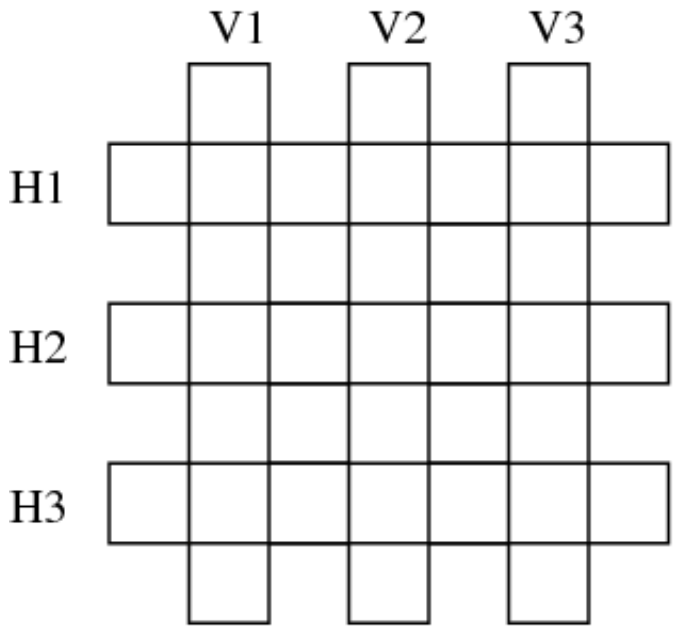
Logic Programming
(PLP 11, CTM 9.2, 9.4, 12.1-12.2)
Constraint Satisfaction Problems,
Natural Language Parsing (Definite Clause Grammars)

Carlos Varela
Rensselaer Polytechnic Institute

November 19, 2024

Constraint Satisfaction Example*

- Given six Italian words:
  - astante, astoria, baratto, cobalto, pistola, statale.
- They are to be arranged, crossword puzzle fashion, in the following grid:



Constraint Satisfaction Example(2)*

- The following knowledge base represents a lexicon containing these words:

```
word(astante,  a,s,t,a,n,t,e).
word(astoria,  a,s,t,o,r,i,a).
word(baratto,  b,a,r,a,t,t,o).
word(cobalto,  c,o,b,a,l,t,o).
word(pistola,  p,i,s,t,o,l,a).
word(statale,  s,t,a,t,a,l,e).
```

- Write a predicate `crossword/6` that tells us how to fill in the puzzle. The first three arguments should be the vertical words from left to right, and the last three arguments the horizontal words from top to bottom.

Constraint Satisfaction Example(3)*

- Try solving it yourself before looking at this solution!

```
crossword(V1,V2,V3,H1,H2,H3) :-
      word(V1,_,H1V1,_,H2V1,_,H3V1,_),
      word(V2,_,H1V2,_,H2V2,_,H3V2,_),
      word(V3,_,H1V3,_,H2V3,_,H3V3,_),
      word(H1,_,H1V1,_,H1V2,_,H1V3,_),
      word(H2,_,H2V1,_,H2V2,_,H2V3,_),
      word(H3,_,H3V1,_,H3V2,_,H3V3,_).
```

Generate and Test Example: Finding digit pairs that add to 10

- Using generate and test to do combinatorial search:

```
digit(D) :- between(0,9,D).

pairAdd10(D1,D2) :-
      digit(D1),
      digit(D2),
      D1 + D2 =:= 10.

allPairs(L) :-
     findall(p(D1,D2),pairAdd10(D1,D2),L).
```

Finding palindromes

- Find all four-digit palindromes that are products of two-digit numbers:

```
palindrome(S) :-
    digit(D1), digit(D2), digit(D3), digit(D4),      % generate
    S is (10*D1+D2)*(10*D3+D4),
    S >= 1000,                                       % test
    mod(div(S,1000),10) =:= mod(S,10),               % 1st = 4th
    mod(div(S,100),10) =:= mod(div(S,10),10).        % 2nd = 3rd

allPalindromes(S,L) :- findall(P,palindrome(P),S),length(S,L).
```

Propagate and Search

- The *generate and test* programming pattern can be very inefficient (e.g., Palindrome program explores 10000 possibilities).

- An alternative is to use a *propagate and search* technique.

  Propagate and search filters possibilities during the generation process, to prevent combinatorial explosion when possible.

Propagate and Search

Propagate and search approach is based on three key ideas:

•*Keep partial information*, e.g., "in any solution, X is greater than 100".

•*Use local deduction*, e.g., combining "X is less than Y" and "X is greater than 100", we can deduce "Y is greater than 101" (assuming Y is an integer.)

•*Do controlled search*. When no more deductions can be done, then search. Divide original CSP problem P into two new problems: (P ^ C) and (P ^ ¬C) and where C is a new constraint. The solution to P is the union of the two new sub-problems. Choice of C can significantly affect search space.

Propagate and Search Example

- Find two digits that add to 10, multiply to more than 24:

```
:- use_module(library(clpfd)).
q(D1,D2) :-
    D1 in 0..9, D2 in 0..9,     % initial constraints
    D1+D2 #= 10,     % D1 and D2 cannot be 0.
                     % reduces search space from 100 to 81 possibilities
    D1*D2 #>= 24,    % D1 and D2 must be between 4 and 6.
                     % reduces search space to 9 possibilities
    D1 #< D2.        % D1 must be 4 or 5 and D2 must be 5 or 6.
                     % reduces search space to 4 possibilities
                     % It does not find unique solution D1=4 and D2=6.
```

Propagate and Search Example(2)

- Find a rectangle whose perimeter is 20, whose area is greater than or equal to 24, and width less than height:

```
rectangle([W,H]) :-
    W in 0..9, H in 0..9,
    W+H #= 10,
    W*H #>= 24,
    W #< H.
rectangleSolve(rect(W,H)) :-
    rectangle([W,H]),
    label([W,H]).
?- rectangleSolve(S).
S = rect(4, 6).
```

Finding palindromes (revisited)

- Find all four-digit palindromes that are products of two-digit numbers:

```
palindrome(A,B,C,X,Y) :-
    A in 1000..9999, B in 0..99, C in 0..99,
    A #= B * C,
    X in 1..9, Y in 0..9,
    A #= X*1000+Y*100+Y*10+X.

palindromeSolve(A) :-
    palindrome(A,_,_,X,Y),
    labeling([ff], [X,Y]).                    % 36 solutions
```

Natural Language Parsing

(Example from "Learn Prolog Now!" Online Tutorial)

```
word(article,a).
word(article,every).
word(noun,criminal).
word(noun,'big kahuna burger').
word(verb,eats).
word(verb,likes).

sentence(Word1,Word2,Word3,Word4,Word5) :-
        word(article,Word1),
        word(noun,Word2),
        word(verb,Word3),
        word(article,Word4),
        word(noun,Word5).
```

Parsing natural language

- *Definite Clause Grammars (DCG)* are useful for natural language parsing.

- Prolog can load DCG rules and convert them automatically to Prolog parsing rules.

DCG Syntax

`-->`

DCG *operator*, e.g.,

```
sentence-->subject,verb,object.
```

Each goal is assumed to refer to the *head* of a DCG rule.

`{prolog_code}`

Include Prolog code in generated parser, e.g.,

```
subject-->modifier,noun,{write('subject')}.
```

`[terminal_symbol]`

Terminal symbols of the grammar, e.g.,

```
noun-->[cat].
```

Natural Language Parsing

(example rewritten using DCG)

```
sentence --> article, noun, verb, article, noun.

article --> [a] | [every].

noun --> [criminal] | ['big kahuna burger'].

verb --> [eats] | [likes].
```

Natural Language Parsing (2)

(example rewritten using DCG)

Let us look at Prolog's generated Horn clause for the `sentence` non-terminal:

```
?- listing(sentence).
sentence(A, F) :-
        article(A, B),
        noun(B, C),
        verb(C, D),
        article(D, E),
        noun(E, F).
```

A-F is a *difference list*. B, C, D, and E are *accumulators*. Sample usage:

```
?- sentence([a,criminal,likes,every,'big kahuna burger'],[]).
true
```

Natural Language Parsing (3)

(example rewritten using DCG)

Now, let us look at Prolog's generated Horn clause for the `verb` non-terminal:

```
?- listing(verb).
verb(A, B) :-
        (   A=[eats|B]
        ;   A=[likes|B]
        ).
```

A-B is a *difference list*. Sample usage:

```
?- verb([likes],[]).
true.

?- verb([likes,cats],[cats]).
true.
```

Natural Language Parsing and Information Extraction

```
sentence(V) --> subject, verb(V), subject.
sentence(V) --> subject, verb(V).

subject --> article, noun.

article --> [a] | [every].

noun --> [criminal]
         | ['big kahuna burger']
         | [dog].

verb(eats) --> [eats].
verb(likes) --> [likes].
```

Natural Language Parsing and Information Extraction

Prolog's generated Horn clauses for the `sentence` non-terminal:

```
?- listing(sentence).
sentence(B, A, E) :-
        subject(A, C),
        verb(B, C, D),
        subject(D, E).
sentence(B, A, D) :-
        subject(A, C),
        verb(B, C, D).
```

A-E and A-D are *difference lists*. B is the extracted information (which could be a parse tree). C and D are *accumulators*. Sample usage:

```
?- sentence(Verb, [a,dog,eats],[]).
Verb = eats.
```

Natural Language Parsing and Information Extraction

Now, let us look at Prolog's generated Horn clause for the `verb` non-terminal:

```
?- listing(verb).
verb(eats, [eats|A], A).
verb(likes, [likes|A], A).
```

Sample usage:

```
?-  verb(Verb, [eats], []).
Verb = eats.

?- verb(Verb,S,T).
Verb = eats,
S = [eats|T] ;
Verb = likes,
S = [likes|T].
```

Exercises

77. How would you translate DCG rules into Prolog rules?
78. PLP Exercise 11.8 (pg 571).
79. PLP Exercise 11.14 (pg 572).
80. CTM Exercise 12.6.2 (pg 774). Solve it in Prolog.