Pedro Paulo Vezzá Campos
Rafael Elias Pedretti

Implementação de Problema de Busca em Espaço de Estados Utilizando Prolog

Florianópolis - SC, Brasil

10 de dezembro de 2010

Pedro Paulo Vezzá Campos
Rafael Elias Pedretti

Implementação de Problema de Busca em Espaço de Estados Utilizando Prolog

Trabalho apresentado para avaliação na disciplina INE5416, do curso de Bacharelado em Ciências da Computação, turma 04208, da Universidade Federal de Santa Catarina, ministrada pelos professores João Cândido Lima Dovicchi e Jerusa Marchi

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis - SC, Brasil

10 de dezembro de 2010

Introdução

Para este trabalho final de INE5416 foi proposto pela professora a escolha e implementação em Prolog de um problema de busca em espaço de estados. Os alunos escolheram o problema do lobo, da ovelha e da alface, um problema clássico de travessia de rio, que remonta pelo menos ao século IX [1].

Este relatório está organizado da seguinte forma: Primeiramente será apresentada uma descrição do problema a ser modelado e resolvido. Depois será apresentada a definição dos estados inicial, final e operadores envolvidos. Em seguida será exiblida a representação dos estados em Prolog. Posteriormente será visualizada a implementação da função geradora de sucessores. Ainda, os alunos apresentarão algumas considerações sobre a execução do projeto. Por fim será apresentado o código fonte completo e documentado da aplicação finalizada.

Descrição do Problema Escolhido

Um fazendeiro viajava com três compras: Um lobo, uma ovelha e alface. Ao retornar para casa ele se deparou com uma margem de um rio, arrendando um barco para realizar a travessia. Porém, o barco arrendado era pequeno e somente comportava o fazendeiro e no máximo uma de suas compras.

Enquanto viajavam junto do fazendeiro nenhuma compra atacava a outra, no entanto, assim que fossem deixados sozinhos o lobo comeria a ovelha e a ovelha comeria a alface.

O problema consiste em terminar a travessia tanto do fazendeiro quanto de suas compras para a margem oposta sem permitir que uma compra ataque a outra. [2]

Definição dos Estados Inicial, Final e Operadores

Conforme explicado anteriormente o problema consiste em mover todos os itens de uma margem a outra, portanto foram modelados ambos os lados do rio da seguinte forma:

Estado Inicial

Margem direita: {barco, lobo, ovelha, alface}

Margem esquerda: $\varnothing$

Estado Final

Margem direita: ∅

Margem esquerda: {barco, lobo, ovelha, alface}

Operadores

`moveOvelha/2` Este predicado sempre irá realizar a mudança da ovelha e do barco de margem, uma vez que o problema não impõe restrições ao lobo estar junto da alface.

`moveLobo/2` Este predicado realiza a mudança do lobo e do barco de uma margem a outra somente se a ovelha e a alface não ficarem sozinhas em um lado do rio

`moveAlface/2` Este predicado realiza a mudança da alface e do barco de uma margem a outra somente se o lobo e a ovelha não ficarem sozinhos em um lado do rio

`moveBarco/2` Este predicado realiza a mudança somente do barco de uma margem a outra com a condição que tanto o lobo e a ovelha quanto a ovelha e a alface não ficarão sozinhos em um lado do rio

Representação dos Estados

Para a representação em Prolog foram adotadas listas de átomos como estrutura de dados principal devido a sua flexibilidade e a presença de uma biblioteca de predicados para sua manipulação produzida como atividade prática da disciplina durante o semestre. Os estados do problema estão assim representados:

Estado Inicial `[[],[barco,ovelha, lobo, alface]]`

Estado Intermediário Alguma combinação dos quatro itens distribuídos nas duas listas segundo as condições abaixo, tal como `[[lobo, alface],[barco, ovelha]]`

Estado Final `[[barco,ovelha, lobo, alface],[]]`

Por definição, há uma lista mais externa que comporta duas listas internas. A primeira lista interna contém os itens presentes na margem direita e a segunda os itens da margem esquerda. Tais listas são mutuamente exclusivas, sendo a união de ambas sempre igual a `[barco,ovelha, lobo, alface]`.

Implementação da Função Geradora de Sucessores

Em uma decisão de projeto, os alunos escolheram manter a lógica das restrições aos movimentos válidos, ou seja, garantir que nem lobo e ovelha nem ovelha e alface permaneçam sozinhos, junto com as cláusulas responsáveis por mover os itens de uma margem para outra. Dessa forma, a função geradora de sucessores tornou-se bastante simples. Sua implementação consiste em tentar invocar uma das quatro cláusulas de movimentação de itens, `moveOvelha/2`, `moveLobo/2`, `moveAlface/2` e `moveBarco/2`, delegando ao mecanismo de unificação do Prolog a busca por uma transição válida dentre as quatro possíveis.

Como algoritmo de busca em espaço de estados foi utilizada a implementação de busca em profundidade (DFS) apresentado por Palazzo em [3].

Considerações sobre o Trabalho

Este trabalho final de INE5416 proposto pela professora Jerusa foi de grande valia por apresentar como proposta a escolha de um exercício dentre um rol de problemas clássicos que permitem evidenciar ao aluno as vantagens trazidas pelo uso do paradigma lógico para a solução de diversos problemas.

Através deste trabalho foi possível experimentar diversos problemas inerentes à programação em um paradigma completamente diferente do habitual, treinando as habilidades na abordagem e solução dessas dificuldades.

Ainda, este trabalho contribuiu ao resumir bem diversos conceitos vistos durante o semestre abrangendo desde recursão, unificação e listas até culminar no projeto, desenvolvimento e testes de uma aplicação prática. Por outro lado, como os exercícios não demandaram grandes e tediosos esforços de programação os acadêmicos puderam concentrar-se nas modelagens e abordagens de solução adotadas para resolver o problema, as quais foram apresentadas anteriormente neste relatório.

Código Fonte

```
% INE5416 - PARADIGMAS DE PROGRAMACAO
% ALUNOS: PEDRO PAULO VEZZA CAMPOS - 09132033 E RAFAEL ELIAS PEDRETTI - 09132035
% TRABALHO FINAL: IMPLEMENTACAO DE BUSCA EM ESPACO DE ESTADO - PROBLEMA DA OVELHA, LOBO E ALFACE

% operacao: inicial
% especificacao: retorna o estado inicial da busca em espaco de estados
% argumento: o estado inicial
inicial(X):- X = [[],[barco,ovelha, lobo, alface]].

% operacao: final
% especificacao: retorna o estado final da busca em espaco de estados
% argumento: o estado final
final(X):- X = [[barco,ovelha, lobo, alface],[]].

% operacao: moveBarco
% especificacao: retorna o novo estado gerado apos a troca do barco (sem nenhuma compra)
% de margem dado que nem o lobo e a ovelha nem a alface e a ovelha fiquem sozinhos
% 1o. argumento: o estado atual
% 2o. argumento: o proximo estado com o barco na outra margem
moveBarco([D,E],[Dn,En]):- \+mesmaMargem(lobo,ovelha,D), \+mesmaMargem(alface,ovelha,D),
                           membro(barco,D),troca(barco,D,E,Dn,En).

moveBarco([D,E],[Dn,En]):- \+mesmaMargem(lobo,ovelha,E), \+mesmaMargem(alface,ovelha,E),
                           membro(barco,E),troca(barco,E,D,En,Dn).

% operacao: moveOvelha
% especificacao: retorna o novo estado gerado apos a troca do barco e da ovelha de margem
% 1o. argumento: o estado atual
% 2o. argumento: o proximo estado com o barco e a ovelha na outra margem
moveOvelha([D,E],[Dn,En]):- membro(ovelha,D), membro(barco,D), troca(ovelha,D,E,Dni,Eni),
                            moveBarco([Dni,Eni],[Dn,En]).

moveOvelha([D,E],[Dn,En]):- membro(ovelha,E), membro(barco,E), troca(ovelha,E,D,Eni,Dni),
                            moveBarco([Dni,Eni],[Dn,En]).

% operacao: moveAlface
% especificacao: retorna o novo estado gerado apos a troca do barco e da alface de margem dado
% que o lobo e a ovelha nao fiquem sozinhos
% 1o. argumento: o estado atual
% 2o. argumento: o proximo estado com o barco e a alface na outra margem
moveAlface([D,E],[Dn,En]):- membro(alface,D), membro(barco,D),
                            \+mesmaMargem(lobo,ovelha,D), troca(alface,D,E,Dni,Eni),
                            moveBarco([Dni,Eni],[Dn,En]).

moveAlface([D,E],[Dn,En]):- membro(alface,E), membro(barco,E),
```

```
                        \+mesmaMargem(lobo,ovelha,E), troca(alface,E,D,Eni,Dni),
                        moveBarco([Dni,Eni],[Dn,En]).

% operacao: moveLobo
% especificacao: retorna o novo estado gerado apos a troca do barco e do lobo de margem dado
% que a ovelha e a alface nao fiquem sozinhos
% 1o. argumento: o estado atual
% 2o. argumento: o proximo estado com o barco e o lobo na outra margem
moveLobo([D,E],[Dn,En]):- membro(lobo,D), membro(barco,D),
                        \+mesmaMargem(alface,ovelha,D), troca(lobo,D,E,Dni,Eni),
                        moveBarco([Dni,Eni],[Dn,En]).

moveLobo([D,E],[Dn,En]):- membro(lobo,E), membro(barco,E),
                        \+mesmaMargem(alface,ovelha,E), troca(lobo,E,D,Eni,Dni),
                        moveBarco([Dni,Eni],[Dn,En]).

% operacao: resolve
% especificacao: metodo principal da resolucao do problema. Inicia a busca em profundidade a
% partir do estado inicial e exibe a solucao encontrada em ordem
resolve:- inicial(X),profundidade([], X, Solucao), mostraSolOrdem(Solucao).

% operacao: profundidade
% especificacao: executa uma busca em profundidade ate que a clausula objetivo/1 retorne verdade ou
% esgote os espaco de solucoes. Retorna os estados percorridos da origem fornecida no 2o argumento
% ate a solucao
% 1o. argumento: os estados ja visitados anteriormente
% 2o. argumento: o estado atual da busca em profundidade
% 3o. argumento: a solucao encontrada pela busca em profundidade
profundidade(Caminho, Atual, [Atual | Caminho]) :- objetivo(Atual).
profundidade(Caminho, Atual, Solucao) :- sucessor(Atual, Sucessor), \+membro(Sucessor, Caminho),
                                        profundidade([Atual | Caminho], Sucessor, Solucao).

% operacao: objetivo
% especificacao: retorna verdadeiro se a busca em profundidade atingiu o estado final e suspende o
% backtracking
% argumento: retorna o estado final da busca em espaco de estados
objetivo(N):- final(N),!.

% operacao: sucessor
% especificacao: clausula geradora de proximo estado. Tenta mover um dos itens: ovelha + barco,
% alface + barco, lobo + barco, somente barco
% 1o. argumento: estado atual
% 2o. argumento: proximo estado gerado
sucessor(Atual,Sucessor):- moveOvelha(Atual,Sucessor);moveAlface(Atual,Sucessor);
                        moveLobo(Atual,Sucessor); moveBarco(Atual,Sucessor).

% operacao: mesmaMargem
% especificacao: retorna verdadeiro se os dois itens fornecidos estao em uma mesma margem
% 1o. argumento: primeiro item a ser verificado
% 2o. argumento: segundo item a ser verificado
% 3o. argumento: o estado representando as margens do rio
mesmaMargem(Coisa1,Coisa2,Margem):- membro(Coisa1,Margem), membro(Coisa2,Margem).

% operacao: mostraSolucao
```

```
% especificacao: exibe cada um dos estados da solucao final um por linha
% argumento: a solucao a ser exibida
mostraSolucao([H|[]]):- write(H), nl.
mostraSolucao([H|T]):- write(H), nl, mostraSolucao(T).

% operacao: mostraSolOrdem
% especificacao: inverte a lista de estados da solucao e a exibe na tela
% argumento: a lista de estados a ser exibida
mostraSolOrdem(L):- inverte(L,Ln), mostraSolucao(Ln).


% operacao: troca
% especificacao: remove um item dado de uma lista e insere no inicio de outra
% 1o. argumento: item a ser trocado de lista
% 2o. argumento: lista de origem
% 3o. argumento: lista de destino
% 4o. argumento: nova lista de origem sem o elemento trocado
% 5o. argumento: nova lista de destino com o elemento trocado
% exemplo: troca(a, [a,b,c], [d,e,f], [b,c], [a,d,e,f]).

troca(_, [], [], D, D).
troca(E, [E|T], D, T, Dn) :- insereNoInicio(E, D, Dn).
troca(E, [H|T], D, On, Dn) :- troca(E, T, D, On2, Dn), insereNoInicio(H, On2, On).

% operacao: inverte
% especificacao: inverte os elementos de uma lista
% 1o. argumento: lista
% 2o. argumento: lista invertida
inverte([],[]).
inverte([H|T],I) :- inverte(T, R), concat(R, [H], I).

% operacao: membro
% especificacao: verifica se um termo eh membro de uma lista
% 1o. argumento: termo
% 2o. argumento: lista
membro(X,[X|_]).
membro(X,[_|Y]):- membro(X,Y).

% operacao: concat
% especificacao: concatena 2 listas
% 1o. argumento: 1o. lista
% 2o. argumento: 2o. lista
% 3o. argumento: lista concatenada
concat(X, [], X).
concat([], X, X).
concat([H|T], Y, C) :- concat(T, Y, C1), cons(H, C1, C).

% operacao: insereNoInicio
% especificacao: insere um termo no inicio de uma lista
% 1o. argumento: termo
% 2o. argumento: lista
% 3o. argumento: lista com o termo inserido
insereNoInicio(X, L, I) :- concat([X], L, I).
```

```
% operacao: cons
% especificacao: retorna uma lista
% 1o. argumento: cabeca da lista
% 2o. argumento: cauda da lista
% 3o. argumento: a lista construida
cons(H, T, L) :- L = [H|T].
```

Referências Bibliográficas

[1] PRESSMAN, I.; SINGMASTER, D. The Jealous Husbands and The Missionaries and Cannibals. *The Mathematical Gazette*, The Mathematical Association, v. 73, n. 464, p. 73–81, 1989. ISSN 00255572. Disponível em: <http://dx.doi.org/10.2307/3619658>.

[2] WIKIPÉDIA. *Problema do fazendeiro, o lobo, o carneiro e a alface — Wikipédia, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2010. Acesso em 3 de dezembro de 2010. Disponível em: <http://pt.wikipedia.org/w/index.php?title=Problema_do_fazendeiro,_o_lobo,_o_carneiro_e_a_alface&oldid=22161357>.

[3] PALAZZO, L. A. M. *Introdução à Programação em Prolog*. Pelotas: Editora da Universidade Católica de Pelotas, 1997.