10). go(3) <- go(1).
go(11) <- go(3). go(4) <- go(3).
go(5) <- go(2). go(6) <- go(2). go(7) <- go(2).
go(g) <- go(11). go(g) <- go(12). go(8) <- go(4).
```

Pictorial representation of the database:



Trace:

```
1. fw2 is a heuristic forward chainer.
: fw2(go(g))!
C|>fw2(go(g))
C||>forward2(go(s))
F||<forward2(go(s))
R|||>forward2(go(1))
F|||<forward2(go(1))
R||||>forward2(go(3))
F||||<forward2(go(3))
```

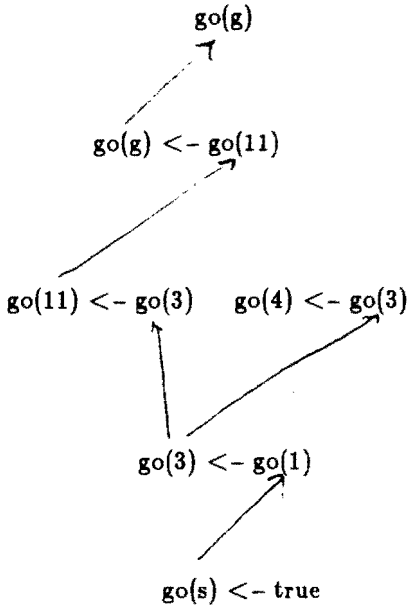
R|||||| >forward2(go(4))
F|||||| <forward2(go(4))
R||||||| >forward2(go(11))
E||||||| <forward2(go(11))
E| <fw2(go(g))

2. fw is a blind forward chainer on the same database

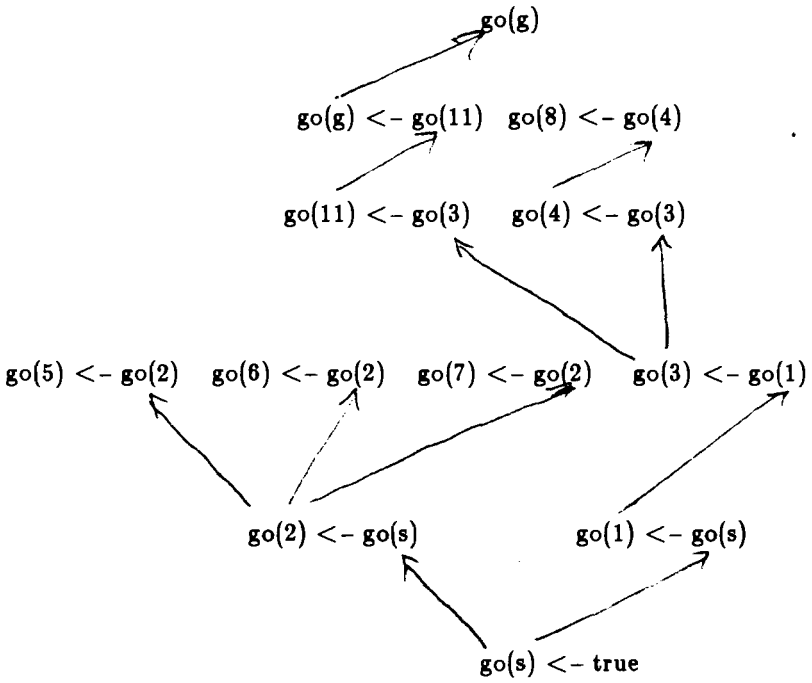
: fw(go(g))!
C| >fw(go(g))
C|| >forward(go(s))
F|| <forward(go(s))
R||| >forward(go(2))
F||| <forward(go(2))
R|||| >forward(go(1))
F|||| <forward(go(1))
R||||| >forward(go(5))
F||||| <forward(go(5))
R|||||| >forward(go(6))
F|||||| <forward(go(6))
R||||||| >forward(go(7))
F||||||| <forward(go(7))
R|||||||| >forward(go(3))
F|||||||| <forward(go(3))
R||||||||| >forward(go(11))
E||||||||| <forward(go(11))
E| <fw(go(g))

Proof tree:

1. Heuristic forward chainer



2. Blind forward chainer


go(g)
go(g) <- go(11)
go(8) <- go(4)
go(11) <- go(3)
go(4) <- go(3)
go(5) <- go(2)
go(6) <- go(2)
go(7) <- go(2)
go(3) <- go(1)
go(2) <- go(s)
go(1) <- go(s)
go(s) <- true

7. Heuristic bidirectional search

As the name implies bidirectional search utilizes both backward and forward search procedures. In order to make use of two different procedures, it needs to be decided when the control is passed to one another. The control of a procedure should be reliquished when the promise of the procedure is no greater than the other procedure. How do we calculate the promise of the procedure? For backward chainer considering a goal, G, the promise is the number of procedures whose consequent matches with G. For forward chainer, the promise is the minimum branching factor among the known assertions. Note that the basic philosophy is to minimize the alternatives considered at any point in the proof. Since the basic algorithms for each backward and forward procedures is explained elsewhere int the report, explanation of the algorithm is redundant and only the implementation is given. And some of the procedures are ommited for the same reason.

Implementation:

```
% bd is for proving G bidirectionally. It calls init to initialize for forward
% chainer, fwd, and calls backward chainer, bw6, first.
bd(G) :- init,bw6(G).

% Just like the procedure using the number of procedures as a heuristic before
% except the third clause. After establishing the current goal, G, as the
% possible goal for forward chaining, call fwd with the number of procedures
% corresponding to G. This will enable the forward
% chainer to terminate before reaching the ultimate final goal as would be the
% case with usual forward chainer.
bw6((G1 , G2)) :- bw6(G1),bw6(G2).
bw6(G) :- clause((G <- true),_).
bw6(G) :- fm(G),  retract(min(Body,N)),   assert(final(G)),
        fwd(N),  bw6(Body).

% The only difference from previous fw2 procedure, this procedure will find
% the fact minimum heuristic measure as long as it is no greater than that
% of the current goal from bw6.
fwd(N) :- findmin,    min(Fact,N1),     N1 <= N,
        retract(countf(Fact,N1)),
        N2 is N1+1,  asserta(countf(Fact,N2)),   forward2(Fact).
fwd(_).
```

Database:

The path finding problem from previous section is used for comparison purpose.

Trace:

1. Heuristic bidirectional search.
: bd(go(g))!
$C^{!}_{!} >$ bw6(go(g))
$C^{!!}_{!!} >$ forward2(go(s))
$F^{!!}_{!!} <$ forward2(go(s))
$R^{!!!}_{!!!} >$ bw6(go(11))
$C^{!!!!}_{!!!!} >$ forward2(go(1))
$F^{!!!!}_{!!!!} <$ forward2(go(1))
$R^{!!!!!}_{!!!!!} >$ bw6(go(3))
$E^{!!!!!}_{!!!!!} <$ bw6(go(3))
$E^{!!!}_{!!!} <$ bw6(go(11))
$E^{!}_{!} <$ bw6(go(g))

2. Heuristic backward search.
: bw5(go(g))!
$C^{!}_{!} >$ bw5(go(g))
$R^{!}_{!} >$ bw5(go(g))
$C^{!!}_{!!} >$ bw5(go(11))
$R^{!!}_{!!} >$ bw5(go(11))
$C^{!!!}_{!!!} >$ bw5(go(3))
$R^{!!!}_{!!!} >$ bw5(go(3))
$C^{!!!!}_{!!!!} >$ bw5(go(9))
$R^{!!!!}_{!!!!} >$ bw5(go(9))
$F^{!!!!}_{!!!!} <$ bw5(go(9))
$R^{!!!!!}_{!!!!!} >$ bw5(go(10))
$R^{!!!!!}_{!!!!!} >$ bw5(go(10))
$F^{!!!!!}_{!!!!!} <$ bw5(go(10))
$R^{!!!!!!}_{!!!!!!} >$ bw5(go(1))
$R^{!!!!!!}_{!!!!!!} >$ bw5(go(1))
$C^{!!!!!!!}_{!!!!!!!} >$ bw5(go(s))
$E^{!!!!!!!}_{!!!!!!!} <$ bw5(go(s))
$E^{!!!!!!}_{!!!!!!} <$ bw5(go(1))
$E^{!!!}_{!!!} <$ bw5(go(3))
$E^{!!}_{!!} <$ bw5(go(11))
$E^{!}_{!} <$ bw5(go(g))

3. Heuristic forward search:

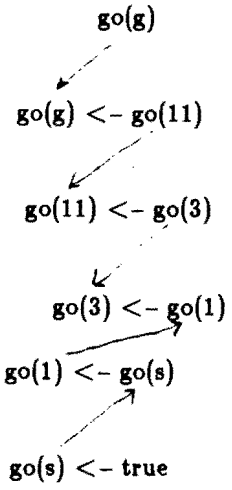
Please refer to the first example in previous section.

4. Blind forward search:

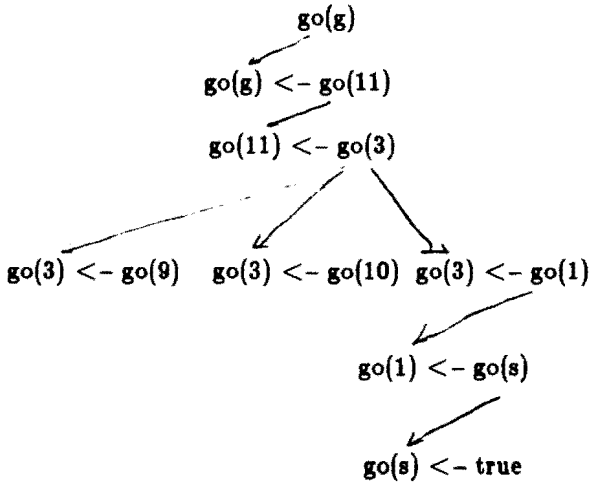
Please refer to the second example in previous section.

Proof tree:

1. Heuristic bidirectional search:



2. backward search procedure:



8.

Conclusion

I hope that the reader is convinced that prolog can be used as a general purpose programming language by now even though it is cumbersome to do some simple task like loops. Even though prolog is a functional programming language, some of the side effect primitives like assert and retract were invaluable in my project. It is very difficult to predict the effect of using both backtracking and assert or retract even though there are situations one should not avoid using such facility. In section 3, the procedure, chsg, to find the goal with maximum instantiation, is unnecessarily ugly and inefficient. At that time, side effect was avoided at all costs. Now, I would write the code which would have given the desired goal in special predicate,max, as a side effect:

```
chsg((G1,G2)) :- num(G1,N1), max(G,N),
                 N1 > N,
                 retract(max(_,_)), assert(max(G1,N1)), chsg(G2).
chsg((G1,G2)) :- chsg(G2).
chsg(G1)      :- num(G1,N1), max(G,N),
                 N1 > N,
                 retract(max(_,_)), assert(max(G1,N1)).
chsg(_).
```

It is evident the above procedure is more elegant. What is less obvious is that above procedure is space efficient. Previously, there were 6 arguments and four of them were redundant in a sense that they all shared the same goals as the first argument. Since space efficiency was achieved through minimal processing (retract and assert as compared to appending), time efficiency is also apparent. In short, constrained use of side effect may yield both efficient and elegant code.

Another note of technical kind is that when using a procedure for side effects, be sure to include false drop like the last clause in the above program. Since these procedures are called most likely in the middle of the antecedent, the call should succeed always so that the rest of the goals which might use the produced side effect is continued.

Finally, it would be fruitfull to combine forward and backward chainers as follows. For backward chainer, introduce goal ordering with the degree of instantiation of its arguments and procedure ordering with estimated number of branching factor among the clauses in the procedure (Note the procedure is a

group of clauses which share the same consequent). For forward chainer, choose the assertion which is likely to produce least number of clauses. Since the new assertions produced becomes indistinguishable from the original, it is possible the newly produced assertion is the very next assertion chosen in the next iteration. That is, the probe into search space is slightly more depth oriented. The switching condition between forward and backward chainer is which has fewer branching factor.