Chapter 7: Controlling Search

Where we discuss how to make the search for a solution more efficient

1

Controlling Search

- Estimating Efficiency of a CLP Program
- Rule Ordering
- Literal Ordering
- Adding Redundant Constraints
- Minimization
- Identifying Deterministic Subgoals
- Example: Bridge Building

2

Estimating Efficiency

- Evaluation is a search of the derivation tree
- Size and shape of derivation tree determines efficiency (ignores solving cost)
  - smaller: less search
  - answers in the leftmost part: less search before first answer
- Derivation tree depends on the *mode of usage*

3

Mode of Usage

- **mode of usage**: defines the kinds of constraints on the argument of a predicate when evaluated
- *fixed*: constraint store implies a single value in all solutions
- *free*: constraint store allows all values
- others: bounded above, bounded below

4

Mode of Usage Example

```
sumlist([], 0).
sumlist([N|L], N+S) :- sumlist(L, S).
```

- mode of usage first arg *fixed* second *free*
  - `sumlist([1],S).`
  - `L=[1,2],S > Z, sumlist(L,S).`
- states in derivation tree with sumlist called
  - `< sumlist([1], S) | true >`
  - `< sumlist(L',S') | [1]=[N'|L'] /\ S = N' + S' >`

5

Controlling Search Example

- Imagine writing a program to compute
  $$S = 0 + 1 + 2 + \cdots N$$
- Reason recursively:
  - sum of *N* numbers is sum of *N-1* + *N*
  - sum of 0 numbers is 0

```
(S1) sum(N, S+N) :- sum(N-1, S).
(S2) sum(0, 0).
```

- Problem sum(1,S) doesnt answer

6

Controlling Search Example

$$
\begin{array}{ll}
\langle sum(1,S) \mid true \rangle & \\
\Downarrow S1 & \Downarrow S2 \\
\langle sum(0,S') \mid S = 1 + S' \rangle & \langle [] \mid false \rangle \\
\Downarrow S1 & \Downarrow S2 \\
\langle sum(-1,S'') \mid S = 1 + S'' \rangle & \langle [] \mid S = 1 \rangle \\
\Downarrow S1 & \Downarrow S2 \\
\langle sum(-2,S''') \mid S = 0 + S''' \rangle & \langle [] \mid false \rangle \\
\Downarrow S1 &
\end{array}
$$

Simplified derivation tree for sum(1,S)

7

Controlling Search Example

- Infinite derivation before answer

```
(S3) sum(0, 0).
(S4) sum(N, S+N) :- sum(N-1, S).
```

- sum(1,S) answers *S=1*,
- but sum(1,0)?

$$
\begin{array}{c}
\langle sum(1,0) \mid true \rangle \\
\Downarrow S4 \\
\langle sum(0,-1) \mid true \rangle \\
\Downarrow S4 \\
\langle sum(-1,-1) \mid true \rangle \\
\Downarrow S4 \\
\langle sum(-2,0) \mid true \rangle \\
\Downarrow S4
\end{array}
$$

8

Controlling Search Example

- Program was not intended to work for negative numbers. Correct it

```
(S5) sum(0, 0).
(S6) sum(N, S+N) :- sum(N-1, S), N >= 1.
```

$$\langle sum(1,0) | true \rangle$$
$$\Downarrow S6$$
$$\langle sum(0,-1), 0 \geq 1 | true \rangle$$
$$\Downarrow S6$$
$$\langle sum(-1,-1), 0 \geq 1, -1 \geq 1 | true \rangle$$
$$\Downarrow S6$$

9

Controlling Search Example

- Remember left to right processing

```
(S7) sum(0, 0).
(S8) sum(N, S+N) :- N >= 1, sum(N-1, S).
```

- sum(1,S) gives *S* = , sum(1,0) answers *no*
- Methods:
  - rule reordering
  - adding redundant constraints
  - literal reordering

10

Rule Ordering

- general rule
  - place non-recursive rules before recursive rules
  - (this will tend to avoid infinite derivations)
- heuristic rule
  - place rules which are “more likely to lead to answers” before others
  - (tend to move the success to the left of tree)

11

Literal Ordering

- Primitive Constraints
  - place a constraint at the earliest point in which it could cause failure (for mode of usage)
- *fac(N,F)* with *N* fixed and *F* free

```
fac(N, F) :- N = 0,F = 1.
fac(N, FF) :- N >= 1, FF = N *F,
     N1 = N - 1, fac(N, F).

fac(N, F) :- N = 0,F = 1.
fac(N, FF) :- N >= 1,  N1 = N - 1,
     fac(N, F), FF = N * F.
```

12

Literal Ordering

- User-define constraints:
  - place deterministic literals before others
- **deterministic**: *p(s1,...,sn)* in program is deterministic for a derivation tree if at each choicepoint where it is rewritten all but one derivation fails before rewriting a user-defined constraint (at most one succeeds)
- deterministic predicate *p* for mode of usage

13

Deterministic Predicates

- *sumlist(L,S)* is deterministic for mode of usage *L* fixed *S* free. Not for *L* free *S* fixed.
- *sum(N,S)* is similar
- deterministic predicates require little search to find an answer
- BEWARE moving a predicate can change whether it is deterministic or not

14

Literal Reordering Example

```
father(jim,edward).      mother(maggy,fi).
father(jim,maggy).       mother(fi,lillian).
father(edward,peter).
father(edward,helen).
father(edward,kitty).
father(bill,fi).
```

father(F,C) is deterministic with *C* fixed *F* free, but not with both free of *F* fixed and *C* free. *mother(M,C)* also

Every child can only have one father

A father can have many children

15

Literal Reordering Example

```
grandf(GF,GC) :- father(GF,P),father(P,GC).
grandf(GF,GC) :- father(GF,P),mother(P,GC).
```

For mode of usage *GC* fixed *GF* free:

- What modes of usage for first rule literals?
- *father(GF,P)* both free, *father(P,GC)* both fixed
- What is the body literals are reversed?
- *father(P,GC)* free fixed, *father(P,GC)* free fixed

16

Literal Reordering Example

```
grandf(GF,GC) :- father(P,GC),father(GF,P).
grandf(GF,GC) :- mother(P,GC),father(GF,P).
```

More efficient for mode of usage free fixed

e.g. grandf(X,peter)

63 states in simplified derivation tree for first prog

versus23 states for second prog.

17

Adding Redundant Cons.

- A constraint that can be removed from a rule without changing the answers is **redundant**.
- **answer redundant**: same set of answers for
  - H :- L1, ..., Li, Li+1, ..., Ln
  - H :- L1, ..., Li, c, Li+1, ..., Ln
- advantage (for store *C* in mode of usage)
  - *<L1,...,Li,c|C>* fails but not *<L1,...,Li|C>*

18

Adding Redundant Cons.

The constraint *N >= 1* added to the sum program was answer redundant!

Another example *sum(N,7)* (new mode of usage)

$$
\begin{array}{c}
\langle sum(N,7) \mid true \rangle \\
\Downarrow S8 \\
\langle sum(N',S') \mid N = N'+1 \wedge S' = 6 - N' \wedge N' \geq 0 \rangle \\
\Downarrow S8 \\
\langle sum(N'',S'') \mid N = N''+2 \wedge S'' = 4 - 2N'' \wedge N'' \geq 0 \rangle \\
\Downarrow S8 \\
\langle sum(N''',S''') \mid N = N'''+3 \wedge S''' = 1 - 3N''' \wedge N''' \geq 0 \rangle \\
\Downarrow S8 \\
\langle sum(N'''',S'''') \mid N = N''''+4 \wedge S'''' = -3 - 4N'''' \wedge N'''' \geq 0 \rangle
\end{array}
$$

19

Adding Redundant Cons.

We know each sum of number is non-negative

```
(S9)  sum(0, 0).
(S10) sum(N, S+N) :-
          N >= 1, S >= 0, sum(N-1, S).
```

$$
\begin{array}{c}
\langle sum(N,7) \mid true \rangle \\
\Downarrow S10 \\
\langle sum(N',S') \mid N = N'+1 \wedge S' = 6 - N' \wedge N' \geq 0 \wedge N' \leq 6 \rangle \\
\Downarrow S10 \\
\langle sum(N'',S'') \mid N = N''+2 \wedge S'' = 4 - 2N'' \wedge N'' \geq 0 \wedge N'' \leq 2 \rangle \\
\Downarrow S10 \\
\langle sum(N''',S''') \mid N = N'''+3 \wedge S''' = 1 - 3N''' \wedge N''' \geq 0 \wedge N''' \leq 1/3 \rangle \\
\Downarrow S10 \\
\langle [] \mid false \rangle
\end{array}
$$

20

Solver Redundant Constraints

- **solver redundant:** a primitive constraint *c* is solver redundant if it is implied by the constraint store
- advantages: if solver is partial can add extra information (failure)
- `F >= 1, N >= 1, FF = N*F, FF >= 1`

21

Solver Redundant Example

```
(F1) fac(N, F) :- N = 0,F = 1.
(F2) fac(N, FF) :- N >= 1,  N1 = N - 1,
            FF = N * F, fac(N, F).
```

Goal *fac(N,7)* runs forever like *sum(N,7).*

```
(F3) fac(N, F) :- N = 0,F = 1.
(F4) fac(N, FF) :- N >= 1,  N1 = N - 1,
            FF = N * F, F >= 1, fac(N, F).
```

Goal *fac(N,7)* still runs forever !

22

Solver Redundant Example

$$
\begin{array}{c}
\langle fac(N,7) \mid true \rangle \\
\Downarrow F4 \\
\langle fac(N-1, F') \mid F' \geq 1 \wedge N \geq 1 \wedge 7 = N \times F' \rangle \\
\Downarrow F4 \\
\langle fac(N-2, F'') \mid F'' \geq 1 \wedge N \geq 2 \wedge 7 = N \times (N-1) \times F'' \rangle \\
\Downarrow F4 \\
\langle fac(N-3, F''') \mid F''' \geq 1 \wedge N \geq 3 \wedge 7 = N \times (N-1) \times (N-2) \times F''' \rangle \\
\Downarrow F4 \\
\langle fac(N-4, F'''') \mid F'''' \geq 1 \wedge N \geq 4 \wedge 7 = N \times (N-1) \times (N-2) \times (N-3) \times F'''' \rangle
\end{array}
$$

Given that *F’’’’ >= 1* and *N >= 4* then

$$N \times (N-1) \times (N-2) \times (N-3) \times F''''$$

must be at least 24. constraint is unsatisfiable not detected (partial solver)

23

Solver Redundant Example

Fix: add solver redundant constraint *N * F >= N*

is implied by *N >= 1, F >= 1*

CAREFUL: *1 = N * F , 2 = N * F* succeeds, therefore use the same name for each *N*F*

```
(F3) fac(N, F) :- N = 0,F = 1.
(F4) fac(N, FF) :- N >= 1,  N1 = N - 1,
           FF = N * F, FF >= N, F >= 1,
           fac(N, F).
```

Now the goal *fac(N,7)* finitely fails

24

Minimization

- Minimization literals cause another derivation tree to be searched
- Need to understand the form of this tree
- `minimize(G,E)` has mode of usage the same as `E < m, G`
- For efficient minimization, ensure that *G* is efficient when *E* is bounded above

25

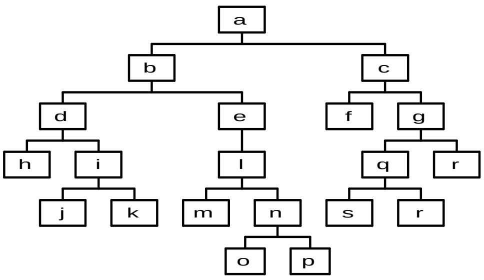
Minimization Example

Program which finds leafs and their level (depth)

```
leaf(node(null,X,null),X,0).
leaf(node(L,_,_),X,D+1) :- leaf(L,X,D).
leaf(node(_,_,R),X,D+1) :- leaf(R,X,D).
```

Answers: X=h/\D=3

(h,3),(j,4),(k,4),(m,4),
(o,5),(p,5),(f,2),(s,4),
(t,4),(r,3)

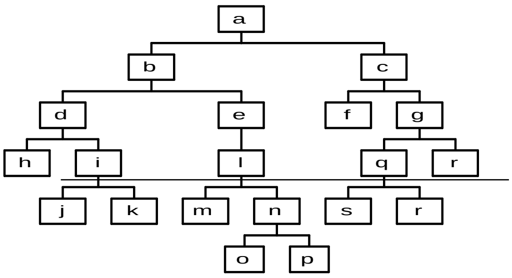


26

Minimization Example

Goal `minimize(leaf(t(a),X,D), D)`:
After finding $X = h \wedge D = 3$, acts like
`D < 3 leaf(t(a),X,D)`, should never visit nodes below depth 3

$$\langle D < 3, leaf(t(a), X, D) | true \rangle$$
$$\Downarrow$$
$$\langle leaf(t(a), X, D) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(b), X, D-1) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(d), X, D-2) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(i), X, D-3) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(k), X, D-4) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle [] | false \rangle$$



27

Minimization Example

Improve `leaf` for mode of usage D bounded above: add an answer redundant constraint

```
leaf(node(null,X,null),X,0).
leaf(node(L,_,_),X,D+1) :-
     D >= 0, leaf(L,X,D).
leaf(node(_,_,R),X,D+1) :-
     D >= 0, leaf(R,X,D).
```

$$\langle D < 3, leaf(t(a), X, D) | true \rangle$$
$$\Downarrow$$
$$\langle leaf(t(a), X, D) | D < 3 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(b), X, D-1) | D < 3 \wedge D \geq 1 \rangle$$
$$\Downarrow$$
$$\langle leaf(t(d), X, D-2) | D < 3 \wedge D \geq 2 \rangle$$
$$\Downarrow$$
$$\langle [] | false \rangle$$

28

Minimization

- The search may not always benefit from the bounds
  - e.g. `minimize(leaf(t(a),X,D), -D)`
  - must still visit every node after finding one leaf
  - arguably the original formulation is better since it involves less constraints
- Key: remember the mode of usage $E<m, G$

29

Identifying Determinism

- CLP languages involve constructs so that the user can identify deterministic code so that the system can execute it efficiently
- if-then-else literals
- once literals

30

If-Then-Else

- **if-then-else** literal: (*Gtest* -> *Gthen* ; *Gelse*)
- first test the goal *Gtest*, if it succeeds execute *Gthen* otherwise execute *Gelse*
- **if-then-else derivation step**: *G1* is *L1, L2, ..., Lm, where L1* is (Gt -> Gn ; Ge)
  - **if** <*Gt* | *C1*> succeeds with leftmost successful derivation <*Gt* | *C1*> => ... => < *[]* | *C*>
  - *C2* is *C, G2* is *Gn, L2, ..., Lm*
  - **else** *C2* is *C1, G2* is *Ge, L2, ..., Lm*

31

If-Then-Else Example

- `abs(X,Y) :- (X >= 0, Y = X ; Y = -X).`
- if *X* is pos abs value is *X*, otherwise -*X*

$$\langle abs(4,A) | true \rangle$$
$$\Downarrow$$
$$\langle (X \geq 0 \rightarrow Y = X ; Y = -X) | 4 = X \wedge A = Y \rangle$$
$$\Downarrow$$
$$\langle Y = X | 4 = X \wedge A = Y \wedge X \geq 0 \rangle$$
$$\Downarrow$$
$$\langle [] | 4 = X \wedge A = Y \wedge X \geq 0 \wedge Y = X \rangle$$

$$\langle X \geq 0 | 4 = X \wedge A = Y \rangle$$
$$\Downarrow$$
$$\langle [] | 4 = X \wedge A = Y \wedge X \geq 0 \rangle$$

$$\langle abs(-4,A) | true \rangle$$
$$\Downarrow$$
$$\langle (X \geq 0 \rightarrow Y = X ; Y = -X) | -4 = X \wedge A = Y \rangle$$
$$\Downarrow$$
$$\langle Y = -X | -4 = X \wedge A = Y \wedge X \geq 0 \rangle$$
$$\Downarrow$$
$$\langle [] | -4 = X \wedge A = Y \wedge X \geq 0 \wedge Y = -X \rangle$$

$$\langle X \geq 0 | -4 = X \wedge A = Y \rangle$$
$$\Downarrow$$
$$\langle [] | false \rangle$$

32

If-Then-Else Example

- What happens to the goals
  - `abs(X,2), X < 0` and `X < 0, abs(X,2)`

    fails ?! succeeds *X* = *-2* ?

DANGERS

- answers strongly depend on mode of usage
- only the first answer of the test goal is used

33

If-Then-Else Examples

```
far_eq(X,Y) :- (apart(X,Y,4)-> true ; X = Y).
apart(X,Y,D) :- X >= Y + D.
apart(X,Y,D) :- Y >= X + D.
```

X and *Y* are equal or at least 4 apart

- `far_eq(1,6)` succeeds, `far_eq(1,3)` fails
- `far_eq(1,Y), Y = 6` fails
- WHY? test goal commits to first answer $X >= Y + 4$

34

If-Then-Else

- **safe usage**: the mode of usage makes all variables in *Gtest* fixed
- example: safe when *N* and *P0* fixed

```
cumul_pred([],_,P,P).
cumul_pred([N|Ns],D,P0,P) :-
     (member(N,P0) ->
          P1 = P0
     ;
          pred(N,D,[N|P0],P1)
     ),
     cumul_pred(Ns,D,P1,P).
```

35

Once

- **once** literal: once(*G*)
- find only the first solution for *G*
- **once derivation step**: *G1* is *L1, L2, ..., Lm, where L1* is once(*G*)
  - **if** <*G* | *C1*> succeeds with leftmost successful derivation <*G* | *C1*> => ... => < *[]* | *C*>
  - *C2* is *C*, *G2* is *L2, ..., Lm*
  - **else** *C2* is *false*, *G2* is *[]*

36

Once Example

- Sometimes all answers are equivalent
- example: intersection

```
intersect(L1,L2) :-
     member(X,L1), member(X,L2).
```

- intersect([a,b,e,g,h],[b,e,f,g,,i]) 72 states

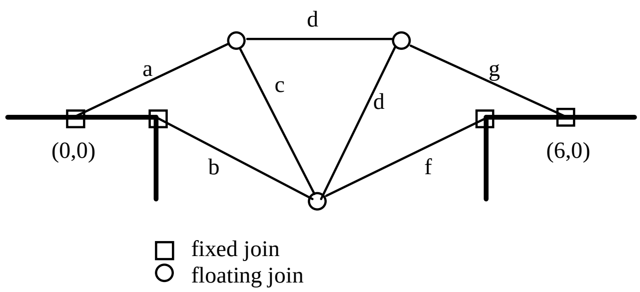
```
intersect(L1,L2) :-
     once(member(X,L1), member(X,L2)).
```

- 18 states

37

Bridge Building Example

- AIM: build 2 dimensional spaghetti bridges
- Approach: first build a program to analyze bridges, then use it constrain designs



38

Bridge Building Example

- Constraints:
  - 20cm of struts,
  - strut of length L can sustain any stretch, only 0.5*(6-L)N compression,
  - floating joins can sustain any stretch, only 2Ncompression, sum of forces at a floating join is zero, once join in the center, at least 3 incident struts to a join, except center join which needs 2

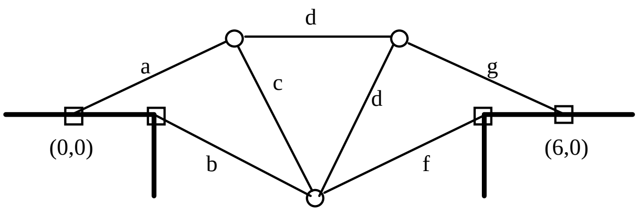
39

Representing Bridges

- list of joins
  - *cjoin(x,y,l)* (xy coords, list of incident struts)
  - *join(x,y,l)*
- list of struts: *strut(n,x1,y1,x2,y2)* name and coords of endpoints
- analysis of the bridge will create an association list of stretching forces in each strut *f(n,f)*

40

Representing Bridges



js = [join(2,1,[a,c,d]), join(4,1,[d,e,g]),
    cjoin(3,-1,[b,c,e,f])]
ss = [strut(a,0,0,2,1), strut(b,1,0,3,-1), strut(c,2,1,3,-1),
    strut(d,2,1,4,1), strut(e,3,-1,4,1),
    strut(g,4,1,6,0)]

41

Strut Constraints

```
strutc([],[],0).
strutc([strut(N,X1,Y1,X2,Y2)|Ss],
       [f(N,F)              |Fs], TL):-
     L = sqrt((X1-X2)*(X1-X2)+
              (Y1-Y2)*(Y1-Y2)),
     F >= -0.5 * (6 - L),
     TL = L + RL,
     strutc(Ss, Fs, RL).
```

Builds force association list, calculates total length, asserts max compression force

42

Strut Constraints

- Given a fixed list of struts works well
- Like sum total length only causes failure at end
  - FIX add answer redundant constraint *RL >= 0*
- If the coords of the struts are not fixed length calculation is non-linear (incomplete)
  - (partial) FIX add solver redundant constraints (linear approximation)

$$L \geq X1 - X2 \wedge L \geq X2 - X1 \wedge L \geq Y1 - Y2 \wedge L \geq Y2 - Y1$$

43

Summing Forces

```
sumf([],_,_,_,0,0).
sumf([N|Ns],X,Y,Ss,Fs,SFX,SFY) :-
     member(strut(N,X1,Y1,X2,Y2),Ss),
     end(X1,Y1,X2,Y2,X,Y,X0,Y0),
     member(f(N,F),Fs), F <= 2,
     L = sqrt((X1-X2)*(X1-X2)+
              (Y1-Y2)*(Y1-Y2)),
     FX = F*(X-X0)/L, FY = F*(Y-Y0)/L,
     SFX = FX+RFX, SFY = FY+RFY,
     sumf(Ns,X,Y,Ss,Fs,RFX,RFY).
end(X,Y,X0,Y0,X,Y,X0,Y0).
end(X0,Y0,X,Y,X,Y,X0,Y0).
```

44

Join Constraints

```
joinc([],_,_,_).
joinc([J|Js],Ss,Fs,W) :-
     onejoin(J,Ss,Fs,W).
     joinc(Js,Ss,Fs,W).
onejoin(cjoin(X,Y,Ns),Ss,Fs,W) :-
     Ns = [_,_|_],
     sumf(Ns,X,Y,Ss,Fs,0,W).
onejoin(join(X,Y,Ns),Ss,Fs,W) :-
     Ns = [_,_,_|_],
     sumf(Ns,X,Y,Ss,Fs,0,0).
```

Apply minimum incident struts and sum forces cons.

45

Join Constraints

- Given a fixed list of struts for each join, works well
- non-deterministic because of `end` although there is only one answer
- hence use inside once

46

Bridge Analysis

- For the illustrated bridge

```
TL <= 20,
strutc(ss,Fs,TL),
once(joinc(js,ss,Fs,W).
```

- Answer is *W <= 2.63*

47

Bridge Design

- `strutc` and `joinc` require the topology to be known to avoid infinite derivations
- too many topologies to search all
- one approach user defines topology `tpl(Js,Ss,Vs)` where Vs are the coordinate variables
- system performs minimization search

48

Bridge Design

- Unfortunately constraints are nonlinear so minimization goal will not work
- instead add explicit search to minimize on which fixes all coordinates

```
tpl(Js,Ss,Vs), TL <= 20,
strutc(Ss,Fs,TL),
once(joinc(Js,Ss,Fs,W)),
minimize(position(Vs), -W).
```

- Answer *W=6.15 /\ Vs=[2,2,5,1,3,3]*

49

Bridge Design

- Integer coordinates are very restrictive
- Idea: use local search to improve the design
  - find an optimal (integer) solution
  - try moving coordinate + or - 0.5 for better sol
  - if so then try +/- 0.25 etc. until solution doesnt improve very much
- Best local search answer
  - *W=6.64 /\ Vs=[2.125,2.625,3.875,2.635,3,3.75*]

50

Controlling Search Summary

- Efficiency is measured as size of derivation tree
- Depends on the mode of usage of predicates
- Change size and shape by reordering literals and rules (doesnt change answers)
- Add redundant constraints to prune branches (doesnt change answers)
- Use if-then-else and once to identify sub-computations which dont need backtracking