Programowanie w Logice

Poszukiwanie rozwiązań

Przemysław Kobylański

Poszukiwanie rozwiązań

Generowanie wszystkich rozwiązań

- Prolog nie tylko potrafi **sprawdzić** czy dana spełnia warunek ale również potrafi **wygenerować wszystkie** dane spełniające warunek.
- Jest to możliwe dzięki temu, że Prolog wyraża **relacje** a nie tylko **funkcje** (dopuszcza dowolne przepływy danych).

```
kwadrat(0,  0).
kwadrat(1,  1).
kwadrat(2,  4).
kwadrat(3,  9).
kwadrat(4, 16).
```

```
?- kwadrat(3, X).
X = 9.

?- kwadrat(X, 16).
X = 4.
```

```
?- kwadrat(X, X).
X = 0 ;
X = 1 ;
false
```

Poszukiwanie rozwiązań

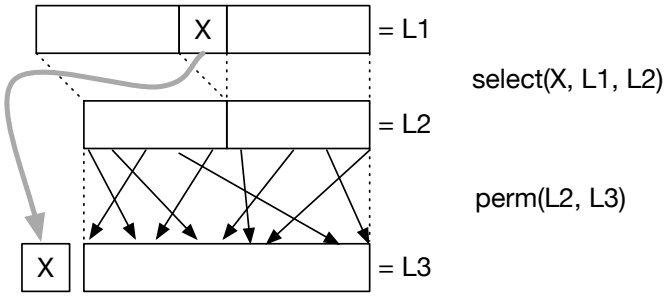
Generowanie wszystkich rozwiązań

```
% perm(L1, L2) zachodzi, gdy
%     lista L2 jest permutacją listy L1.

perm([], []).
perm(L1, [X | L3]) :-
    select(X, L1, L2),
    perm(L2, L3).
```

Poszukiwanie rozwiązań

Generowanie wszystkich rozwiązań

Generowanie listy `[X | L3]` będącej permutacją danej listy `L1`:



Wybór elementu `X` na tyle sposobów jaka jest długość listy `L1`.

Poszukiwanie rozwiązań

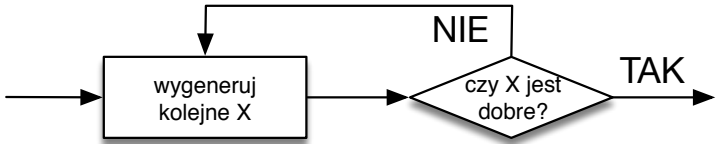
Generowanie wszystkich rozwiązań

```
?- perm([1, 2, 3], [3, 1, 2]).
true .

?- perm([1, 2, 3], [3, 2, 2]).
false.

?- perm([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Poszukiwanie rozwiązań

Generowanie i testowanie

- Niech `rozwiązanie(X)` będzie warunkiem generującym wszystkie rozwiązania.
- Niech `dobre(X)` będzie warunkiem sprawdzającym czy rozwiązanie jest dobre.
- Wówczas warunek `rozwiązanie(X), dobre(X)` generuje wszystkie dobre rozwiązania.
- Generowanie odbywa się zgodnie ze schematem następującej pętli:



Poszukiwanie rozwiązań

Generowanie i testowanie

- Zaprezentujemy program znajdujący ustawienie `N` hetmanów na szachownicy o `N` wierszach i `N` kolumnach.
- Ustawienie będziemy kodować w postaci permutacji listy liczb od `1` do `N`.
- Na *i*-tej pozycji takiej permutacji zapisany będzie numer wiersza, w którym stoi hetman z *i*-tej kolumny.
- Gdy ustawimy hetmany zgodnie z permutacją, to żadne dwa nie będą się biły w kolumnie i w wierszu.
- Pozostaje do sprawdzenia, czy żadne dwa nie biją się po ukosie.

Poszukiwanie rozwiązań

Generowanie i testowanie

```
dobra(X) :-
    \+ zła(X).

% zła(X) zachodzi, gdy wsród hetmanów ustawionych
% zgodnie z permutacją X są dwa które się biją

zła(X) :-
    append(_, [Wi | L1], X),
    append(L2, [Wj | _], L1),
    length(L2, K),
    abs(Wi - Wj) =:= K + 1.

%   abs(Wi - Wj) = odległość hetmanów w pionie
%          K + 1 = odległość hetmanów w poziomie
```

Poszukiwanie rozwiązań

Generowanie i testowanie

```
% hetmany(N, P) zachodzi, gdy permutacja P
% koduje poprawne ustawienie N hetmanów

hetmany(N, P) :-
    numlist(1, N, L), % stworzenie listy liczb 1 .. N
    perm(L, P),
    dobra(P).
```

Poszukiwanie rozwiązań

Generowanie i testowanie

```
?- hetmany(4, X).
X = [2, 4, 1, 3] ;
X = [3, 1, 4, 2] ;
false.

?- findall(X, hetmany(8, X), L), length(L, N).
L = [[1, 5, 8, 6, 3, 7, 2, 4], [...|...]|...],
N = 92.
```

Metapredykat `findall(X, p(X), L)` tworzy listę `L` wszystkich wartości `X` spełniających warunek `p(X)` (przedrostek *meta–* wskazuje, że argumentem predykatu są nie tylko dane ale również warunki).

Poszukiwanie rozwiązań

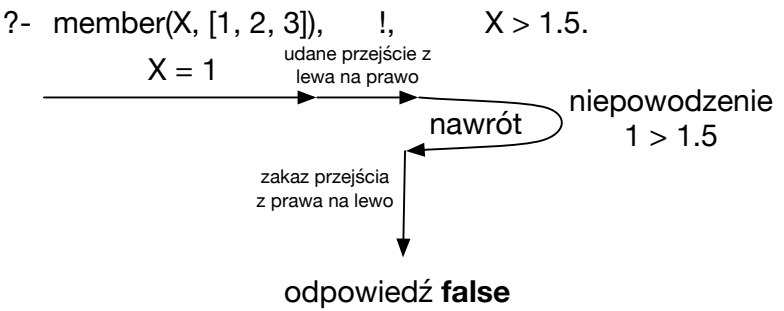
Odcięcie

- Jeśli warunek dostarcza wiele wartości, to stawiając po nim wykrzyknik (odcięcie), zostanie znaleziona tylko pierwsza z nich (odcinamy poszukiwanie kolejnych).
- Predykat `!` jest zawsze spełniony ale wpływa na poszukiwanie rozwiązać i nie dopuszcza do wykonania nawrotu celem szukania kolejnego rozwiązania.

```
?- member(X, [1, 2, 3]), X > 1.5.
X = 2 ;
X = 3.

?- member(X, [1, 2, 3]), !, X > 1.5.
false.
```

Poszukiwanie rozwiązań

Odcięcie



Odcięcie działa w celu jak „błona półprzepuszczalna”: pozwala przejść z lewa na prawo ale nie odwrotnie.

Poszukiwanie rozwiązań

Odcięcie

Zupełnie inaczej działa odcięcie jeśli użyto go w ciele reguły.

Example (Algorytm Świętego Mikołaja <wersja 1>)

Święty Mikołaj stosuje poniższe reguły do rozstrzygnięcia czy dziecko `X` dostanie nagrodę `Y` albo karę `Y`.

```
dostanie1(X, Y) :-    grzeczne(X), nagroda(Y).
dostanie1(X, Y) :- \+ grzeczne(X), kara(Y).
```

- W powyższym przykładzie, jeśli dziecko `X` nie było grzeczne, to warunek `grzeczne(X)` jest sprawdzany dwukrotnie.
- W takim przypadku zawodzi warunek `grzeczne(X)` z pierwszej reguły predykatu `dostanie1(X, Y)` i Prolog sięga do drugiej reguły.
- W drugiej regule sprawdzane jest ponownie czy dziecko było grzeczne i jeśli warunek ten zawodzi, to wybierana jest dla niego kara `Y`.

Poszukiwanie rozwiązań

Odcięcie

Example (Algorytm Świętego Mikołaja <wersja 2>)

```
dostanie2(X, Y) :- grzeczne(X), nagroda(Y).
dostanie2(X, Y) :-              kara(Y).
```

- Ta wersja nie jest poprawna.
- Jeśli będziemy próbowali wywnioskować wszystko co może dostać grzeczne dziecko, to po wyczerpaniu wszystkich nagród, predykat stwierdzi na podstawie drugiej reguły, że należy dawać mu kolejne kary.

Poszukiwanie rozwiązań

Odcięcie

Example (Algorytm Świętego Mikołaja <wersja 3>)

```
dostanie3(X, Y) :- grzeczne(X), !, nagroda(Y).
dostanie3(X, Y) :-                 kara(Y).
```

- Ta wersja jest poprawna.
- Jeśli dziecko `X` jest grzeczne, to pierwsza reguła odpowie jakie nagrody może ono dostać.
- Jeśli wyczerpią się już wszystkie nagrody, to przy próbie wycofania się do sprawdzania czy dziecko jest grzeczne, odcięcie ! spowoduje niepowodzenie i porzucenie dalszego sprawdzania warunku `dostanie3(X, Y)`, nawet jeśli są jeszcze jakieś inne reguły w jego definicji.

Poszukiwanie rozwiązań

Przykłady zastosowań odcięcia

Example (`once/1`)

Metapredykat `once(Goal)` znajduje pierwszą odpowiedź na cel `Goal` i nie będzie szukał kolejnych.

```
once(Goal) :-
    Goal, !.
```

Example (Negacja)

Metapredykat `\+ Goal` zawodzi gdy cel `Goal` jest spełniony. W przeciwnym przypadku jest spełniony.

```
\+ Goal :-
    Goal,
    !,
    fail.
\+ _.
```


Example (`forall/2`)

Metapredykat `forall(Generator, Test)` sprawdza czy wszystkie dane generowane warunkiem `Generator` spełniają warunek `Test`.

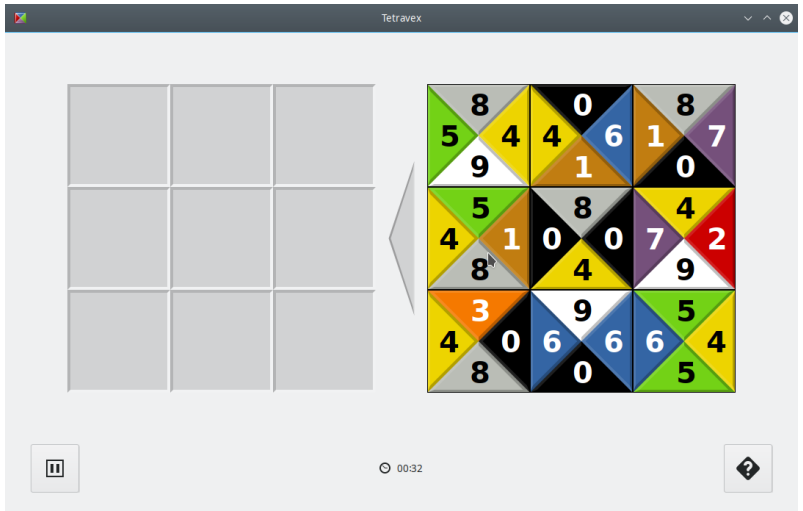
```
forall(Generator, Test) :-
    \+ (Generator, \+ Test).
```

Zwróć uwagę na konieczność oddzielenia spacją operatora `\+` od nawiasu. Bez tej spacji system uznałby, że wywoływana jest niezdefiniowana dwuargumentowa negacja.

Example (Własna wersja `var/1`)

```
myvar(X) :-
    \+ \+ X = a,
    \+ \+ X = b.
```

- Pierwszy warunek `\+ \+ X = a` jest spełniony tylko gdy `X` jest zmienną (unifikuje się z czymkolwiek) lub jest stałą `a`, przy czym jeśli `X` jest zmienną to nic nie zostanie pod nią podstawione.
- Drugi warunek `\+ \+ X = b` jest spełniony tylko gdy `X` jest zmienną (unifikuje się z czymkolwiek) lub jest stałą `b`, przy czym jeśli `X` jest zmienną to nic nie zostanie pod nią podstawione.
- Zatem oba warunki spełnione są tylko gdy `X` jest zmienną i nic nie zostanie pod nią podstawione.



Poszukiwanie rozwiązań

Gra Tetravex

- Każdy kwadratowy klocek o czterech kolorach `n`, `e`, `s` i `w`, odpowiednio na górnym, prawym, dolnym i lewym brzegu, jest reprezentowany listą czteroelementową `[n, e, s, w]`, gdzie $n, e, s, w \in \{0, 1, \ldots, 9\}$.
- Zbiór klocków reprezentowany jest listą list reprezentujących klocki.
- Plansza reprezentowana jest listą pól, przy czym każde pole reprezentowane jest listą czterech zmiennych `[N, E, S, W]`.
- Jeśli dwa pola planszy stykają się, to odpowiednie zmienne na ich listach są zunifikowane (np. jeśli pole `[N1, E1, S1, W1]` leży nad polem `[N2, E2, S2, W2]`, to `S1 = N2`).

```
% solve(+ListaKlocków, -ListaPólPlanszy)
solve(Tiles, Board) :-
    board(Board),
    insert(Board, Tiles).
```

Poszukiwanie rozwiązań

Gra Tetravex

```
board(Board) :-
    Board = [[_ ,E1,S1,_ ], [_ ,E2,S2,W2], [_ ,_ ,S3,W3],
             [N4,E4,S4,_ ], [N5,E5,S5,W5], [N6,_ ,S6,W6],
             [N7,E7,_ ,_ ], [N8,E8,_ ,W8], [N9,_ ,_ ,W9]],
    E1 = W2, E2 = W3,
    S1 = N4, S2 = N5, S3 = N6,
    E4 = W5, E5 = W6,
    S4 = N7, S5 = N8, S6 = N9,
    E7 = W8, E8 = W9.

insert([], []).
insert([B | Bs], Tiles) :-
    select(B, Tiles, Rest),
    insert(Bs, Rest).
```

Poszukiwanie rozwiązań

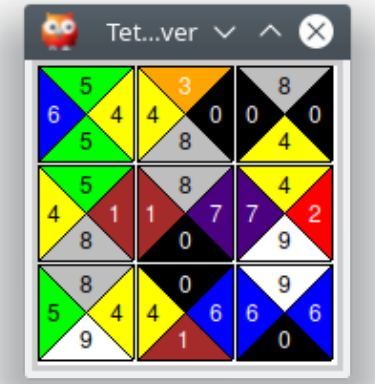
Gra Tetravex

```
?- set_prolog_flag(answer_write_options,[max_depth(0)]).
true.

?- time(solve([[8,4,9,5],[0,6,1,4],[8,7,0,1],
               [5,1,8,4],[8,0,4,0],[4,2,9,7],
               [3,0,8,4],[9,6,0,6],[5,4,5,6]], X)).
% 404 inferences, 0.000 CPU in 0.000 seconds (99% CPU, 590
X = [[5,4,5,6],[3,0,8,4],[8,0,4,0],
     [5,1,8,4],[8,7,0,1],[4,2,9,7],
     [8,4,9,5],[0,6,1,4],[9,6,0,6]] ;
% 17 inferences, 0.000 CPU in 0.000 seconds (92% CPU, 31017
false.
```

Poszukiwanie rozwiązań

Gra Tetravex

```
?- solve([[8,4,9,5],[0,6,1,4],[8,7,0,1],[5,1,8,4],[8,0,4,0],
[4,2,9,7],[3,0,8,4],[9,6,0,6],[5,4,5,6]],X), show(X, 3).
```



- Okno z rozwiązaniem stworzono dzięki modułowi XPCE, który będzie omówiony w dalszej części wykładu.
- Do łamigłówki Tetravex powrócimy jeszcze na liście zadań, gdzie do rozwiązania przykładów 6×6 potrzebne będzie użycie programowanie ograniczeń, omawiane w dalszej części wykładu.