Relatório Técnico

As múltiplas possibilidades da Programação em Lógica

Prof. Evandro de Oliveira Araújo
Departamento de Eletrônica
UFMG

Índice

1. Introdução

A grande motivação para realização deste trabalho foi a necessidade de divulgar uma linguagem de programação de alto nível capaz de realizar processamento simbólico bem como cálculos matemáticos essenciais na vida profissional de qualquer engenheiro.
Quando comecei a projetar um sistema de reconhecimento de padrões, julguei natural a utilização de uma linguagem voltada para o processamento numérico e bastante amigável como Matlab. Durante o desenvolvimento, percebi alguns inconvenientes do Matlab no tratamento de “strings”. Estes inconvenientes e a natureza do problema que estava abordando (reconhecimento de textos e diagnóstico de falhas) levaram-me naturalmente a buscar na linguagem Prolog a solução para os problemas de programação que estava encontrando. Aos poucos, a linguagem Prolog foi assumindo um papel relevante no sistema que estava sendo desenvolvido. Ao final do projeto, foi possível realizar todas as tarefas de programação em Prolog. A única perda, em relação ao Matlab, foi a falta de visualização gráfica dos conjuntos nebulosos, utilizados no sistema de reconhecimento de padrões. A realização deste sistema totalmente em Prolog foi possível graças ao desenvolvimento de uma série de *predicados Prolog* para representação e operações com matrizes, descritas por listas. O desenvolvimento destes *predicados Prolog* tornou fácil a utilização da linguagem Prolog numa grande gama de aplicações em Engenharia. A dificuldade ou inadequação da linguagem Prolog em traçar gráficos é talvez uma das suas maiores fraquezas. Esta deficiência na visualização gráfica dos resultados (que pode ser atenuada, integrando o Prolog com uma outra linguagem) é compensada pela facilidade de desenvolvimento, pela organização e descoberta de conhecimento que uma linguagem de Programação em Lógica propicia. Por entender que esta extensão da linguagem Prolog representa uma alternativa interessante para resolução de problemas para os alunos dos diversos cursos de engenharia, ela é apresentada neste presente relatório técnico.

2. Objetivo

O objetivo deste trabalho é mostrar a grande versatilidade da Programação em Lógica, representada pela linguagem Prolog, na resolução de problemas das mais diversas áreas do conhecimento. Será dada uma ênfase ao desenvolvimento de pequenos programas que serão úteis para determinados ramos da engenharia. O desenvolvimento destes programas, predicados Prolog, permitirão aos usuários permanecer numa linguagem de alto nível enquanto realizam suas diversas tarefas tais como tratamento de Linguagem Natural, Reconhecimento de Padrões, Sistemas Especialistas, e muitas outras.

3. Relevância

Este trabalho complementa o Projeto de Ensino desenvolvido há alguns anos. Ele representa uma real possibilidade de divulgação de uma excelente ferramenta de

programação, voltada para o tratamento do conhecimento, o planejamento de ações, a modelagem e a resolução de problemas propiciando uma grande interação com o corpo discente dos cursos de Engenharia Elétrica, Engenharia de Controle e Automação e Engenharia de Produção.

4. Desenvolvimento

O trabalho exposto a seguir enfatizará a utilização da Linguagem Prolog em aplicações da Álgebra Linear (como em Controle de Processos, Reconhecimento de Padrões) como também em Planejamento de Ações e Modelagem de Problemas. Na exposição dos predicados desenvolvidos, foi adotada a seguinte filosofia de programação: sempre que possível, os predicados referidos na definição de algum outro predicado são apresentados logo a seguir, visando uma leitura fácil e um entendimento rápido do programa. As exceções serão alguns predicados básicos envolvendo operações com listas. Por serem de uso generalizado, eles estarão reunidos na seção 7 (anexo) e na chamada a eles, haverá o símbolo * indicando que eles estão definidos no final do texto, em ordem alfabética. O símbolo % representa comentário em Prolog.

4.1 Aplicações da Álgebra Linear

Para efetuar eficientemente as operações da Álgebra Linear é necessário encontrar uma representação adequada para as matrizes. Uma lista em Prolog é uma coleção de elementos separados por vírgulas e envolvida por colchetes. Cada elemento pode ser um *atom* (número ou uma cadeia de caracteres começando por uma letra minúscula) ou uma lista de elementos. Neste último caso, teremos uma lista de sublistas. As listas constituem uma estrutura de dados muito poderosa na linguagem Prolog, devido a sua grande flexibilidade e poder de representação.

4.1.1 Representação das matrizes

As listas constituem assim uma escolha natural para descrição de *arrays* multidimensionais como as matrizes. Elas são uma representação muito simples e eficiente para as matrizes devido à grande flexibilidade que elas apresentam. Podemos representar matrizes de qualquer tamanho sem necessidade prévia de dimensionamento, alocação de memória, etc. Uma matriz será representada por uma lista de sublistas. Cada sublista é uma linha da matriz. Por exemplo, a matriz identidade 3x3, denominada aqui de i3, será representada por uma lista contendo três sublistas: [[1,0,0],[0,1,0],[0,0,1]]. Nesta seção, todas as matrizes serão armazenadas seqüencialmente através da primitiva *recordz(Chave,Info,Endereço)*. A chave escolhida é *mtr*, e a informação será *matr(Nome_da_matriz,Lista_lista)*. Assim, para guardar seqüencialmente a matriz i3 acima, teremos:

```
recordz(mtr,matr(i3, [[1,0,0],[0,1,0],[0,0,1]],_).
```

O primeiro argumento da primitiva recordz é o *atom* mtr escolhido como chave. O segundo argumento, a informação a ser armazenada, é a estrutura constituída pelo *atom* matr, escolhido como *functor*, e como argumentos temos o nome

escolhido para a matriz, i3, e a lista de listas, descrevendo as linhas da matriz identidade 3x3.

4.1.2 Lendo um arquivo e armazenando matrizes

O primeiro desenvolvimento a ser apresentado trata do armazenamento de uma série de matrizes escritas (através de um editor de programas qualquer), à parte, no arquivo *mtrz.txt*. O programa apresentado, a seguir, lê o arquivo *mtrz.txt*, e grava as matrizes no formato acima, sob a chave *mtr*, conforme o código abaixo. Este arquivo contém matrizes de ordens variadas. Cada linha deste arquivo descreve uma matriz e tem o seguinte formato *mt(Nome_da_matriz,Lista_listas)*. Por exemplo, a primeira linha descreve a matriz rotulada de *a*:

```
mt(a,[[1,3,2,4],[-2,3,2,5],[2,0,3,5],[1,2,4,6]]).
```

O predicado Prolog *grava_mtr*, definido abaixo, lê este arquivo, grava no formato acima as matrizes *a*, *b*, …, *f*, descritas no arquivo, através da primitiva *recordz*. Ele grava também as matrizes identidade de ordem um até a ordem desejada, fixada no programa. No presente caso, são gravadas as matrizes *i1*, …, *i5*.

```
%le matrizes do arquivo mtrz.txt e grava sob a chave mtr
grava_mtr :- apg(mtr)*,  %apaga registro
             open(H,$mtrz.txt$,r), ctr_set(0,1),
             le_grv_mtr(H),close(H),grv_ident.

le_grv_mtr(H) :- repeat,
            [! read(H,Mt) !],
            [! fim_arq(Mt,Ind) !],
            Ind == fim.

fim_arq(end_of_file,fim) :- !.
fim_arq(Mt,nao) :- ctr_inc(0,I),
                   grv_mtr(Mt),
                   write(I:Mt), nl.

grv_mtr(mt(Nome,L)) :- recordz(mtr,matr(Nome,L),_).

%grava matriz identidade: i1(1x1), i2 (2x2), ..., i5(5x5)
grv_ident :- grv_ident(5,0). %até ordem 5

grv_ident(N,N).
grv_ident(N,I) :- II is I+1, NI is I+2,
                  ident(NI,L,0),int_text(II,SI),
                  concat(i,SI,Ni), atom_string(Ai,Ni),
                  recordz(mtr,matr(Ai,L),_), grv_ident(N,II).
```

```
ident(N,[],M) :- N=<M+1.
ident(N,[H|T],I) :- II is I+1, get_line(N,II,H,1),
ident(N,T,II).

get_line(N,_,[],N).
get_line(N,I,[1|T],I) :- J is I+1, get_line(N,I,T,J).
get_line(N,I,[0|T],J) :- JJ is J+1,get_line(N,I,T,JJ).
```

4.1.3 Introduzindo uma matriz via teclado

Uma outra forma de introduzir uma matriz é fornecer, via teclado, seu nome e suas linhas. O predicado para executar esta operação é *entra_mtr*,

```
%le uma matriz linha por linha via teclado e grava
entra_mtr :- write($nome: $), read(Nome), nl,
      recordz(mtr,matr(Nome,[]),_), write($numero_linhas: $),
      read(N), le_linhas(Nome,N,0).

le_linhas(_,N,N) :- !, write($**** FIM ****$).
le_linhas(Nome,N,I) :- II is I+1, write($linha $), write(II),
                       write($: $), read(L), nl,
                       recorded(mtr,matr(Nome,L0),R),
                       inclui_fim(L,L0,NL)*,
                       replace(R,matr(Nome,NL)),
                       le_linhas(Nome,N,II).
```

4.1.4 Mostrando as matrizes armazenadas

Para mostrar as matrizes armazenadas, foram definidos os predicados *mostra_mtr(Nome)*, *mostra_mtrf(Nome)*, *mostra_mtr* e *mostra_mtrf.* O primeiro predicado mostra a matriz como uma lista de listas, o segundo, a matriz linha por linha e o terceiro, mostra todas as matrizes armazenadas, como lista de listas. Por fim, *mostra_mtrf* exibe todas as matrizes armazenadas linha por linha. Consultas com os predicados *mostra_mtr(f)* e *mostra_mtrf(f)* exibem a matriz *f* respectivamente como

```
                [[3,4,0],[1,2,3],[1,1,1]],
                         [3,4,0]
                         [1,2,3]
                         [1,1,1]
```

```
%mostra a matriz A (mostra_mtr(A)),
%mostra a matriz A linha por linha (mostra_mtrf(A),
%mostra todas as matrizes gravadas (mostra_mtr)
%mostra todas as matr. gravadas, linha p/ linha (mostra_mtrf)

mostra_mtr(A) :- recorded(mtr,matr(A,L),_),
                 write($matriz $),write(A),
                 write($:$),nl,write(L).
```

```
mostra_mtrf(Nome) :- recorded(mtr,matr(Nome,L),_),
                     write($matriz $), write(Nome),
                     write($:$), nl, escr_linhas(L).

mostra_mtr :- recorded(mtr,L,_),[! write(L), nl !], fail.
mostra_mtr.

mostra_mtrf :- recorded(mtr,matr(Nome,L),_),
               [! write($matriz $), write(Nome), write($:$),
               nl, escr_linhas(L) !], fail.

mostra_mtrf. % escreve a matriz linha por linha

escr_linhas([]).
escr_linhas([H|T]) :- write(H), nl, escr_linhas(T).
```

4.1.5 Dimensão de uma matriz e Operações Elementares

A seguir, são definidos o predicado *dim(Nome,NL,NC)* para determinar as dimensões de uma matriz e o predicado *fixa_valor(A,I,J,K,NL)* para fixar o valor do elemento (i,j) da matriz A em k, onde *NL* é a lista de listas da matriz modificada. As operações elementares sobre matrizes são efetuadas a partir de modificações em alguns elementos da matriz identidade. Por exemplo, para trocar a 1ª coluna de uma matriz *A*(3x3) pela 3ª coluna, basta partir da matriz identidade *i3*, fixar *i3*(1,1)=0, *i3*(1,3)=1, *i3*(3,1)=1, *i3*(3,3)=0, obtendo *i3m*, *i3* modificada, e efetuar o produto de matrizes *A*i3m*.

```
%determina a dimensao da matriz A (NL,NC)
dim(A,NL,NC) :- recorded(mtr,matr(A,L),_), dimm(L,NL,NC).

dimm(L,NL,NC) :- compr(L,NL)*, nc(L,NC).
nc([H|_],NC) :- compr(H,NC).

%fixa o valor Aij em K e retorna a nova matriz A (L==>NL)
fixa_valor(A,I,J,K,NL) :- recorded(mtr,matr(A,L),R),
                          enesimo(L,I,Li)*,
                          fix_aij_em_k(L,NL,I,J,K,1),
                          replace(R,matr(A,NL)).

%formacao das matrizes que executam as operacoes elementares
%mtr elementar:ident(NxN) N=3,i3 modif.=> i3m: NL(I,J)=K

mtr_el(N,I,J,K,NL) :- int_text(N,SN), concat(i,SN,Si),
                      atom_string(S,Si),
                      recorded(mtr,matr(S,L),_),
                      enesimo(L,I,Li)*, concat(Si,m,Sm),
                      fix_aij_em_k(L,NL,I,J,K,1),
```

```
                        recordz(mtr,matr(Sm,NL),_).
fix_aij_em_k([],[],_,_,_,_).
fix_aij_em_k([H|T],[NH|NT],I,J,K,I) :- fij(H,NH,J,K,1),
                 II is I+1, fix_aij_em_k(T,NT,I,J,K,II).
fix_aij_em_k([H|T],[H|NT],I0,J,K,I) :- II is I+1,
                             fix_aij_em_k(T,NT,I0,J,K,II).
fij([],[],_,_,_).
fij([H|T],[K|NT],J,K,J) :- JJ is J+1, fij(T,NT,J,K,JJ).
fij([H|T],[H|NT],J0,K,J) :- JJ is J+1, fij(T,NT,J0,K,JJ).

%determina o elemento aij da matriz A
aij(A,I,J,Aij) :- recorded(mtr,matr(A,L),_), enesimo(L,I,Li),
                  enesimo(Li,J,Aij)*.

%submatriz L contem as linhas Lis e colunas Cis da matriz A
subm(A,Lis,Cis,L) :- recorded(mtr,matr(A,LL),_),
                     get_lc(LL,Lis,Cis,L,[]).

get_lc(_,[],_,L,L) :- !.
get_lc(LL,[H|T],Cis,L,Li) :- enesimo(LL,H,Lm)*,
                             get_l(Lm,Cis,Lmf,[]),
                             inclui_fim(Lmf,Li,Lfi)*,
get_lc(LL,T,Cis,L,Lfi).

%get_l(L,Cis,L_el,L_el): os elementos de L que estão em Cis
get_l(Lm,[],L,L) :- !.
get_l(Lm,[H|T],Lmf,L0) :- enesimo(Lm,H,E)*,
                          inclui_fim(E,L0,Lp)*,
                          get_l(Lm,T,Lmf,Lp).
```

4.1.6 Produto de Matrizes, Matriz Inversa, Cálculo de Determinante

Outras operações importantes com matrizes são definidas abaixo: produto de matrizes, determinante de uma matriz, matriz adjunta e matriz inversa. O determinante é obtido através dos elementos da 1ª linha, cabeça da lista de listas.

```
%produto de matrizes pm(A,B,AB) - A, B, AB: lista de listas
pm([],_,[]).
pm([H|T],B,[L1|T1]) :- linha_prod(H,B,L1), pm(T,B,T1).

linha_prod(L,M,LM) :- trnsp(M,Mt), lprod(L,Mt,LM).

trnsp(A,AT) :- klis(A,[],AT). %AT: transposta de A

klis([],L,L).
klis([H|T],Li,L) :- klist(H,Li,M), klis(T,M,L).
```

```
klist([],[],[]).
klist(H,[],L) :- flat(H,L)*.

klist([H|T],[H1|T1],[H2|T2]) :- inclui_fim(H,H1,H2)*,
                                 klist(T,T1,T2).
lprod(_,[],[]).
lprod(L,[H|T],[P|S]) :- detpesc(L,H,P)*, lprod(L,T,S).

%Cálculo de determinante det(L,D), L é a lista de listas
det(L,D) :- compr(L,2)*, !, det2(L,D).
det(L,D) :- compr(L,N)*, proc(L,D,0,0,N).

det2([[A,B],[C,D]],R) :- R is A*D-B*C.

proc(_,D,D,N,N) :- !.
proc([H|Tc],D,S,I,N) :- II is I+1, calcdetsubm(Tc,Dm,II),
            impar(II,M), enesimo(H,II,X)*, SS is S + X*Dm*M,
            proc([H|Tc],D,SS,II,N).

impar(X,-1) :- Y is X mod 2, Y == 0, !.
impar(X,1).

%Cálculo dos determinantes das submatrizes
calcdetsubm(Tc,Dm,I) :- detsubm(Tc,Tr,I), det(Tr,Dm).

detsubm([],[],_).
detsubm([Li2|T],[L|M],I) :- exclpos(Li2,I,L)*,detsubm(T,M,I).
                            %exclui o Iésimo elemento de Li2

Ex.:A=[[1,3,2,4],[-2,3,2,5],[2,0,3,5],[1,2,4,6]]).
    B=[[3,2,1,4],[1,2,4,5],[7,1,2,4],[2,3,4,5]]).

AB=[[28,22,33,47],[21,19,34,40],[37,22,28,45],[45,28,41,60]]
   det(A)=-43  det(B)=-64  det(AB)=2752

%matriz adjunta de A: matriz dos cofatores transposta
%inversa de A: inv(A)=adj(A)/det(A)
%B é a inversa de A, poderia ter sido definido inv(A,B)

inv(A) :- recorded(mtr,matr(A,MA),_), adj(MA,C,MA,0),
          det(MA,DA), %C é a matriz dos cofatores
          divm(C,DA,Bi), trnsp(Bi,B), % Bi=C/DA
          write(A), write($:$), write(MA),nl,
          write($inv($), write(A), write($):$), write(B).
```

```
adj([],[],_,_).
adj([H|T],[L|R],A,I) :- II is I+1, adjl(H,L,A,II,0),
adj(T,R,A,II).

adjl([],[],_,_,_).
adjl([H|T],[L|R],A,I,J) :- JJ is J+1, exclui_n(I,A,NL)*,
          exclui_c(NL,JJ,NLC), det(NLC,L0),
          L is L0*(-1)^(I+JJ), adjl(T,R,A,I,JJ).

exclui_c([],_,[]).%exclui_c(L,JJ,NL) exclui col JJ de L => NL
exclui_c([H|T],JJ,[S|R]) :- exclui_n(JJ,H,S)*,
exclui_c(T,JJ,R). %exclui o elemento da posição JJ de H => S

divm([],_,[]). %divm(L,N,NL) divide L por N => NL
divm([H|T],DA,[L|R]) :- divl(H,DA,L), divm(T,DA,R).

divl([],_,[]).
divl([H|T],DA,[A|S]) :- A is H/DA, divl(T,DA,S).
```

4.1.7 Cálculo dos menores de uma matriz

Para o cálculo dos menores, o predicado lin(I,A,LI) determina a linha I da matriz A. Idem, em relação a col(J,A,CJ). O predicado dlm(L,P,LL) determina todas as listas de P elementos com valores pertencentes a L, dispostos em ordem crescente. Finalmente, o predicado dmnr(M,N,P,LL,LC) determina todas as combinações de P linhas (colunas), LL (LC), de uma matriz (MxN).

```
lin(I,A,LI) :- recorded(mtr,matr(A,L),_), enesimo(L,I,LI)*.
col(J,A,CJ) :- recorded(mtr,matr(A,L),_), trnsp(L,LT),
               enesimo(LT,J,CJ)*.

%LL:lista de listas de P elementos de 1 a M
%LC:lista de listas de P elementos de 1 a N
%lnb(3,Lm): Lm=[1,2,3], dlm(Lm,2,LL): LL=[[1,2],[1,3],[2,3]]

dmnr(M,N,P,LL,LC) :- apg*(mnl), apg(mnc),
     lnb(M,Lm), lnb(N,Ln), dlm(Lm,P,LL), dlm(Ln,P,LC),
     recordz(mnl,LL,_), recordz(mnc,LC,_).

lnb(M,Lm) :- lnb(M,Lm,0,[]). %Lm:lista de 1 a M

lnb(M,L,M,L) :- !.
lnb(M,L,I,Li) :- II is I+1, inclui_fim(II,Li,LL)*,
lnb(M,L,II,LL).

dlm(L,P,[L]) :- compr(L,P)*, !. %L=[1,2,3, ..., M]
dlm(L,1,LP) :- flat(L,LP)*.
dlm([H|T],P,LP) :- PP is P-1, dlm(T,PP,Lp1), incl(H,Lp1,L1),
```

```
                    dlm(T,P,L2), append(L1,L2,LP)*.
incl(_,[],[]).
incl(A,[H|T],[[A|H]|S]) :- incl(A,T,S).
```

4.1.8 Simulando o comando *For*

A idéia do comando *for* pode ser simulada através dos predicados abaixo. Em seguida, é apresentada uma forma de tratar números complexos.

```
%simular o for para achar e exibir o quadrado de 1 a N
calcula_mostra_quadrado_1aN(N) :- fori(1,N).

fori(I,N) :- I>N,!.
fori(I,N) :- Iq is I*I, NI is I+1, write(I:Iq), nl,
             fori(NI,N).

%Generalizando o for de N1 a N2, variação numa única dimensão
%Generalizando o for para múltiplas dimensões
%pred(X,Y) é um predicado qualquer a ser definido

sqr(N1,N2) :- fori(N1,N2,N1).
fori(I1,I2,I) :- I>I2, !.
fori(I1,I2,I) :- Iq is I*I, NI is I+1, write(I:Iq), nl,
                 fori(I1,I2,NI).

forij(X1,X2,Y1,Y2) :- forij(X1,X2,Y1,Y2,X1).
forij(X1,X2,Y1,Y2,X) :- X > X2, !.
forij(X1,X2,Y1,Y2,X) :- [! forj(X1,X2,Y1,Y2,X,Y1) !],
                         NX is X+1, forij(X1,X2,Y1,Y2,NX).
forj(X1,X2,Y1,Y2,X,Y) :- [! pred(X,Y) !], Y < Y2, NY is Y+1,
                          forj(X1,X2,Y1,Y2,X,NY).
forj(_,_,_,_,_,_). %pred(X,Y) a definir
```

4.2 Operações com números complexos

A Base para efetuar operações com números complexos será a representação destes como listas de dois elementos.

```
%Numero complexo Z=[Parte Real,Parte imaginaria]
modulo2([A,B],M) :- M is A*A+B*B.
modulo([A,B],M) :- S is A*A+B*B, M is sqrt(S).

conj([A,B],[A,-B]). %conjugado

prod_compl([A,B],[C,D],[Re,Im]) :- Re is A*C-B*D,
                                   Im is A*D+B*C.
```

```
div_compl(Z1,Z2,Z) :- conj(Z2,Z2c), modulo2(Z2,M2),
  prod_compl(Z1,Z2c,Zp), prod_compl(Zp,[1/M2,0],Z).

%mudança de representação da forma cartesiana z=a+bj para a
%forma polar z=|z|e^j θ

cart_polar([A,B],[M,Teta]) :- modulo([A,B],M),
                              c_atan(A,B,Teta).

c_atan(0,B,Teta) :- B>0, !, Teta is round(pi/2,2).
c_atan(0,B,Teta) :- B<0, !, Teta is -round(pi/2,2).
c_atan(0,0,0) :- !.
c_atan(A,B,Teta) :- Teta is atan(B/A).

cart_polarg(Z,[M,Tetag]) :- cart_polar(Z,[M,Teta]),
                    Tetag is round(Teta*180/pi,2).
```

4.3 Reconhecimento de Padrões

O grande desenvolvimento tecnológico das últimas décadas é responsável pelo fluxo cada vez maior de informação exigindo o uso de sistemas de recuperação de informação cada vez mais sofisticados. Com o advento da internet, aumentou bastante a procura por sistemas capazes de pesquisar grandes Bases de Dados de Documentos a partir de descrições de seus conteúdos. Quando se pretende buscar um documento na rede sobre determinado tema, um problema de Classificação de Padrões é definido. A busca de uma representação adequada para estes problemas de classificação / recuperação de textos é uma questão chave nestas aplicações de Reconhecimento de Padrões. A possibilidade de representar um texto como uma forma geométrica plana a ser descrita através de umm modelo nebuloso permite que uma ampla variedade de técnicas de Reconhecimento de Padrões possam ser empregadas. A identificação de um texto com uma classe de textos versando sobre um tema corresponde à procura de um protótipo cujo modelo é o mais similar ao texto. Um texto pode ser representado para fins de reconhecimento de padrões pelo conjunto de palavras relevantes que o compõe, levando-se em conta o número de ocorrências destas palavras no texto. Um procedimento geral para construção de um sistema de classificação de textos pode ser o seguinte: reúne-se um grande conjunto de assuntos de interesse e para cada assunto, um número suficiente de textos versando sobre o tema. Para cada tema, determina-se um texto protótipo, representando o assunto, um conjunto de palavras juntamente com a incidência de cada uma neste conjunto protótipo. A seguir, forma-se uma Base de Dados Geral de palavras pela reunião de todas as palavras ocorrendo em todos conjuntos de palavras que deram origem aos protótipos. A partir daí, a tarefa de classificação de um texto em um tema pode ser feita, calculando-se qual protótipo é mais próximo do texto apresentado. Esta comparação é feita com os modelos nebulosos dos protótipos e do texto em teste. O modelo nebuloso do texto de amostra (e do protótipo) pode ser obtido, representando o texto como uma forma

geométrica plana, um conjunto de pontos (x,y), onde $x=K_x.i$, representa a seqüência de palavras, i=1,2,… e $y=K_y.n_i$. Os parâmetros K_x, K_y, n_i representam fatores de escala horizontal e vertical e a incidência da palavra *i* da Base de Dados Geral de palavras no texto, respectivamente. Desenhada a figura que representa o texto, é escolhida uma partição lingüística e determinada a representação dos conjuntos nebulosos. Todo este processamento foi feito na Linguagem Prolog. A seguir, é mostrado, em linhas gerais, o programa que determina a classe que melhor identifica um texto qualquer, obtido na internet. O procedimento começa com a caracterização dos temas escolhidos (conjuntos protótipos). Para tal , diversos textos sobre cada assunto foram obtidos na rede no formato *txt*. Outra série de arquivos texto foram extraídos para fins de teste. Abaixo, é fornecido o programa em linhas gerais.

```
clsf_txt(Arq,Lclf) :- mtime, [-pln],
  int_text(N,SN), concat([lcl,Arq,$.txt$],Aclsf),
  create(H,Aclsf), clsf(Arq,Lclf,H), nl(H), escr_rslt(H),
  close(H), tproc.

mtime :- eraseall(hms), time(time(H,M,S,_)),
         recordz(hms,tempo(H,M,S),_).

tproc :- time(time(Hf,Mf,Sf,_)),
         recorded(hms,tempo(H,M,S),_),
         Tsf is Hf*3600+ Mf*60+Sf, Tsi is H*3600+ M*60+S,
         Dt is Tsf-Tsi, th(Dt,H,RH), tm(RH,M,S),
         ev(H,h), ev(M,m), ev(S,s).

th(DT,DH,RH) :- DH is DT//3600, RH is DT-DH*3600.
tm(RH,DM,DS) :- DM is RH//60, DS is RH-DM*60.

ev(0,_) :- !.
ev(H,h) :- write(H), write($ hora$), bnum(H,h,Str),
           write(Str).
ev(M,m) :- write(M), write($ minuto$), bnum(M,m,Str),
           write(Str).
ev(S,s) :- write(S), write($ segundo$), bnum(S,s,Str),
           write(Str), nl.

bnum(H,h,$s, $) :- H>1, !.
bnum(_,h,$, $) :- !.
bnum(M,m,$s e $) :- M>1, !.
bnum(_,m,$ e $) :- !.
bnum(S,s,$s.$) :- S>1, !.
bnum(_,s,$.$).
```

A lista de palavras consideradas irrelevantes na representação de textos (como preposições, advérbios, etc) faz parte do arquivo *pln.ari* e se torna disponível ao

ambiente do interpretador com o uso da primitiva Prolog, reconsult, representada pela notação [-pln]. Neste arquivo, também foram colocadas partes do programa consideradas invariáveis cujas ausências não prejudicam o entendimento do programa. Por exemplo, uma grande série de sinônimos e uma parte do processamento de *strings* fazem parte do *pln.ari.* Isto foi feito visando reduzir o tamanho do programa e aumentar a sua legibilidade.

```
cllf([],[],Hg) :- !.
cllf([H|T],[Lh|Th],Hg) :- clf(H,Lh,Hg), cllf(T,Th,Hg).

clsf(Arq,Lclf,H) :- lprfx(Lp), clflp(Arq,Lclf,Lp,H).

lprot([at,ci,ec,ep]) :- !.% artes, ciências, economia e esportes

clflp(_,[],[],_) :- !.
clflp(Arq,[Hc|Tc],[Hp|Tp],Hg) :-
  concat([pr,Hp,$.txt$],Aprot),
  open(Ha,Aprot,r), grprot(Ha,Hp), close(Ha),
  dsml(Arq,Hc,Hp,Hg), clflp(Arq,Tc,Tp,Hg).
```

O predicado grprot/2 grava as informações relativas aos protótipos. Elas foram obtidas num outro programa que trata da formação dos protótipos e da obtenção do modelo nebuloso correspondente, a partir dos arquivos textos versando sobre o tema caracterizado pelo protótipo.
O predicado dsml/4 calcula a similaridade entre o texto do arquivo de entrada *Arq* e os textos que caracterizam os protótipos. A variável acima *Hc* é o valor da similaridade entre *Arq* e *Hp* (um dos protótipos).

```
dsml(Arq,Sml,Ref,Hg) :- processa(Arq),
  forij(Arq,Ref,Fx,Fy,H), calc_sml(Arq,Ref,Fx,Fy,Sml,H).
```

O predicado processa/1 é responsável pela leitura do do arquivo de entrada *Arq* e pelo processamento e gravação das *strings* do texto. A leitura é feita através da primitiva de leitura read_line/2. O resultado é a série de *strings* correspondente a cada uma das linhas do texto. Estas séries de *strings* são então trabalhadas com as várias primitivas da linguagem. A primeira primitiva usada, string_lower/2, converte todas as palavras lidas em letras minúsculas. Isto é necessário pois palavras começando com letra maiúscula representam em Prolog variáveis. Outra operação realizada foi a eliminação dos acentos e sinais gráficos (acento agudo, circunflexo, vírgula, trema, etc) e de brancos existentes no texto ou resultantes do processamento do texto realizado pelo sistema. A busca das palavras relevantes de cada tema foi realizada através de diversas operações sobre os textos, reduzindo-se assim o número de palavras representativas de cada classe. Foram criados diversos operadores com a função de modificar, substituir e eliminar palavras. Como exemplo de modificação, temos a supressão de desinências de flexão verbal, de gênero e número. Palavras como *exames*, *examinar*, *examinou*, *examinado* são substituídas pelo radical *exam*. Foi também montado um pequeno

dicionário temático que permite a substituição de determinadas palavras características de um assunto por sinônimos (contido no arquivo *pln.ari*). Após esta filtragem, foram consideradas as cem primeiras palavras do arquivo por ordem de aparição.
O predicado forij/5 determina os $N_x.N_y$ conjuntos nebulosos referentes à partição dos eixos x e y em N_x e N_y termos lingüísticos. A variável *Ref* acima representa o protótipo considerado, F_x e F_y são listas de lista representando os conjuntos nebulosos e *H* é a variável correspondente ao arquivo de saída onde vão ser escritos os resultados.

```
calc_sml(Arq,Ref,Fx,Fy,Sml,H) :-
  recorded(cnp,cnp(Frx,Fry),_), compara(Fx,Fy,Frx,Fry,Sml),
  escreve_sml(Sml,H).

compara(Fx,Fy,Frx,Fry,Sml,H) :- mllpl(Fx,Lx), mllpl(Fy,Ly),
   mllpl(Frx,Lrx),mllpl(Fry,Lry), fator(F),
   csml(Lrx,Lx,Smx), csml(Lry,Ly,Smy),
   soma(Smx,Sx), soma(Smy,Sy), Sxy is (Sx+Sy)/2,
   Sml is round(Sxy/F,4).

mllpl([H|T],NL) :- mllpl(H,LH), mllpl(T,LT),append(LH,LT,NL).
mllpl([],[]).
mllpl(X,[X]).

fator(F) :- lfor(1,Nx,1,Ny), F is Nx*Ny. %número de células
```

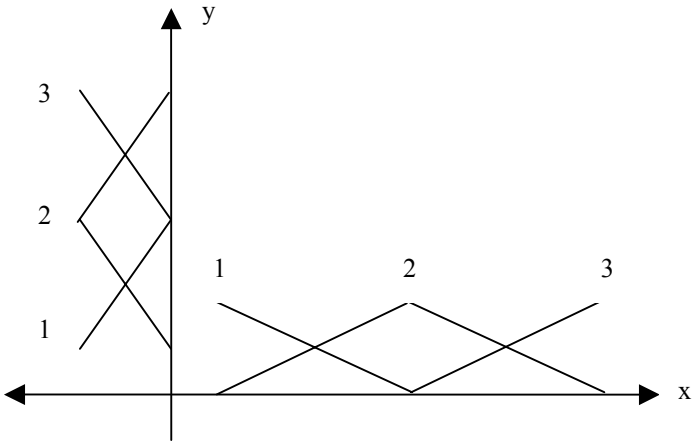
Cada uma das listas Lx, Ly (Lrx, Lry) correspondentes aos conjuntos nebulosos do arquivo de entrada (protótipo) tem F elementos, o número de células decorrente da partição lingüística. No cálculo da similaridade entre uma certa célula dos dois conjuntos (amostra e protótipo) se apenas um dos conjuntos tem grau de pertinência nulo, a similaridade entre eles é nula e é total (um) se o grau de pertinência dos dois for nulo. Nos outros casos, a similaridade é calculada pela fórmulas mostradas nas duas últimas cláusulas.

```
csml([],[],[]) :- !.
csml([Hr|Tr],[Ht|Tt],[Hd|Td]) :- smlx(Hr,Ht,Hh),
                                 csml(Tr,Tt,Td).

smlx(0,Fx0,0) :- Fx0 > 0, !.
smlx(0,_,1) :- !.
smlx(Fx,0,0) :- Fx > 0, !.
smlx(Fx,Fx0,S) :- Fx > 2 * Fx0, !, S is 1 - (Fx-Fx0)/Fx.
smlx(Fx,Fx0,S) :- S is 1 - abs((Fx-Fx0))/Fx0.
```

Exemplo de cáculo dos conjuntos nebulosos

O cálculo do grau de pertinência realizado pelo programa supõe conjuntos nebulosos de forma triangular como na figura abaixo. Nesta figura, temos $N_x=N_y=3$. As duas primeiras cláusulas abaixo se referem ao cálculo do 1^o e 3^o conjunto nebuloso. Enquanto o cálculo do 2^o conjunto nebuloso é realizado pelas duas últimas cláusulas. O segundo argumento H_r é um valor $x \in X$, Universo do Discurso. L_1 e L_2 são os limites inferior e superior do intervalo relativo à célula em questão e o grau de pertinência é *H*. O conjunto X tanto pode ser a coleção de número de ocorrências (representada pelo eixo vertical) como os valores das abscissas. O cálculo dos graus de pertinência, através do predicado *cgp/5*, é realizado para todo o conjunto de palavras relevantes do texto, de acordo com a célula em consideração.

```
cgp(1,Hr,L1,L2,H) :- H is (L2-Hr)/(L2-L1), !.
cgp(3,Hr,L1,L2,H) :- H is (Hr-L1)/(L2-L1), !.
cgp(_,Hr,L1,L2,H) :- Lm is (L1+L2)/2, Hr =< Lm, !,
                     H is (Hr-L1)/(Lm-L1).
cgp(_,Hr,L1,L2,H) :- Lm is (L1+L2)/2, H is (L2-Hr)/(L2-Lm).
```

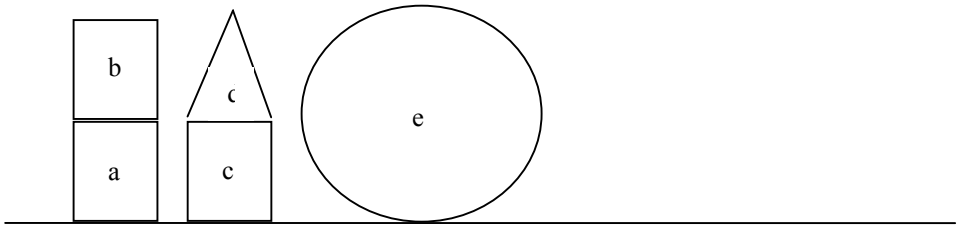


Para ilustração da utilização da Linguagem Prolog nesta aplicação de Reconhecimento de Padrões, foi feito *download* de vários arquivos na internet. Alguns foram separados para a formação dos modelos das classes (protótipos) e o restante foi usado como teste de classificação. Como exemplo, um texto obtido sobre trabalhos científicos, arquivo ci8.txt, foi aplicado ao sistema. O resultado obtido foi a seguinte classificação do texto nos temas artes, ciências, economia e esportes, nesta ordem: L=[0.3724,0.6922,0.4117,0.1942]
O resultado acima indica uma maior similaridade (0.6922) do texto amostra (ci8.txt) com o protótipo correspondente ao tema ciências, conforme esperado.

4.4 Planejamento de ações de um robô manipulador

Vamos construir um cenário, geralmente denominado de Mundo dos Blocos. Este cenário será constituído de cinco objetos, denominados *a*, *b*, *c*, *d*, *e*. Os três primeiros são blocos (paralelepípedos), *d* é uma pirâmide e *e* é uma bola. Inicialmente, eles estão dispostos nas posições 3, 3, 10, 10 e 17, conforme mostra a figura abaixo. Para fins de ilustração, vamos executar o comando colocar o

bloco *a* sobre o bloco *c*. O programa que permite a construção, movimentação e apresentação dos blocos é mostrado abaixo.




```
:- public main/0.

main :- inicializa_movimenta, get0(_), cls.

inicializa_movimenta :- cls, define_janelas,
      tmove(3,56), write($O que deseja fazer?$),
      tmove(4,56), write($colocar objeto $), read(X),
      tmove(5,56), write($         sobre $), read(Y),
      assertz(pvez(0)), inicia, mostra_figuras,
      abolish(pvez/1), assertz(pvez(10)),
      write($aperte uma tecla$), get0(_), colocar_sobre(X,Y),
      write($aperte uma tecla$), get0(_), mostra_figuras.

define_janelas :-
       define_window(inicial,'',(0,0),(24,79),(31,0)),
       define_window(msg_usr,'',(3,56),(9,79),(31,31)),
       current_window(_,inicial), tmove(0,10),
       write($ SISTEMA ESPECIALISTA - MUNDO DOS BLOCOS $).

inicia :- apaga, grfg(bloco,[a,b,c]), grfg(piramide,[d]),
       grfg(bola,[e]), grspp([a/mesa,c/mesa,e/mesa,b/a,d/c]),
       grpos([mao/0,a/3,c/10,e/17]),grpl(28),
       recordz(str,str(livre),_), tmove(3,1).

grfg(_,[]).
grfg(Fg,[H|T]) :- recordz(eh,fg(H,Fg),_), grfg(Fg,T).

grspp([]).
grspp([Ac/Bb|T]) :- recordz(spp,spp(Ac,Bb),_), grspp(T).

grpos([]).
grpos([Obj/Pos|T]) :- recordz(pos,obj(Obj,Pos),_), grpos(T).

grpl(X) :- recordz(pli,X,_).
```

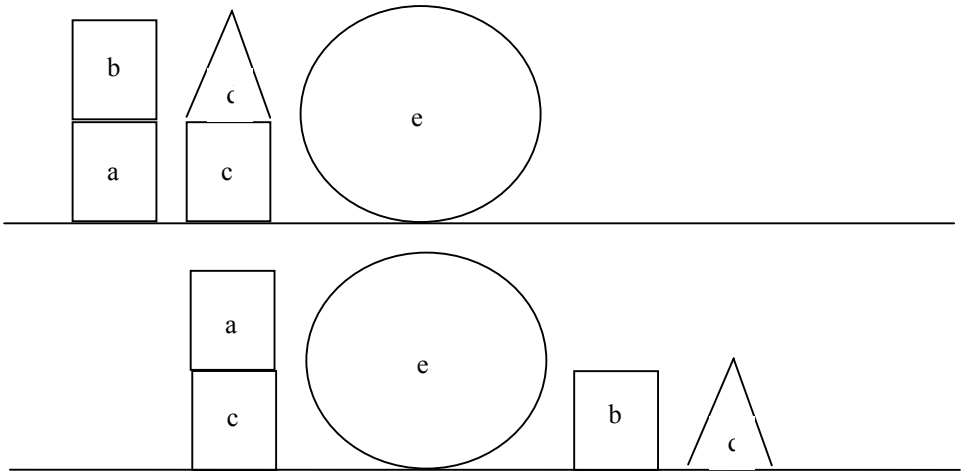
```
colocar_sobre(X,X) :- !, write($ Objetos identicos! $), nl.
colocar_sobre(Obj_c,Base) :- recorded(eh,fg(Base,bloco),_),
                        recorded(spp,spp(Obj_c,Base),_), !,
                        write($Jah estah lah$), nl.
colocar_sobre(Obj_c,Base) :- recorded(eh,fg(Base,bloco),_),
                           !, livre(Obj_c), livre(Base),
                            posicao(Base,Pos_b),
                            colocar_em(Obj_c,Pos_b,Base).
colocar_sobre(_,Base) :- recorded(eh,fg(Base,Fg),_),
    write($Nao eh possivel, pois a base eh $), write(Fg), nl.

livre(X) :- recorded(spp,spp(X,Y),R), Y \== mesa, !,
            colocar_sobre_mesa(X), replace(R,spp(X,mesa)).
livre(X) :- recorded(spp,spp(Y,X),R), !, relata(X,Y),
            colocar_sobre_mesa(Y), replace(R,spp(Y,mesa)).
livre(X) :- relata(X,livre).

relata(X,livre) :- write($Ok, o topo do objeto $), write(X),
                   write($ estah livre$),nl.
relata(X,Y) :- write($O bloco $), write(X),
        write($ suporta o objeto $), write(Y), write($, $),
        write($achar espaco na mesa para $), write(Y), nl.
relata(r,Obj_c,Pos_c) :-
        write($Encaminhar o robo para a posicao $),
        write(Pos_c), write($ do objeto $), write(Obj_c), nl.
relata(Pos_b,Obj_c,Obj_b) :- write($pegar o objeto $),
        write(Obj_c), write($, leva-lo para a posicao $),
        write(Pos_b), write($ e coloca-lo sobre o objeto $),
        write(Obj_b), nl.

colocar_sobre_mesa(Obj) :- recorded(pli,X,R), N is X+7,
                           recordz(pos,obj(Obj,X),_),
                           replace(R,N).

posicao(Obj,Pos) :- recorded(pos,obj(Obj,Pos),_), !.
posicao(Obj,Pos) :- recorded(spp,spp(Obj,X),_),
                    posicao(X,Pos).

colocar_em(Obj_c,Pos_b,Base) :- apagapos(Obj_c,Pos_c),
          move_r(Pos_b), pega_r_obj(Obj_c,Base,Pos_b),
          recorded(spp,spp(Obj_c,_),E),
          replace(E,spp(Obj_c,Base)).

apagapos(Obj_c,Pos_c) :- recorded(pos,obj(Obj_c,Pos_c),R), !,
                         erase(R), relata(r,Obj_c,Pos_c).
apagapos(Obj_c,Pos_c) :- posicao(Obj_c,Pos_c),
                         relata(r,Obj_c,Pos_c).
```

```
move_r(Pos) :- recorded(pos,obj(mao,_),R),
               replace(R,obj(mao,Pos)).

pega_r_obj(Obj_c,Obj_b,Pos_b) :- recorded(str,str(_),R),
    replace(R,str(peg,Obj)), recorded(spp,spp(Obj_c,_),S),
    replace(S,spp(Obj,mao)), relata(Pos_b,Obj_c,Obj_b).

% Predicados definidos para consultas
config_pecas :- recorded(pos,obj(Obj,Pos),_),
   [! recorded(eh,fg(Obj,Fg),_), escrpos(Fg,Obj,Pos), nl,
      sobre(Obj_c,Obj,Pos) !], fail.
config_pecas.

onde(Obj) :- posicao(Obj,P), recorded(spp,spp(Obj,Obj_b),_),
             suporta(Obj,Obj_c), escr_obj(Obj,P,Obj_b,Obj_c).

onde_todos(L) :- busca_todos([a,b,c,d,e],M), sort(M,S),
                 troca(S,L).

busca_todos([],[]).
busca_todos([H|T],[P/H|R]) :- posicao(H,P), busca_todos(T,R).

troca([],[]).
troca([A/B|T],[B/A|R]) :- troca(T,R).

escrpos(Fg,Obj,Pos) :- write(Fg), write($ $), write($"$),
                       write(Obj), write($"$), write($ na
                       posicao $), write(Pos).

sobre(Obj_c,Obj_b,Pos) :- recorded(spp,spp(Obj_c,Obj_b),_),
        recorded(eh,fg(Obj_c,Fg),_), escrpos(Fg,Obj_c,Pos),
        write($, sobre $), write($"$), write(Obj_b),
        write($"$), nl.

escr_obj(Obj,P,Obj_b,topo_livre) :- !, writefig(Obj),
   write($ estah na posicao $), write(P), writesup(Obj_b),
   writefig(Obj_b), nl.

escr_obj(Obj,P,Obj_b,Obj_c) :- fg(Obj,Fg), write(Fg),
   write($ "$), write(Obj), write($" $), write($ estah na
   posicao $), write(P), writesup(Obj_b), write(Obj_b), nl,
   write($ e eh suporte para $), fg(Obj_c,F), write(F),
   write($ "$), write(Obj_c), write($" $), nl.

writefig(mesa) :- !, write($mesa $).
writefig(Obj) :- fg(Obj,Fg), write(Fg), write($ "$),
```

```
                write(Obj), write($" $).

writesup(mesa) :- !, write($, eh suportado pela $).
writesup(_) :- write($, eh suportado por $).

fg(Obj,Fg) :- recorded(eh,fg(Obj,Fg),_).

suporta(Obj_b,Obj_c) :- recorded(spp,spp(Obj_c,Obj_b),_), !.
suporta(Obj_b,topo_livre).

obtem_obj_sob(Obj_b,Obj_c) :-
          recorded(spp,spp(Obj_c,Obj_b),_).
obtem_obj_sobre(Obj_c,Obj_b) :-
          recorded(spp,spp(Obj_c,Obj_b),_).

mostra_figuras :- [! lista_pecas(L) !], membro(Obj,L),
                  [! posxy(Obj,Px,Py), pvez(K), NPx is Px-K,
                     dfig(Obj,NPx,Py) !], fail.
mostra_figuras :- pvez(V), Y is V*6, tmove(20,Y).

lista_pecas([a,b,c,d,e]).

posxy(Obj,15,Py) :- recorded(spp,spp(Obj,mesa),_), !,
                    posicao(Obj,Pos), Py is Pos-3.
posxy(Obj,12,Py) :- posicao(Obj,Pos), Py is Pos-3.

dfig(Obj,Px,Py) :- recorded(eh,fg(Obj,piramide),_),!,
                   dpira(Obj,Px,Py).
dfig(Obj,Px,Py) :- recorded(eh,fg(Obj,bola),_),!,
                   Npx is Px-4, Npy is Py+5,
                   dbola(Obj,Npx,Npy).
dfig(Obj,Px,Py) :- dbloc(Obj,Px,Py).

dpira(Obj,Px,Py) :- Npy is Py+3, Npx is Px-1,
                    trpira(Obj,Npx,Npy).

trpira(Ch,X,Y) :- tmove(X,Y), wp,
     ncoord(X,1,Y,-1,Nx,Ny), tmove(Nx,Ny), wp,
     ncoord(X,1,Y,1,Nx1,Ny1), tmove(Nx1,Ny1), wp,
     ncoord(X,2,Y,-2,Nx2,Ny2), tmove(Nx2,Ny2), wp,
     ncoord(X,2,Y,0,Nx3,Ny3), tmove(Nx3,Ny3), write(Ch),
     ncoord(X,2,Y,2,Nx4,Ny4), tmove(Nx4,Ny4), wp,
     ncoord(X,3,Y,-3,Nx5,Ny5), tmove(Nx5,Ny5), wp,
     ncoord(X,3,Y,3,Nx6,Ny6), tmove(Nx6,Ny6), wp,
     tmove(20,0).

ncoord(X,Dx,Y,Dy,Nx,Ny) :- Nx is X+Dx, Ny is Y+Dy.
```

```
wp :- write($.$).

dbola(Obj,X,Y) :- tmove(X,Y), wp,
     iv(X,1,Xm1), iv(Y,4,Ym1), iv(Y,-4,Yn1), tmove(Xm1,Yn1),
     wp, tmove(Xm1,Ym1), wp, iv(X,2,Xm2), iv(Y,5,Ym3),
     iv(Y,-5,Yn3), tmove(Xm2,Yn3), wp, tmove(Xm2,Ym3), wp,
     iv(X,3,Xm3), tmove(Xm3,Y), write(Obj), iv(X,4,Xm4),
     tmove(Xm4,Yn3), wp, tmove(Xm4,Ym3), wp, iv(X,5,Xm5),
     tmove(Xm5,Yn1), wp, tmove(Xm5,Ym1), wp, iv(X,6,Xm6),
     tmove(Xm6,Y), wp.

iv(V,D,NV) :- NV is V+D.

dbloc(c,X,Y) :- tmove(X,Y), write($|$), X1 is X+1,
                tmove(X1,Y), write($|$), X2 is X1+1,
                tmove(X2,Y), write($|$), write($.....$),
                Z is Y+6, tmove(X,Z), write($|$),
                tmove(X1,Z), write($|$), tmove(X2,Z),
                write($|$), B is Z-3, tmove(X1,B),
                write(c), X3 is X-1, W is Z-5, tmove(X3,W),
                write($....$).

dbloc(C,X,Y) :- tmove(X,Y), write($|$), X1 is X+1,
                tmove(X1,Y), write($|$), X2 is X1+1,
                tmove(X2,Y), write($|$), write($.....$),
                Z is Y+6, tmove(X,Z), write($|$),
                tmove(X1,Z), write($|$),
                tmove(X2,Z), write($|$),B is Z-3,
                tmove(X1,B), write(C), X3 is X-1, W is Z-5,
                tmove(X3,W), write($.....$).

membro(X,[X|_]).
membro(X,[_|T]) :- membro(X,T).

apg(X) :- eraseall(X).

apaga :- abolish(pvez/1), apg(eh), apg(pos), apg(spp),
apg(pli), apg(str).
```

Configuração inicial e após o comando colocar *a* sobre *c*.



4.5 *Toy Problems*

Os jogos sempre desempenharam um papel importante no desenvolvimento de técnicas de IA. Apesar de geralmente simples e bem estruturados, o espaço de estados da maioria dos jogos é constituído de um número muito grande de estados, dificultando sua exploração na procura de uma solução. No jogo da velha, o primeiro movimento apresenta nove alternativas, o segundo, 8 e assim por diante. Assim, este jogo admite 9! diferentes possibilidades. O espaço de estados do jogo de damas apresenta 10^{40} diferentes possibilidades e o do xadrez, 10^{120}. Estes espaços são tão grandes que, para evitar a explosão combinatória (e a conseqüente exaustão dos recursos computacionais), torna-se imperativo o uso de conhecimento especializado que possibilite focalizar a busca, eliminando ramos da árvore de procura e limitando o número de sucessores. O conhecimento heurístico desempenha um papel essencial na procura nestes espaços.
Alguns problemas clássicos que estão muito ligados ao início do desenvolvimento da Inteligência Artificial serão tratados a seguir. Estes problemas são, geralmente, simples e muito importantes pois servem para desenvolver e ilustrar diversas técnicas de representação e solução de problemas. Com o conhecimento destas técnicas de IA, problemas reais, muito mais complexos, podem ser abordados.

4.5.1 Torre De Hanói

Um dos problemas clássicos é o problema da Torre de Hanói. A Torre de Hanói, jogo criado pelos matemáticos franceses E. Lucas e De Parville em 1894, consiste num conjunto de três pinos fixos numa base comum. Num dos pinos, 7 peças furadas (discos) estão enfiadas em ordem decrescente de tamanho, de baixo para cima. O desafio consiste em transportar uma a uma essas sete peças para um dos outros pinos num menor número possível de movimentos. Não é permitido, em nenhuma etapa, que uma peça fique pousada sobre outra de menor tamanho. O problema da Torre de Hanói é um exemplo clássico da aplicação da técnica de decomposição de um problema em subproblemas mais simples. A solução deste

problema em Prolog ilustra também de forma bastante clara a conveniência e a eficácia do uso da recursividade. Podemos enunciar este problema nos seguintes termos. N discos de tamanhos diferentes estão dispostos no eixo A, estando os menores sobre os maiores. Deseja-se transportá-los para o eixo C, mantendo a mesma disposição relativa entre eles (os menores sobre os maiores). As condições do problema são as seguintes: 1) o eixo B pode ser usado para armazenar temporariamente os discos, 2) apenas um disco pode ser transportado de cada vez e 3) nenhum disco pode estar sobre um disco menor.

A solução deste problema baseia-se na técnica de redução de problemas. Ao invés de tentar resolver diretamente o problema com N discos, resolveremos o problema mais simples envolvendo N-1 discos. Se N-1 for 1, a solução é trivial. Caso contrário, temos de resolver um problema de M=N-1 discos, mais simples que o original. Repetimos o procedimento anterior, isto é, resolvemos um problema com M-1 discos até que este problema seja trivial (M-1=1). A codificação em Prolog apresentada abaixo mostra os discos sendo transferidos. A rapidez desta transferência pode ser controlada por um *predicado Prolog* que simula uma pausa. A visualização é possível através dos desenhos de traços formando os discos. Os termos esquerda, centro e direita correspondem aos eixos A, B e C respectivamente.

```
                                    %esquerda, centro, direita
hanoi(N) :- inicia, move(N,e,c,d),nl, nl,
         write($                    ******* FIM *******$), nl.

inicia :- cls, apg(loc)*, recordz(loc,loc(e,[p,m,g]),_),
          recordz(loc,loc(c,[]),_), recordz(loc,loc(d,[]),_),
          desbarra(0), desfig(e,[p,m,g]), desbarra(20),
          desfig(c,[]), desbarra(40), desfig(d,[]), nl, nl,
          pause(N).
% p: disco pequeno, m: médio, g: grande

pause(N) :- parada(N),ctr_is(8,S), ctr_set(8,N),
            repeat,
            ctr_dec(8,0), !, ctr_set(8,S).

desbarra(H) :- H7 is H+7, tmove(0,H7), write($|$),
               tmove(1,H7), write($|$), tmove(2,H7),
               write($|$), tmove(3,H7), write($|$),
               tmove(4,H7), write($|$), tmove(5,H),
               write($_______$), write($|$),
               write($_______$).

desfig(_,[]) :- !.
desfig(P,L) :- pos(P,H), vlh(H,H3,H4,H5,H7,H8),
               des_disco(H3,H4,H5,H7,H8,L).
```

```
pos(e,0) :- !.
pos(c,20) :- !.
pos(d,40).

vlh(H,H3,H4,H5,H7,H8) :- ih(H,3,H3), ih(H,4,H4), ih(H,5,H5),
                         ih(H,7,H7), ih(H,8,H8)

ih(X,N,XN) :- XN is X+N.

des_disco(H3,H4,H5,H7,H8,[p,m,g]) :- !, tmove(2,H5),
        write($__$), tmove(2,H7), write($|$), tmove(2,H8),
        write($__$), tmove(3,H4), write($___$), tmove(3,H7),
        write($|$), tmove(3,H8), write($___$), tmove(4,H3),
        write($____$), tmove(4,H7), write($|$), tmove(4,H8),
        write($____$).

des_disco(H3,H4,H5,H7,H8,[m,g]) :- !, tmove(3,H4),
        write($___$), tmove(3,H7), write($|$), tmove(3,H8),
        write($___$), tmove(4,H3), write($____$),
        tmove(4,H7), write($|$), tmove(4,H8), write($____$).

des_disco(H3,H4,H5,H7,H8,[p,m]) :- !, tmove(2,H5),
        write($__$), tmove(2,H7), write($|$), tmove(2,H8),
        write($__$), tmove(3,H4), write($___$), tmove(3,H7),
        write($|$), tmove(3,H8), write($___$).

des_disco(H3,H4,H5,H7,H8,[p]) :- !, tmove(2,H5), write($__$),
        tmove(2,H7), write($|$), tmove(2,H8), write($__$).

des_disco(H3,H4,H5,H7,H8,[m]) :- !,  tmove(3,H4),
        write($___$), tmove(3,H7), write($|$), tmove(3,H8),
        write($___$).

des_disco(H3,H4,H5,H7,H8,[g]) :- !,  tmove(4,H3),
        write($____$), tmove(4,H7), write($|$), tmove(4,H8),
        write($____$).

escr(X,Y) :- cls, recorded(loc,loc(X,Lx),Rx),
             recorded(loc,loc(Y,Ly),Ry),
             transf(Lx,Ly,NLx,NLy), replace(Rx,loc(X,NLx)),
             replace(Ry,loc(Y,NLy)), discos(e,Le),
             discos(c,Lc), discos(d,Ld), cls,
             desbarra(0), desfig(e,Le), desbarra(20),
             desfig(c,Lc), desbarra(40), desfig(d,Ld),
             pause(N), nl, nl.

transf([H|T],Ly,T,[H|Ly]).
```

```
discos(X,LX) :- recorded(loc,loc(X,LX),_).

move(1,E,_,D) :- escreve(E,D). %leva um disco da esq p/ a dir

move(N,E,C,D) :- N1 is N-1,    %reduz o numero de discos
   move(N1,E,D,C), %leva  N-1 discos menores da esq p/ centro
   escr(E,D),      %leva o maior disco da esq p/ a direita
   move(N1,C,E,D). %leva N-1 discos menores do centro p/ dir
```

4.5.2 O Problema das N Rainhas

Este problema consiste em posicionar N rainhas num tabuleiro NxN sem que elas se ataquem. A solução apresentada abaixo é geral, vale para qualquer N superior a três. Após a codificação, são apresentadas as soluções para os casos de N entre 4 e 8. O número de soluções possíveis para estes números de rainhas é fornecido na tabela abaixo.

Número de rainhas	4	5	6	7	8
Número de soluções	2	10	4	40	92

Devido ao grande número de soluções, nos casos de 7 e 8 rainhas, são apresentadas apenas a primeira e a última solução encontrada.

```
resolve(N) :- ctr_set(0,1), int_text(N,NS),
              concat([rnh,NS,$.doc$],Narq), create(H,Narq),
              escr_cab(N,H), solucao(N,H).

escr_cab(N,H) :- write($ Problema das $), write(N),
                 write($ Rainhas$), nl,
                 write(H,$ Problema das $), write(H,N),
                 write(H,$ Rainhas$), nl(H).

solucao(N,H) :- [! gera_l_nv(1,N,L) !], resolve(N,L),
   [! ctr_inc(0,I), write($Solucao $), write(I), write($: $),
      write(L), nl, write(H,$Solucao $), write(H,I),
      write(H,$: $),  write(H,L), nl(H) !], fail.

solucao(_,H) :- close(H).

gera_l_nv(N,N,[N/Y]).
gera_l_nv(N1,N2,[N1/Y|T]) :- N1 < N2, M is N1+1,
gera_l_nv(M,N2,T).

resolve(N,[]).
resolve(N,[X1/Y1|T]) :- gera_l_num(1,N,L), membro(Y1,L)*,
                        n_ataca(N,X1/Y1,T), resolve(N,T).
```

```
gera_l_num(N,N,[N]).
gera_l_num(N1,N2,[N1|T]) :- N1 < N2, M is N1+1,
gera_l_num(M,N2,T).

n_ataca(N,_,[]).
n_ataca(N,X1/Y1,[X2/Y2|T]) :- gera_l_num(1,N,L),
                membro(Y2,L)*,
                [! Y1 =\= Y2, abs(X2-X1) =\= abs(Y2-Y1) !],
                n_ataca(N,X1/Y1,T).
```

Soluções encontradas para diferentes valores de N:

```
  Problema das 4 Rainhas
Solucao 1: [1 / 2,2 / 4,3 / 1,4 / 3]
Solucao 2: [1 / 3,2 / 1,3 / 4,4 / 2]

  Problema das 5 Rainhas
Solucao 1: [1 / 1,2 / 3,3 / 5,4 / 2,5 / 4]
Solucao 2: [1 / 1,2 / 4,3 / 2,4 / 5,5 / 3]
Solucao 3: [1 / 2,2 / 4,3 / 1,4 / 3,5 / 5]
Solucao 4: [1 / 2,2 / 5,3 / 3,4 / 1,5 / 4]
Solucao 5: [1 / 3,2 / 1,3 / 4,4 / 2,5 / 5]
Solucao 6: [1 / 3,2 / 5,3 / 2,4 / 4,5 / 1]
Solucao 7: [1 / 4,2 / 1,3 / 3,4 / 5,5 / 2]
Solucao 8: [1 / 4,2 / 2,3 / 5,4 / 3,5 / 1]
Solucao 9: [1 / 5,2 / 2,3 / 4,4 / 1,5 / 3]
Solucao 10: [1 / 5,2 / 3,3 / 1,4 / 4,5 / 2]

  Problema das 6 Rainhas
Solucao 1: [1 / 2,2 / 4,3 / 6,4 / 1,5 / 3,6 / 5]
Solucao 2: [1 / 3,2 / 6,3 / 2,4 / 5,5 / 1,6 / 4]
Solucao 3: [1 / 4,2 / 1,3 / 5,4 / 2,5 / 6,6 / 3]
Solucao 4: [1 / 5,2 / 3,3 / 1,4 / 6,5 / 4,6 / 2]

Problema das 7 Rainhas
Solucao 1: [1 / 1,2 / 3,3 / 5,4 / 7,5 / 2,6 / 4,7 / 6]
        …
Solucao 40: [1 / 7,2 / 5,3 / 3,4 / 1,5 / 6,6 / 4,7 / 2]


  Problema das 8 Rainhas
Solucao 1: [1 / 1,2 / 5,3 / 8,4 / 6,5 / 3,6 / 7,7 / 2,8 / 4]
        …
Solucao 92: [1 / 8,2 / 4,3 / 1,4 / 3,5 / 6,6 / 2,7 / 7,8 / 5]
```

4.5.3 *Eight Puzzle*

Muitas aplicações de IA (controlar as ações de um robô, por exemplo) envolvem uma seqüência de operações. O Quebra-cabeça conhecido como 8-*Puzzle*, por ser bastante simples e bem definido, é muito útil para ilustrar idéias básicas que podem ser usadas para resolver problemas mais complexos. Neste problema temos um quadrado 3x3 e 8 peças móveis, numeradas de 1 a 8. O movimento legal do jogo consiste em deslocar uma peça para ocupar a posição vazia do quadrado. Com a posição vazia (denominada daqui em diante por peça zero) na posição central, pode se movimentar para baixo (*b*), cima(*c*), esquerda(*e*) e direita(*d*). Movendo uma peça para cima, ocupando uma posição vazia (branco), diremos que o branco moveu-se para baixo. O problema 8-*Puzzle* consiste em movimentar as peças a partir de uma configuração inicial até atingir uma certa condição meta. Esta condição meta pode ser, por exemplo, uma configuração em que a soma dos números das peças ocupando a diagonal principal atinja um valor pré-definido. No nosso caso, a condição meta será atingir a configuração mostrada na figura abaixo. Um estado do problema é uma configuração possível das peças no quadrado. No processo de solução deste problema, o conjunto destes estados define o espaço de estados do problema. Há 9! estados possíveis, divididos em dois conjuntos de 9!/2 estados. Se o estado A pertence ao primeiro conjunto e estado meta pertence ao segundo, não é possível partir do estado A e atingir a meta. A escolha de uma forma de representação adequada para o problema é determinante para a sua solução. Há pelo menos duas formas (interessantes) de representação dos estados do problema. Numa representação, a peça P_k (k=0,1...8, onde a peça zero é o branco) é representada por i/j, indicando a i-ésima linha e j-ésima coluna ocupada por ela. As posições de todas peças são armazenadas numa lista. Na outra representação, formamos uma lista contendo as peças nas posições 1/1, 1/2, ..., 3/2, 3/3 (linha por linha da matriz). Os movimentos legais *b*,*c*,*e*,*d* dependem do estado corrente. Por exemplo, a decisão (chamada de regra de produção) *mova o branco para a esquerda* só é possível se este estiver na segunda ou terceira coluna. Esta condição é facilmente verificável na primeira representação.

Condição inicial	Condição Meta
2 8 3	1 2 3
1 6 4	8 0 4
7 0 5	7 6 5

```
Condição inicial
Repr. por coordenadas: [2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3]
Repr. pelas linhas: [2,8,3,1,6,4,7,0,5]
Condição meta
Repr. por coordenadas: [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]
Repr. pelas linhas: [1,2,3,8,0,4,7,6,5]
```

4.5.3.1 Mudança de Representação dos estados do 8 *Puzzle*

A segunda representação é muito apropriada para descrever a evolução dos estados do problema durante o processo de solução. É importante, pois, que se possa passar facilmente de uma forma para outra. Foram definidos, abaixo, os predicados Prolog convertexypecas(N) e convertepecasxy(N) que fazem esta conversão. O número inteiro N identifica a configuração atual.

```
cxp(N) :- convertexypecas(N).
convertexypecas(N) :- abolish(coord/3), le_confxy(N,Rxy),
                      write($Reprxy = $),
                      write(Rxy), nl, gr(Rxy,0), escr([],Rp),
                      write($Repr_pecas = $), write(Rp), nl.
gr([],_).
gr([X/Y|T],I) :- NL is 4-Y, NC is X, assertz(coord(I,NL,NC)),
                 II is I+1, gr(T,II).

le_confxy(1,[2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3]).
%Repr_pecas= [2,8,3,1,6,4,7,0,5]
le_confxy(2,[2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2]). %Reprxy
%Repr_pecas= [2,1,6,4,0,8,7,5,3]

le_confxy(meta,[2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).
%Repr_pecas= [1,2,3,8,0,4,7,6,5]

escr(Li,Rp) :- escr(Li,Rp,1).

escr(L,L,4).
escr(Li,Rp,I) :- escrc(I,1,Li,L), II is I+1, nl,
                 escr(L,Rp,II).

escrc(I,4,L,L).
escrc(I,J,Li,L) :- coord(E,I,J),inclui_fim(E,Li,NL)*,
                   write(E), write($  $), JJ is J+1,
                   escrc(I,JJ,NL,L).

cpx(N) :- convertepecasxy(N).

convertepecasxy(N) :- apaga, le_confpe(N,Rpe),
                      write($Reprpecas = $),
                      write(Rpe), nl, mc(Rpe,1),
                      ctr_set(10,1), escr_lin(Rpe), nl,
                      detnrepr(Rxy,[],0), write($Reprxy = $),
                      write(Rxy), nl.
mc([],_).
mc([H|T],I) :- NI is I-1,Y is 3-NI//3, detx(I,X),
               assertz(coord(H,X,Y)), II is I+1, mc(T,II).
detx(I,X) :- Z is I mod 3, obtemx(Z,X).
```

```
obtemx(0,3) :- !.
obtemx(Z,Z).

escr_lin([]).
escr_lin([H|T]) :- ctr_inc(10,I), dsw(H,I), escr_lin(T).

dsw(H,I) :- I<4, !, write(H), write($  $).
dsw(H,I) :- ctr_set(10,2), nl, write(H), write($  $).

detnrepr(Rxy,Rxy,9).
detnrepr(Rxy,Li,I) :- coord(I,X,Y), inclui_fim(X/Y,Li,NL)*,
                      II is I+1, detnrepr(Rxy,NL,II).

le_confpe(1,[2,8,3,1,6,4,7,0,5]).
le_confpe(2,[2,1,6,4,0,8,7,5,3]).
le_confpe(meta,[1,2,3,8,0,4,7,6,5]).
```

4.5.3.2 O Algoritmo A* aplicado ao 8-*Puzzle*

O algoritmo A* é um algoritmo que procura um caminho mínimo entre dois nós de um grafo conexo. Ele pode ser implementado através dos passos abaixo.

Procedimento de Procura em Grafo

1. Crie um grafo *G* com o nó *s* e uma Lista Aberto contendo o nó *s*.
2. Crie uma Lista Fechado, inicialmente vazia ([ ]).
3. Se Aberto = [ ], *Falha*
4. Seja *n* = Melhor(Aberto). Retire *n* de Aberto e coloque-o em Fechado.
5. Se *n* satisfaz a meta, indique o caminho $n \rightarrow s$.
6. Seja *M* o conjunto de sucessores de *n* que não são seus ancestrais.
7. Estabeleça um ponteiro para *n* partindo dos nós de *M* que não pertenciam a *G*. Acrescente estes nós à Lista Aberto. Para cada membro de *M* que já pertencia à Lista Aberto (ou Fechado) decida redirecionar ou não seu ponteiro para *n*. Para cada membro de *M* em Fechado, decida redirecionar ou não seus descendentes.
8. Ordene a Lista Aberto segundo algum critério heurístico.
9. Vá para 3.

Este procedimento é suficientemente geral para abranger uma ampla variedade de algoritmos de procura em grafo. Ele gera um grafo explícito *G*, grafo de procura e um subconjunto *T* de *G*, árvore de procura. A árvore de procura é formada pelos ponteiros definidos no passo 7. Todo nó (exceto a raiz) tem um ponteiro para um de seus pais em *G*, formando a árvore de procura *T*. Os nós na Lista Aberto (no passo 3) são nós que não foram ainda selecionados para expansão. Estes nós são ordenados no passo 8. A eficiência do algoritmo de procura depende da disponibilidade de informação heurística sobre o domínio do problema que oriente este ordenamento. Se nenhuma informação é disponível, é utilizado algum

ordenamento arbitrário. No passo 4, o melhor nó é escolhido e vai ser expandido no passo 6. Neste passo, é também feita a verificação se nenhum nó sucessor gerado na expansão do nó *n* é seu ancestral. Quando o nó escolhido para expansão satisfaz a meta, o algoritmo pára e o caminho solução pode ser obtido, invertendo-se o sentido dos ponteiros do nó *t* ao nó *s*. O redirecionamento dos ponteiros no passo 7 visa manter uma árvore de caminho (custo) mínimo. Se o grafo *G* que está sendo pesquisado é uma árvore, no momento da expansão de um nó *n*, nenhum dos sucessores encontrados no passo 6 pertencerá a *G* e eles serão instalados na árvore de procura com um ponteiro para *n*. Deste modo não haverá nunca necessidade de redirecionamento. Se o grafo a ser pesquisado não é uma árvore, é possível que se encontre no momento da expansão algum nó já incluído em Aberto ou Fechado (portanto já incluído em *G*). Neste caso, o ponteiro para este nó poderá mudar se o novo caminho encontrado tiver custo menor. O mesmo é feito para os descendentes deste nó.

Função de Avaliação

Na busca de solução para um problema, procura-se usar todo o conhecimento disponível sobre o domínio da aplicação e, em especial, aquelas informações que são potencialmente eficazes (heurísticas). Uma maneira de se usar heurísticas para resolver os problemas é tentar quantificá-las através de uma função de avaliação. Esta função mede a qualidade dos estados do problema durante o processo de solução. Chamando de nó *n*, o estado corrente, o valor f(n) avalia a potencialidade ("promessa" ou "probabilidade") do nó *n* em relação a pertencer ao caminho ótimo. O valor de f(n) pode ser usado então para ordenar os nós da Lista Aberto. Um exemplo de tal função para resolver o 8-*Puzzle* é dado abaixo:

$$f(n) = prof(n) + npmc(n)$$

onde prof(n) representa a profundidade do nó *n* e npmc(n), o número de peças mal colocadas (em relação à configuração meta) do nó *n*.

O algoritmo A

Seja f(.) uma função de avaliação e f(n) a soma dos seguintes estimativas de custos:

- custo g(n) do caminho mínimo do nó *s* até o nó *n* e
- custo h(n) do caminho mínimo do nó *n* até o nó terminal *t*.

$$f(n) = g(n) + h(n)$$

O valor da função de avaliação em *n*, f(*n*), representa, pois, a estimativa do custo do caminho mínimo passando por *n*. Quando o nó *n* está sendo expandido, seus nós sucessores n_i, colocados em Aberto, são avaliados pela função f(.). O nó de Aberto com menor valor na função f(.) é o próximo escolhido para expansão. O algoritmo de procura que utiliza uma função de avaliação da forma acima para ordenar os nós (passo oito do procedimento geral de procura) é chamado de Algoritmo A.

O algoritmo A*

Consideremos um algoritmo A com as estimativas de custo g(n), h(n) e f(n) para um nó *n* com a interpretação dada acima e os seguintes valores reais ótimos

- $g^*(n)$: custo real do caminho mínimo do nó *s* ao nó *n*,
- $h^*(n)$: custo real do caminho mínimo do nó *n* ao nó *t*.

A componente h(n) representa a informação heurística disponível sobre o domínio do problema e é uma estimativa do valor real $h^*(n)$ não conhecido, a priori. Uma estimativa natural para g(n) é a soma dos custos dos arcos no caminho de *s* a *n* na árvore de procura corrente. Podemos facilmente verificar que:

$$(\forall n),\ g(n) \geq g^*(n)$$

Se a seguinte desigualdade é satisfeita

$$(\forall n),\ h(n) \leq h^*(n)$$

então o algoritmo A é denominado A*. O algoritmo A^* fornece uma solução de caminho mais curto do nó *s* ao nó *t* (sempre que existir um caminho entre o nó de partida *s* e o nó terminal *t*) e por isto é de grande interesse para os sistemas de busca.
O algoritmo que faz a procura pela largura (g(n)=profundidade do nó, h(n)=0) pára encontrando a solução ótima, mas é grosseiramente ineficiente pois o número de nós expandidos é muito grande. Ele faz uma procura exaustiva. Por isto, a garantia da otimalidade obtida na procura pela largura não representa uma solução eficiente do ponto de vista computacional. A seleção de uma boa heurística é fundamental para se dotar de poder heurístico o algoritmo de procura. Quanto maior o valor de h(n), mais informação heurística se dispõe e assim um número menor de nós será expandido. Um menor espaço de memória será necessário, aumentando a eficiência do algoritmo. Quando se dispõe de muita informação, grande será o custo para processá-la, o que tende a reduzir a eficiência do algoritmo. Se utilizarmos uma função heurística h(n) de valor baixo, inferior a h*(n), garantimos a otimalidade mas teremos que expandir muitos nós (geralmente inviável). Seguem alguns resultados fundamentais acerca do algoritmo A^*.

1. O algoritmo A^* sempre pára quando o grafo é finito.
2. Se existe um caminho $s \rightarrow t$, o algoritmo pára mesmo se o grafo for infinito.
3. O algoritmo A* é admissível: encontra o caminho ótimo de $s \rightarrow t$ e pára, sempre que o nó *t* for acessível a partir do nó *s*.
6. Sejam A^*_1 e A^*_2 duas versões do algoritmo A*. Dizemos que um algoritmo A^*_2 é mais informado que o algoritmo A^*_1 se $h_2(n) > h_1(n)$ para $\forall n \neq t$. Se A^*_2 é mais informado que A^*_1 então, ao final do processo de procura, todo nó expandido por A^*_2 terá sido expandido por A^*_1.

7. Se a Restrição Monotônica é satisfeita, os valores da função de avaliação f(.) nos nós expandidos é não decrescente, isto é, se os nós expandidos são s, n_1, n_2 ... então $f(s) \leq f(n_1) \leq f(n_2)$....

   Restrição Monotônica:

   h(t) = 0 $h(n_i) \leq h(n_j) + c(n_i, n_j)$, n_j sucessor de n_i.

Esta desigualdade exige que a queda no valor de h(.) entre nós sucessores seja menor do que o custo $c(n_i,n_j)$ entre os nós.

O código em Prolog para implementar o algoritmo A* segue abaixo.

```
pz :- astar(1).  % configuracao 1

puzzle(N) :- astar(N).

astar(I) :- ft(I,H,Hr), resolve(H,s,Hr),close(H).
ft(I,H,Hr) :- apaga, escolhe_heur(Hr), int_text(Hr,SH),
        int_text(I,SI), concat([sol8Pc,SI,$h$,SH,$.doc$],NA),
        create(H,NA), ctr_set(1,1),
        recordz(hnl,H,_), le_confxy(I,Rxy),
        recordz(noh,noh(Rxy,s,0),_),
        escr_cab(H,Hr), escr_conf_ini(H,s), grv_abr_z(s).

escolhe_heur(1).

le_confxy(1,[2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3]).
%Repr_pecas= [2,8,3,1,6,4,7,0,5]
le_confxy(2,[2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2]).
%Repr_pecas= [2,1,6,4,0,8,7,5,3]
le_confxy(meta,[2/2,1/3,2/3,       3/3,3/2,3/1,2/1,1/1,1/2]).
%Repr_pecas= [1,2,3,8,0,4,7,6,5]

escr_cab(H,Heu) :- write(H,$Resolvendo o 8 Puzzle através do
A* com a Heurística$), nl(H),
 heur(Heu,Hr), write(H,Hr), nl(H), nl(H),
 write(H,$Configuração Inicial$), nl(H),
 write($Resolvendo 8 Puzzle atraves do A* com a Heuristica$),
 nl,write(Hr), nl, nl, write($Configuracao Inicial$), nl.

heur(1,$      Número de Peças mal colocadas$).
heur(2,$          Distância de Manhattan$).
heur(3,$                Score          $).
```

```
escr_conf_ini(H,Noh) :- abolish(coord/3),
 recorded(noh,noh(Rxy,Noh,_),_), gr_coord(Rxy,0),
 escr_repr(H,[],Rp), write($Reprxy = $), write(Rxy), nl,
 write(H,$Reprxy = $), write(H,Rxy), nl(H),
 write($Repr_pecas = $), write(Rp), nl, nl,
 write(H,$Repr_pecas = $), write(H,Rp), nl(H), nl(H),
 write($                                Arvore de Procura$),
 nl,write(H,$                                    Arvore de
 Procura$), nl(H), escr_linhas(L0), trnsp([L0],LT),
 wraiz(H,LT,37), grv_poswr(Noh,37).

gr_coord([],_).
gr_coord([X/Y|T],I) :- NL is 4-Y, NC is X,
 assertz(coord(I,NL,NC)), II is I+1, gr_coord(T,II).

escr_repr(H,Li,Rp) :- escr(H,Li,Rp,1).
escr(_,L,L,4).
escr(H,Li,Rp,I) :- escrc(H,I,1,Li,L), II is I+1, nl(H),
                   escr(H,L,Rp,II).

escrc(_,I,4,L,L).
escrc(H,I,J,Li,L) :- coord(E,I,J), inclui_fim(E,Li,NL),
 write(H,E), write(H,$  $), JJ is J+1, escrc(H,I,JJ,NL,L).

escr_linhas(L) :- escrc(1,1,[],L1), escrc(2,1,[],L2),
                  escrc(3,1,[],L3), L=[L1,L2,L3].

escrc(I,4,L,L).
escrc(I,J,Li,L) :- coord(E,I,J), inclui_fim(E,Li,NL),
                   JJ is J+1, escrc(I,JJ,NL,L).
wraiz(_,[],_).
wraiz(Hn,[H|T],N) :- tab(N), tab(Hn,N), wrz(Hn,H), nl,
                     nl(Hn), wraiz(Hn,T,N).

wrz(_,[]).
wrz(Hn,[H|T]) :- wr2(Hn,H), wrz(Hn,T).

wr2(_,[]).
wr2(Hn,[H|T]) :- write(Hn,H), write(Hn,$ $), write(H),
                 write($ $), wr2(Hn,T).

grv_poswr(Noh,Pos) :- recordz(pos,pos(Noh,Pos),_).

grv_abr_z(Nome) :- recorded(labr,L,R), !,
                    inclui_novo(Nome,L,R).
grv_abr_z(Nome) :- recorda(labr,[Nome],_).
inclui_novo(Nome,L,R) :- not membro(Nome,L), !,
```

```
                        replace(R,[Nome|L]).
inclui_novo(_,_,_).
resolve(H,Noc,Hr) :-  df(H,Noc,Hr), resolve1(H,Noc,Hr).
df(H,Noc,Hr) :- le_confxy(meta,Rm),
  rzpv(Rm),  %grv primeira vez
  fh(Hr,Noc,t,FH), prof(Noc,Pr), dfav(FH,Pr,Fav,Hr),
  acha_pai(NohPai,Mov,Noc), ctr_is(1,I), escrpasso(Fav,H,I),
  grv_fav_s(s,Fav), grv_raiz(Noc,NohPai,Mov,Fav,37,Pr).

posraiz(s,37) :- !.
posraiz(Noh,P) :- recorded(pos,pos(Noh,P),_).

rzpv(Rm) :- recorded(noh,noh(Rm,t,_),_), !.
rzpv(Rm) :- recordz(noh,noh(Rm,t,_),_).

fh(1,Noc,Noh,FH) :- recorded(noh,noh(Rc,Noc,_),_),
    recorded(noh,noh(Rm,Noh,_),_), npmc(Rc,Rm,FH).

fh(2,Noc,Noh,FH) :- recorded(noh,noh(Rc,Noc,_),_),
            recorded(noh,noh(Rm,Noh,_),_), dm(Rc,Rm,FH).

fh(3,Noc,Noh,FH) :- recorded(noh,noh(Rc,Noc,_),_),
  recorded(noh,noh(Rm,Noh,_),_), dm(Rc,Rm,D),
  prof(Noc,Pr), mrxype(Rc,L), score(L,S), FH is Pr+D+3*S.

%% heurística especial h(n)=dm(n)+3*S(n) f(n)=prof(n)+h(n)

score(L,S) :- ml(L,LL), excl0t(LL,NL), f_anel(L,NL,ML),
    cs(ML,ML,LV), somasimples(LV,V), cc(L,I), S is V+I.

excl0t(LL,NL) :- membro(0,LL), !, excluit(0,LL,NL).
excl0t(L,L).

f_anel([0,X|_],NL,ML) :- !, inclui_fim(X,NL,ML).
f_anel(_,L,L).

ml([A,B,C|T],[A,B,C|R]) :- ml1(A,T,R).

ml1(A,T,R) :- enesimo(T,3,D), enesimo(T,6,E), enesimo(T,5,F),
  enesimo(T,4,G), enesimo(T,1,H), R=[D,E,F,G,H,A].

cc(L,1) :- [! enesimo(L,5,N) !], N>0, !.
cc(L,0).

cs([X],[X],[0]).
cs([A,B|R],[_|T],[V|S]) :- calc(A,B,V), cs(T,T,S).
```

```
calc(8,1,0) :- !.
calc(A,B,0) :- B is A+1, !.
calc(_,_,2).

mrxype(Rxy,Rp) :- abolish(coord/3), gr(Rxy,0), escr([],Rp).

gr([],_).
gr([X/Y|T],I) :- NL is 4-Y, NC is X, assertz(coord(I,NL,NC)),
                 II is I+1, gr(T,II).

escr(Li,Rp) :- escr(Li,Rp,1).
escr(L,L,4).
escr(Li,Rp,I) :- escrc(I,1,Li,L), II is I+1,
                 escr(L,Rp,II).

npmc(N) :- le_confxy(1,[_|T1]), le_confxy(meta,[_|T2]),
          npmcsh(T1,T2,N).

npmc([_|T1],[_|T2],N) :- npmcsh(T1,T2,N).
npmcsh([],[],0) :- !.
npmcsh([H|T1],[H|T2],N) :- !, npmcsh(T1,T2,N).
npmcsh([_|T1],[_|T2],N) :- npmcsh(T1,T2,M), N is M+1.

dm([_|C1],[_|C2],D) :- dist(C1,C2,D,0). %dist. Manhattan
                                        %entre configurações
dist([],[],D,D).
dist([X1/Y1|T1],[X2/Y2|T2],D,S) :-
    NS is abs(X1-X2) + abs(Y1-Y2) + S, dist(T1,T2,D,NS).

prof(Noh,Pr) :- recorded(noh,noh(_,Noh,Pr),_).

acha_pai(NPai,Mov,N) :- recorded(pai,pai(NPai,Mov,N),_), !.
acha_pai(_,Mov,N).

grv_fav_s(N,F) :- not recorded(fav,fav(s,_),_), !,
                  grv_fav(N,F).
grv_fav_s(_,_).

grv_raiz(s,NohPai,Mov,Fav,Nt,Pr) :- !, pri(Pr,Pp),
       recordb(abr,ch(Fav,Pp), estr(s,NohPai,Mov,Fav,Nt)).
grv_raiz(_,_,_,_,_,_).

pri(0,1) :- !.
pri(Pr,Pp) :- Pp is 1/Pr.

resolve1(H,Noc,Hr) :- testa_solucao(Noc), !,
     recorded(ultmov,L,_), compr(L,N), write(H,N), write(N),
```

```
     write(H,$ Movimentos executados: $),
     write($ Movimentos executados: $),
 inverte(L,LI), write(H,LI), nl(H), nl(H), write(LI), nl, nl,
 exibe_abr(H), nl, exibe_fch(H), nl, verifr8(H,Hr), nl,
 write(H,$Obs:acima de cada noh: nome(Nome do noh
        Pai,avaliacao do noh)$), nl(H),
 write($Obs.: acima de cada noh: Nome do noh Pai,nome
       avaliacao do noh$),  nl(H), nl,
 write($*************************** FIM
       ***************************$), nl, !,
 write(H,$*************************** FIM
        ***************************$), nl(H).

resolve1(H,Noc,Hr) :- rrmm(H,Noc,MNoh,Hr),
                      resolve(H,MNoh,Hr).
resolve1(H,_,_) :- recorded(ultmov,L,_), write(H,$Movimentos
 executados: $),
 write($Movimentos executados: $), inverte(L,LI),
 write(H,LI), nl(H), nl(H),
 write(LI), nl, nl, write(H,$Solução não encontrada após $),
 compr(L,N), write(H,N),
 write(H,$ passos$), nl(H), write(H,$ FIM, SEM SOLUCAO $),
 nl(H), write($Solucao nao encontrada apos $), write(N),
 write($ passos$), nl, write($ FIM, SEM SOLUCAO $), nl.

rrmm(H,Noc,MNoh,Hr) :- ctr_inc(1,M), n_inte(NI), M<NI, !,
 retrieveb(abr,ch(Fv,P),estr(Noc,NPai,Mv,Fv,Nt)),
 removeb(abr,ch(Fv,P),estr(Noc,NPai,Mv,Fv,Nt)),
 retira_abr_z(Noc), grv_fch_z(Noc), move(Noc,LConf,Hr),
 prt(H,LConf,Nt,MNoh).

prt(H,LConf,Nt,MNoh) :- det_nomes(LConf,LNomes,LMov),
 grv_labr(LNomes), escr_lconfig(LNomes,L), trnsp(L,LT),
 lnt(LMov,Ldnt), somav(Ldnt,Nt,LN), grnt(LNomes,LN),
 vtab(LN,Nt,E), somav(E,2,NE), favnoh(LNomes,LFav),
 espfav(E,F), writab(H,LFav,F,LNomes), nl, nl(H),
 wrl_listas(H,NE,LT),
 retrieveb(abr,ch(FF,PP),estr(MNoh,PMNoh,Mov,FF,Nn)),
 grv(Mov), grvt(Mov).

testa_solucao(Noc) :- recorded(noh,noh(R,Noc,_),_),
                      recorded(noh,noh(M,t,_),_),
                      dm(R,M,N), N==0.

exibe_abr(H) :- recorded(labr,L,_), inverte(L,M),
  write($Lista aberto: $), write(M), nl, write(H,$Lista
  aberto: $), write(H,M), nl(H).
```

```
exibe_fch(H) :- recorded(lfch,L,_), inverte(L,M),
  write($Lista fechado: $), write(M), nl, write(H,$Lista
  fechado: $), write(H,M), nl(H).

verifr8(H,Hr) :- Hr<3, !, recorded(lfch,L,_), det_fav(L,M),
  inverte(M,[A|S]), write($Resultado 8 Ok: $), write([A|S]),
  nl, write(H,$Resultado 8 Ok: $), write(H,[A|S]), nl(H),
  int_text(Hr,SH), concat(lf,SH,Ch), detrxy(S,LL), apg(Ch),
  recordz(Ch,LL,_).
verifr8(_,_).

det_fav([],[]).
det_fav([H|T],[H/F|G]) :- recorded(fav,fav(H,F),_),
                          det_fav(T,G).
detrxy([],[]).
detrxy([H/_|T],[R|S]) :- recorded(noh,noh(R,H,_),_),
                         detrxy(T,S).

%Se h2(n)>h1(n) entao todo noh expandido por A2 serah expandido por A1
tr6(R) :- concat(lf,$1$,Ch1), concat(lf,$2$,Ch2),
  recorded(Ch1,L1,_), recorded(Ch2,L2,_), contem(L1,L2,R).

contem(_,[],sim).
contem(L1,[H|T],R) :- membro(H,L1), contem(L1,T,R).
contem(_,_,nao).

n_inte(32). % numero de iteracoes

retira_abr_z(Noc) :- recorded(labr,L,R), excluit(Noc,L,NL),
                     replace(R,NL).

grv_fch_z(Nome) :- recorded(lfch,L,R), !,
                   inclui_novo(Nome,L,R).
grv_fch_z(Nome) :- recorda(lfch,[Nome],_).

%Nome_noh, Lista de Novas conf Rxy
move(Noc,LNC,Hr) :- det_mov(Noc,Lmov),
                    move_av(Lmov,Noc,LNC,Hr).

det_mov(Noc,L) :- rem_umov(Lp), detbl(Noc,Xb,Yb),
                  dlmov(Xb,Yb,Lp,L).

rem_umov(Lp) :- recorded(umov,M,_), !,
                antimov(M,Am), excluit(Am,[e,b,c,d],Lp).
rem_umov([e,b,c,d]).

detbl(Noh,Xb,Yb) :- recorded(noh,noh(Rxy,Noh,_),_),
                    dbl(Rxy,Xb,Yb).
```

```
dbl([Xb/Yb|_],Xb,Yb).


dlmov(_,_,[],[]).
dlmov(Xb,Yb,[H|T],[H|S]) :- cond(Xb,Yb,H), dlmov(Xb,Yb,T,S).
dlmov(Xb,Yb,[_|T],S) :- cond(Xb,Yb,H), dlmov(Xb,Yb,T,S).

cond(Xb,_,e) :- Xb>1, !.
cond(Xb,_,d) :- Xb<3, !.
cond(_,Yb,c) :- Yb<3, !.
cond(_,Yb,b) :- Yb>1.

antimov(c,b) :- !.
antimov(b,c) :- !.
antimov(d,e) :- !.
antimov(e,d).

move_av([],_,[],_).
move_av([H|T],Noc,[Hc|Tc],Hr) :- move_nct(H,Noc,Hc,Nome),
  avalia(Nome,t,Hm,Hr), prof(Nome,Pr), dfav(Hm,Pr,Fav,Hr),
  grv_fav(Nome,Fav), grv_abr(Nome,Noc,H,Fav,_,Pr),
  move_av(T,Noc,Tc,Hr).

dfav(Hm,Pr,Hm,3) :- !.
dfav(Hm,Pr,Fav,_) :- Fav is Hm + Pr.

move_nct(H,Noc,Hc,Nome) :- move_nc(H,Noc,Hc), prox_n(Nome),
 fixa_pai(Noc,H,Nome), assoc(Hc,Nome).

move_nc(Mov,Noc,NConf) :- recorded(noh,noh(Rc,Noc,_),_),
 detbranco(Mov,Rc,Xb,Yb,NX,NY),
 movimenta(Rc,NConf,Xb,Yb,NX,NY).

detbranco(e,[Xb/Yb|T],Xb,Yb,NX,Yb) :- Xb>1,!,NX is Xb-1.
detbranco(d,[Xb/Yb|T],Xb,Yb,NX,Yb) :- Xb<3,!,NX is Xb+1.
detbranco(b,[Xb/Yb|T],Xb,Yb,Xb,NY) :- Yb>1,!,NY is Yb-1.
detbranco(c,[Xb/Yb|T],Xb,Yb,Xb,NY) :- Yb<3,!,NY is Yb+1.

movimenta([],[],_,_,_,_) :- !.
movimenta([Xb/Yb|T],[NX/NY|NT],Xb,Yb,NX,NY) :-
     !,movimenta(T,NT,Xb,Yb,NX,NY).
movimenta([NX/NY|T],[Xb/Yb|NT],Xb,Yb,NX,NY) :-
     !,movimenta(T,NT,Xb,Yb,NX,NY).
movimenta([H|T],[H|NT],Xb,Yb,NX,NY) :-
     movimenta(T,NT,Xb,Yb,NX,NY).
```

```
prox_n(Nome) :- recorded(ultnome,S,R), !, string_length(S,N),
 N1 is N-1,substring(S,1,N1,S1), int_text(I1,S1), I2 is I1+1,
 int_text(I2,S2), concat(n,S2,Nome), replace(R,Nome).

prox_n(n1) :- recordz(ultnome,n1,_).

fixa_pai(NPai,Mov,N) :- recorda(pai,pai(NPai,Mov,N),_).

assoc(Rxy,N) :- acha_pai(P,M,N), !, prof(P,Pr), Pn is Pr+1,
                recorda(noh,noh(Rxy,N,Pn),_).

avalia(X,_,10,_) :- var(X), !.
avalia(Hc,Rm,Hv,Hr) :- fh(Hr,Hc,Rm,Hv).

grv_fav(Nome,Fav) :- recordz(fav,fav(Nome,Fav),_).

grv_abr(Nome,PNome,Mov,Fav,Nt,Pr) - rv_abr_z(Nome),
     pri(Pr,Pp),
     recordb(abr, ch(Fav,Pp), estr(Nome,PNome,Mov,Fav,Nt)).

det_nomes([],[],[]).
det_nomes([Rxy|T],[N|R],[M|S]) :-
     recorded(noh,noh(Rxy,N,_),_), acha_pai(P,M,N),
     det_nomes(T,R,S).
grv_labr([]).
grv_labr([H|T]) :- grv_abr_z(H), grv_labr(T).

escr_lconfig([],[]).
escr_lconfig([Noh|T],[H|R]) :- escr_config(Noh,H),
                               escr_lconfig(T,R).

escr_config(Noh,L) :- abolish(coord/3),
  recorded(noh,noh(Rxy,Noh,_),_),
  gr_coord(Rxy,0), escr_linhas(L).

trnsp(A,AT) :- klis(A,[],AT).
klis([],L,L).
klis([H|T],Li,L) :- klist(H,Li,M), klis(T,M,L).
klist([],[],[]).
klist(H,[],L) :- flat(H,L).
klist([H|T],[H1|T1],[H2|T2]) :- inclui_fim(H,H1,H2),
                                klist(T,T1,T2).

flat([],[]).
flat([H|T],[[H]|S]) :- flat(T,S).

lnt([e,b,c,d],[-3,4,3,6]) :- !.
```

```
lnt([e,b,c],[-7,0,0]) :- !.
lnt([e,b,d],[-7,0,0]) :- !.
lnt([e,c,d],[-7,0,0]) :- !.
lnt([b,c,d],[-7,1,1]) :- !.
lnt([e,b],[-4,1]) :- !.
lnt([e,c],[-4,1]) :- !.
lnt([e,d],[-4,1]) :- !.
lnt([b,c],[-4,1]) :- !.
lnt([b,d],[-4,1]) :- !.
lnt([c,d],[-4,1]) :- !.
lnt([e],[0]) :- !.
lnt([b],[0]) :- !.
lnt([c],[0]) :- !.
lnt([d],[0]).

grnt([],[]).
grnt([Nome|T],[Nt|S]) :-
  removeb(abr,ch(Fav,Pr),estr(Nome,PNome,Mov,Fav,_)),
  recordb(abr,ch(Fav,Pr), estr(Nome,PNome,Mov,Fav,Nt)),
  grv_poswr(Nome,Nt), grnt(T,S).

vtab([A,B,C,D],Nt,[NA,NB,NC,ND]) :- NA is A-Nt+20,
  NB is B-Nt+1, NC is C-Nt+1, ND is D-Nt+3, !.
vtab([A,B,C],Nt,[A,NB,NC]) :- NB is B-Nt+1, NC is C-Nt+1, !.
vtab([A,B],Nt,[A,NB]) :- !, NB is B-Nt+1.
vtab([A],_,[A]).

favnoh([],[]).
favnoh([H|T],[U|V]) :- recorded(fav,fav(H,U),_), favnoh(T,V).

espfav([A,B,C,D],[NA,NB,NC,ND]) :-
      NA is A, NB is B-2, NC is C-2, ND is D-2.
espfav([A,B,C],[NA,NB,NC]) :- NA is A, NB is B, NC is C.
espfav([A,B],[NA,NB]) :- NA is A, NB is B.
espfav([A],[NA]) :- NA is A.

writab(_,[],[],[]).
writab(Hn,[H|T],[U|V],[N|S]) :- acha_pai(P,_,N), sts(U,P,X),
 tab(X), write(N), write($($), write(P), write($,$),
 write(H), write($)$), tab(Hn,X), write(Hn,N), write(Hn,$($),
 write(Hn,P), write(Hn,$,$), write(Hn,H), write(Hn,$)$),
 writab(Hn,T,V,S).

sts(U,P,X) :- string_length(P,N), ss(N,U,X).
ss(1,U,X) :- !, X is U+1.
ss(2,U,U) :- !.
ss(3,U,X) :- X is U-1.
```

```
wrl_listas(_,_,[]).
wrl_listas(Hn,Ld,[H|T]) :- wr1(Hn,H,Ld), nl, nl(Hn),
                           wrl_listas(Hn,Ld,T).

wr1(_,[],[]).
wr1(Hn,[H|T],[D|S]) :- tab(D), tab(Hn,D), wr2(Hn,H),
                       wr1(Hn,T,S).

grv(Mov) :- recorded(umov,_,S), replace(S,Mov).
grv(Mov) :- recordz(umov,Mov,_).

grvt(Mov) :- recorded(ultmov,L,R), !, replace(R,[Mov|L]).
grvt(Mov) :- recordz(ultmov,[Mov],_).

escrpasso(N,H,I) :- N>0, !, nl(H), nl.
escrpasso(_,_,_).

apaga :- apg(ultmov), apg(ultnome), apg(noh), apg(prf),
         apg(pos), apg(umov), apg(fav), rmb(abr), apg(labr),
         apg(lfch), abolish(coord/3).
apg(X) :- eraseall(X).
rmb(X) :- removeallb(X).

append([],T,T).
append([H|T],L,[H|TL]) :- append(T,L,TL).

compr([],0).
compr([_|T],N) :- compr(T,N0), N is N0 + 1.

enesimo([H|_],1,H).
enesimo([_|T],N,X) :- NN is N-1, enesimo(T,NN,X).
enesimo([],_,_) :- !, write($ a lista tem menos de N elementos$).

excluit(X,[X|L],M) :- excluit(X,L,M).
excluit(X,[H|T],[H|R]) :- excluit(X,T,R).
excluit(_,L,L).

inclui(X,L,[X|L]).

inclui_fim(X,[],[X]).
inclui_fim(X,[H|T],[H|R]) :- inclui_fim(X,T,R).

inverte([],[]).
inverte([H|T],L) :- inverte(T,Ti),append(Ti,[H],L).

membro(H,[H|_]).
membro(X,[_|T]) :- membro(X,T).
```

```
somasimples([],0).
somasimples([H|T],S) :- somasimples(T,R),S is H+R.

somav([],_,[]).
somav([H|T],Nt,[NH|R]) :- NH is H+Nt, somav(T,Nt,R).
```

5. Conclusões

Este trabalho, ainda preliminar, representa um esforço de difundir as facilidades que um ambiente de Programação em Lógica pode oferecer aos usuários.

6. Bibliografia

Arity Corporation (1999). The Arity / Prolog Language Reference Manual.

Bratko, I. (1990). Prolog Programming for Artificial Intelligence, 2ª edição, Addison Wesley.

Casanova, M.A. & Giorno, F.A.C. & Furtado, A.L. (1987). Programação em Lógica e a Linguagem Prolog, Edgard Blücher, LTDA.

Clocksin, W.F. & Mellish, C.S. (1984). Programming in Prolog, 2ª edição , New York: Springer-Verlag.

Kowalski, R. (1979) Logic for Problem Solving, New York: North Holland.

Luger, G.F. & Stubblefield, W.A. (1989). Artificial Intelligence and the Design of Expert Systems, Benjamin Cummings.

Nilsson, N.J. (1982). Principles of Artificial Intelligence, Springer Verlag.

Rowe, N.C. (1988). Artificial Intelligence through Prolog, Prentice Hall.

7. Anexo

```
% São definidos alguns predicados básicos para a operação com
% listas que foram chamados nos programas acima.
% * Predicados de finalidade geral

apg(X) :- eraseall(X).
apaga :- apg(eh), apg(pos),apg(spp),apg(pli),apg(str).

append([],L,L).
append([H|T],L,[H|M]) :- append(T,L,M).

compr([],0).
compr([_|T],N) :- compr(T,N0), N is N0 + 1.

detpesc([],[],0).
detpesc([Hl|Tl],[Hv|Tv],C) :- detpesc(Tl,Tv,P), C is Hl*Hv+P.
```

```
enesimo([H|_],1,H).
enesimo([_|T],N,X) :- NN is N-1, enesimo(T,NN,X).
enesimo([],_,_) :- !,write($lista tem menos de N elementos$),
                    nl, fail.
exclui(X,[X|L],L).
exclui(X,[H|T],[H|R]) :- exclui(X,T,R).
exclui(_,L,L).

exclui_n(N,L,NL) :- enesimo(L,N,E), exclui(E,L,NL).

exclpos([H|T],1,T).
exclpos([H|T],N,[H|L]) :- N1 is N-1, exclpos(T,N1,L).

flat([],[]).
flat([H|T],[[H]|S]) :- flat(T,S).

inclui(X,L,[X|L]).

inclui_fim(X,[],[X]).
inclui_fim(X,[H|T],[H|L]) :- inclui_fim(X,T,L).

inverte([],[]).
inverte([H|T],L) :- inverte(T,Ti),append(Ti,[H],L).

max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).

max([A],A).
max([A|T],A) :- max(T,X), A >= X, !.
max([_|T],X) :- max(T,X).

membro(X,[X|_]).
membro(X,[_|T]) :- membro(X,T).

min(X,Y,X) :- X =< Y, !.
min(_,Y,Y).

min([A],A).
min([A|T],A) :- min(T,X), A =< X, !.
min([_|T],X) :- min(T,X).
rmb(X) :- removeallb(X).
```