Programação não Determinística

- Não determinismo como técnica de programação,
- Generate-and-test,
- Tipos de não determinismo,

Não determinismo

Um aspecto da programação em lógica que não se encontra noutros paradigmas de programação é o *não determinismo.* Isto é, o facto de uma interrogação ter potencialmente múltiplas respostas. Não determinismo é útil para definir e implementar certos algoritmos de uma forma eficiente. Por intuição, podemos dizer que uma máquina não determínistica é uma máquina que consegue escolher a sua próxima operação correctamente quando enfrentada com diferentes alternativas.

O Prolog aproxima-se do comportamento não determinístico do procedimento abstracto SLD da programação em lógica através da pesquisa sequencial com backtracking.

O não determinismo da programação em lógica está intimamente ligado ao facto deste formalismo usar relações, contrastante por exemplo com formalismos que usam funções. Isto é, as árvores de prova tem nós que dão origem a vários ramos o que significa que várias cláusulas unificam e resolvem com uma interrogação.

No entanto há programas que tanto podem ser determinísticos como não determinísticos.

Por exemplo append é determinístico para a query

$$?\mathsf{append}([1,2,3],[4,5],\mathsf{Z})$$

mas é não determinístico para

$$?\mathsf{append}(\mathsf{X},\mathsf{Y},[1,2,3,4,5])$$

Generate-and-test

Generate and test é uma técnica muito usada em desenho de algoritmos. Consiste na definição de dois procedimentos; um para gerar candidatos a solução de um dado problema e outro para testar estes mesmos candidatos. A ideia é procurar um candidato que seja realmente solução do problema.

Programas deste tipo são mais fáceis de construir do que programas que calculam a solução de um problema directamente. No entanto estes programas são também menos eficientes. Uma forma de optimização é tentar embeber o mais possível o processo de teste dentro do processo de geração.

Em Prolog o esquema de implementar esta técnica tem o seguinte padrão:

```
find(X) :- generate(X), test(X).
```

Como é normal no modelo computacional do Prolog, a subquery ?generate(X) sucede retornando algum valor para X. Depois esse valor é usado em ?test(X). Se esta subquery falhar a execução faz backtracking para a subquery ?generate(X) que gera novamente uma nova substituição para X. Este processo continua até que uma substituição para X satisfaça ?test(X) ou que todas as soluções tenham sido geradas por ?generate(X).

Um exemplo deste tipo de programas seria:

```
acha_primo(X,L) :- member(X,L), primo(X).
```

Generate-and-test (cont)

Gerar horários...(foi o trabalho prático de há uns anos atrás!)

```
gerar(Disc) :-
      gera_en(Disc,Ents),
      gera_dia(Dia),
      gera_hora(Hora),
      sala_disponivel(Dia,Hora,Disc,Sala),
      verifica_restricoes(Ents,Dia,Hora,Sala,Disc),
      propaga_horario(Ents,Dia,Hora,Sala,Disc).
```

Usando este técnica de geração e teste podemos implementar um mecanismo de geração de horários que satisfaçam um conjunto de restrições.

Perante uma dada disciplina Disc gera-se um dia e uma hora e uma sala para alocar essa aula (ou seja essa disciplina). Depois verifica-se se esta alocação, candidata a aula, satisfaz as restrições. Em caso afirmativo propaga-se esta aula pelos horários das várias entidades que são afectadas por ela.

Outros exemplos (Jogos)

Implementar geração de uma jogada (usando o algoritmo MiniMax) para o conhecido *quatro-em-linha* (também foi o trabalho prático em tempos idos!).

```
% gera jogadas do computador.
% perante Tabuleiro gera a Jogada melhor.
gera_jogada(Tabuleiro,Jogada):- % comeca pela coluna 1.
        tenta_comp(Tabuleiro,1,1,Jogada,-9999).

% maximizar para obter melhor jogada.
% stop quando ja tentamos as 7 possiveis jogadas!
tenta_comp(_,8,J,J,_).
tenta_comp(Tabuleiro,Coluna,_,Coluna,_):-
        computador(P), % identifica ficha do computador.
        % inserir ficha P na posicao X gerando tabuleiro Novo.
        inserir(Tabuleiro,Coluna,P,Novo),
        vitoria(Novo,P). % se vitoria entao ja' encontrou maximo!
tenta_comp(Tabuleiro,Coluna,JT,Jogada,Max):- % caso contrario:
        computador(P),
        inserir(Tabuleiro,Coluna,P,Novo),
        gerar_oponente(Novo,Valor), % gera todas as jogadas do oponente.
        % escolha a melhor para computador.
        escolher_max(Valor,Coluna,JT,Max,NovoJ,NMax),
        % gera nova coluna e processa...
        NC is Coluna + 1,
        tenta_comp(Tabuleiro,NC,NovoJ,Jogada,NMax).
tenta_comp(Tabuleiro,Coluna,JT,Jogada,Max):-  % se jogada invalida
        X1 is Coluna + 1,          % entao tenta proxima.
        tenta_comp(Tabuleiro,X1,JT,Jogada,Max).
```

Para uma possível jogada Coluna num dado tabuleiro Tabuleiro gera-se todas as possíveis respostas feita pelo oponente para esse novo tabuleiro (usando gerar_oponente). Daqui obtém-se o Valor (mínimo para o computador e máximo para o oponente) correspondente ao tabuleiro Novo. Escolhe-se a Jogada que origina o tabuleiro com maior valor. Isto é, que maximiza a posição do computador.

MiniMax (continuação)

Implementar geração das jogadas do oponente minimizando.

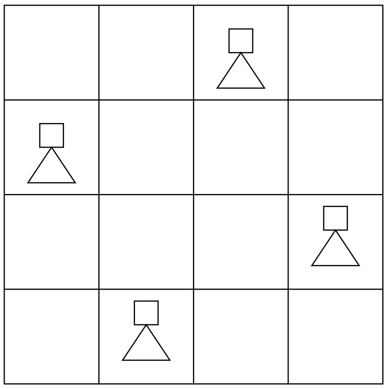
```
% gera jogadas do oponente. Obtem-se o Valor da funcao valoracao
% para as jogadas respostas do oponente perante o novo Tabuleiro gerado.
gerar_oponente(Tabuleiro,Valor):-
        tenta_op(Tabuleiro,1,9999,Valor).

% minimizar
% stop quando ja forem visitadas todas as colunas.
tenta_op(_,8,Valor,Valor).
% se vitoria entao ja encontrou minimo (-999)!
tenta_op(Tabuleiro,Coluna,_,-999):-
        oponente(P), % identifica ficha oponente.
        inserir(Tabuleiro,Coluna,P,Novo), % gera Novo tabuleiro
        vitoria(Novo,P).
tenta_op(Tabuleiro,Coluna,Tmin,Min):-
        oponente(P),
        inserir(Tabuleiro,Coluna,P,Novo),
        fval(Novo,Valor), % calcula valor para o tabuleiro gerado.
        escolher_min(Valor,Tmin,NMin), % verifica se e' minimo.
        X1 is Coluna + 1,  % a coluna que se segue...
        tenta_op(Tabuleiro,X1,NMin,Min).
tenta_op(Tabuleiro,Coluna,T,Min):- % se coluna invalida
        X1 is Coluna + 1,    % entao passa a proxima.
        tenta_op(Tabuleiro,X1,T,Min).

% Funcao de VALoracao. Usando clausulas deterministicas.
fval(Tabuleiro,999):- computador(X), vitoria(Tabuleiro,X), !.
fval(Tabuleiro,-999):- oponente(X), vitoria(Tabuleiro,X), !.
.
.
```

Aqui, tenta_op determina (usando a função de valoração fval) o valor mínimo que um tabuleiro tem depois de se tentar todas as respostas possíveis para o oponente. A função de valoração é implementada com cláusulas determinísticas. Por exemplo, 999 (valor máximo possível) se ocorrer vitória para o computador. Assim, só uma cláusula é que determina o valor para um tabuleiro. Há no entanto, variantes desta implementação!

Outro exemplo de *Generate-and-Test*

O problema das 8 rainhas (ou N rainhas)

O problema requer uma solução para a colocação de N rainhas num tabuleiro de NxN quadrados por forma a que duas peças não estejam na mesma linha (vertical, horizontal ou mesmo diagonal).



Este tabuleiro é uma solução para $\mathsf{N} = \mathsf{4}$.

O predicado queens resolve este problema gerando uma permutação da lista $[\mathsf{1}, \ldots, \mathsf{N}]$. O primeiro elemento indica a posição de linha da rainha a colocar na primeira coluna, o segundo refere-se à segunda coluna e assim sucessivamente. Por exemplo, a solução desta figura seria a lista $[\mathsf{2}, \mathsf{4}, \mathsf{1}, \mathsf{3}]$.

Assume-se assim que cada solução para este problema terá uma rainha em cada coluna e em cada linha.

N Rainhas (uma solução)

```
queens(N,Qs) :-
            range(1,N,Ns),
            permutation(Ns,Qs),
            safe(Qs).

safe([Q|Qs]) :-
            safe(Qs),
            \+ 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).

permutation(Xs,[Z|Zs]) :-
            select(Z,Xs,Ys),
            permutation(Ys,Zs).
permutation([],[]).

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

range gera uma lista de inteiros dentro de um dado intervalo.
permutation gera permutações de uma dada lista.

Tipos de Não Determinismo

- Don't Care
- Don't Know

Não-Determinismo *Don't Care*: A escolha pode ser feita arbitráriamente. Ou seja o Prolog pode escolher qualquer cláusula para reduzir a query em questão para chegar à solução. Exemplo:

```
maximo(X,Y,X) :- X =< Y.
maximo(X,Y,Y) :- Y > X.
```

Não-Determinismo *Don't Know*: A escolha é fundamental para encontrar a solução mas não é conhecida a opção correcta. Exemplo:

```
path(X,Y) :- arco(X,Y).
path(X,Y) :- path (X,Z), arco(Z,Y).
```

A melhor distinção entre os dois tipos de não-determinismo é que no primeiro a pesquisa de soluções é desnecessária enquanto que no segundo a pesquisa é essencial.