O MUNDO WUMPUS

GRUPO DANDY

-RESUMO DO MUNDO WUMPUS.

-ALGORÍTMO E EXPLICAÇÕES.

RESUMO

O Wumpus é um exercício para a criação de inteligência artificial; nele, programadores devem criar soluções inteligentes para que o agente desvie de obstáculos e alcance seu objetivo. Os tipos de obstáculos e a sensação que cada um provoca são:

- Wumpus: provoca fedor nas casas adjacentes.
- Morcego: provoca ruído nas casas adjacentes
- Buraco: provoca sensação de brisa nas casas adjacentes.

Pepitas de ouro brilham, mas o agente só percebe isso ao estar na mesma célula que o objeto.

O objetivo do agente é evitar o encontro com os três obstáculos, pegar a pepita de ouro, matar o wumpus e sair da caverna.

ALGORITMO E EXPLICAÇÕES

1. VARIÁVEIS

O algoritmo começa usando dynamic, que indica que a definição dos predicados pode mudar durante a execução do programa.

```
:-dynamic([ultacao/1,      %Fatos Dinâmicos
        ouro/1,
        aposicao/1,
        angulo/1,
        casas_visitadas/1,
        visitadas/1,
        cont_acoes/1,
        flechas/1,
        wumpus/1]).
```

O predicado “ouro” é uma variável que conta quantas pepitas de ouro o agente tem. O predicado “aposicao” e “angulo” referem-se ,respectivamente, às coordenadas X,Y em que o agente está e a sua angulação, contando 0 graus como a direita e o primeiro ângulo do agente; “casas_visitadas”, refere-se às casas por quais o agente já passou desde suas trajetória inicial. A variável “cont_acoes” conta a quantidade de ações que o agente já tomou, tentando limita-las à um número máximo de ações possíveis até o Wumpus morrer. A variável “flechas” é em relação ao número de flechas que o agente possui e, se o agente conseguir matar o Wumpus, este irá de vivo à morto, portanto foi necessário também uma variável para indicar sua situação.

2. INICIALIZAÇÃO

A função “init_agent” é usada para a inicialização do programa. Esta função chama outras funções usando retractall e assert.

```
init_agent :-
   retractall(ultacao(_)),
   retractall(ouro(_)),
   retractall(aposicao(_)),
   retractall(casas_visitadas(_)),
   retractall(visitadas(_)),
   retractall(angulo(_)),
   retractall(cont_acoes(_)),
   retractall(flechas(_)),
   retractall(wumpus(_)),
   assert(aposicao([[1,1]])),          %casa inicial
   assert(ouro(0)),                    %Quantidade inicial de ouro
   assert(casas_visitadas([[1,1]])),   %casas visitadas
   assert(angulo(0)),                  %Orientacao inicial do agente
   assert(ultacao(acao)),
   assert(visitadas([[1,1]])),         %casas visitadas
   assert(cont_acoes(0)),              % conta a quantidade de acoes
   assert(flechas(1)),                 %Quantidade de flechas
   assert(wumpus(vivo)).
```

O uso do retractall é usado para remover fatos do database do Prolog enquanto o programa ainda está sendo executado e os assert adiciona um novo fato ao database, o termo é então declarado como o ultimo fato ou clausula com o mesmo tipo de predicado-variável.

3. RUN_AGENT

O run_agent inicializa as funções, ele recebe as percepções e o meio ao seu redor e as interliga com ações, portanto, para o programa ser compilado com sucesso, as funções nunca poderão ser falsas.

```
run_agent(Percepcao, Acao) :-
    acao(Percepcao, Acao),
    %Fatos%
    aposicao(Posicao),    %Posicao inicial
    ouro(Quantidade),     %Quantidade de ouro
    cont_acoes(Qacoes),   %quantidade de acoes
    wumpus(Estado),       %Vivo/Morto
    %Predicados
    cria_casasvi,         %Cria casas visitadas
    visitadas(Visitadas),
    verifica(Acao),
    atualiza_cont_acoes,  %Quantidade de acoes do a
    %   casa_frente(Posicao,Sentido,Frente),
    %Impressao
    write('Quantidade de acoes: '),
    writeln(Qacoes),
    write('Quantidade de ouro: '),
    writeln(Quantidade),
    write('Minha posicao: '),
    writeln(Posicao),
    write('Casas visitadas'),
    writeln(Visitadas),
    write('Estado do Wumpus :'),
    writeln(Estado).
```

No run_agent colocamos “write” para informar, a cada ação do e momento do programa, a quantidade de ações, ouro, posições e percepções. Além disso, informamos as casas visitadas e o estado do Wumpus e do agente.

4. QUANTO A AÇÕES:

O programa possui múltiplas cláusulas em relação as ações do agente. Cada cláusula será explicada abaixo:

```
acao(_,X) :-
    random_between(1,10,Y),
    andar(Y,X).

andar(Y,X) :-
    Y =< 8,
    X = goforward;
    Y =:= 9,
```

```
    X = turnright;
    Y =:= 10,
    X = turnleft
```
Essa parte do código pega o valor aleatório que foi gerado na cláusula anterior e dependendo do número ele realiza uma das ações apresentadas.

```
atualiza_cont_acoes :-
    cont_acoes(Qacoes),
    Qacoes1 is Qacoes+1,
    retractall(cont_acoes(_)),
    assert(cont_acoes(Qacoes1)).
```

Essa cláusula atualiza a quantidade de ações de acordo com o número de ações já feitas pelo agente.

```
acao([_,_,yes|_],grab):-
    ouro(X),
    retractall(ouro(_)),
    Xf is X+1,
    assert(ouro(Xf)),
    write('Estou rico!'),
    nl.
```
Essa cláusula relaciona a percepção de brilho com a ação de pegar. Com isso é adicionado +1 ao número de pepitas de ouro, isso é feito pelo uso de retractall para apagar a antiga quantidade do database e, com assert, adicionar uma nova quantidade. Ao rodar o programa será transmitida a mensagem “Estou rico!”.

```
acao([_,yes|_],X) :- %Vento
    ultacao(Y),
    Y = turnleft,
    X = goforward.

acao([_,yes|_],turnleft). %Vento
```

As cláusulas são em relação à percepção de brisa(ou vento) sentida pelo agente. Na primeira, se a ultima ação do agente foi virar a esquerda, então o agente deve ir em frente; na última cláusula temos um caso geral de virar à esquerda caso essa percepção aconteça.

```
acao([_,_,_,_,_,yes],X) :- %Ruido
    ultacao(Y),
    Y = turnright,
    X = goforward.

acao([_,_,_,_,_,yes],turnright). %Ruido
```
Ambas as cláusulas acima são em relação à percepção de ruído. Na primeira se a última ação for virar à direita, então o agente continua em frente; na segunda, que é a geral, o agente vira a direita ao escutar ruído.

```
acao([_,_,_,yes,_,_],turnleft).  %Trombada
```

A cláusula acima refere-se ao caso em que o agente bata contra a parede da caverna. Se isso acontecer o agente vira à esquerda e tenta continuar seu caminho.

```
flechas(X),
    X>0,
    X1 is X-1,
    retractall(agente_flecha(_)),
    assert(agente_flecha(X1)).
```

Essa cláusula trata do número de flechas. Primeiro ele checa o número de flechas e depois decrementa -1 desse número, no fim ele atualiza a quantidade.

```
acao([yes,yes,_,_,_,_],shoot):-  %fedor
    wumpus(vivo),
    flechas(F),
    F>0,
    atira.

acao([_,_,_,_,yes],_):-
    retractall(wumpus(_)),
    assert(wumpus(morto)).
```

A parte acima é caso o agente sentir fedor e possuir flechas, se ambos forem verdadeiros, o agente atira. Se o agente atirar e ouvir o grito do Wumpus, então o estado do Wumpus é mudado de vivo para morto com o uso de retractall e assert.

```
acao([yes,_,_,_,_,_],turnright). %fedor
```

Caso o grito não aconteça, o agente virara à direita e continuará seu percurso.

5. QUANTO À POSIÇÃO

Foi necessário uma lista com as casas adjacentes à casa que o agente se encontra a cada momento.

```
adjacentes(X,Y,Z):-
    aposicao[[X,Y]],
    X1 is X + 1,
    X2 is X - 1,
    Y1 is Y + 1,
    Y2 is Y - 1,
    Z = [[X, Y1],[X, Y2],[X1, Y],[X2, Y]],
    write('Casas adjacentes'),
    writeln(Z).
```

Convencionamos X sendo colunas e Y sendo linhas.

As funções chamadas pelo grupo de “salvapos” salva a posição e atualiza as coordenadas.

```
salvapos :-
    angulo(A),
    A =:= 0,
    aposicao([[X,Y]]),
    Xf is X+1,
    retractall(aposicao(_)),
    assert(aposicao([[Xf,Y]])).
```

```
salvapos :-
    angulo(A),
    A =:= 180,
    aposicao([[X,Y]]),
    Xf is X-1,
    retractall(aposicao(_)),
    assert(aposicao([[Xf,Y]])).

salvapos :-
    angulo(A),
    A =:= 90,
    aposicao([[X,Y]]),
    Yf is Y+1,
    retractall(aposicao(_)),
    assert(aposicao([[X,Yf]])).

salvapos :-
    angulo(A),
    A =:= 270,
    aposicao([[X,Y]]),
    Yf is Y-1,
    retractall(aposicao(_)),
    assert(aposicao([[X,Yf]])).
```

A função verifica o ângulo e a posição do agente e atualiza a posição se o agente continuar em frente com a mesma angulação.

```
salvaangulo(turnright) :-
    angulo(A),
    A1 is A - 90,
    convencao(A1, A2),
    retractall(angulo(_)),
    assert(angulo(A2)).

salvaangulo(turnleft) :-
    angulo(A),
    A1 is A + 90,
    convencao(A1, A2),
    retractall(angulo(_)),
    assert(angulo(A2)).
```

As funções atualizam o ângulo se o agente se virar para direita ou esquerda, o uso de retractall e assert são, portanto, necessários outra vez.

```
convencao(-90, 270).
convencao(360, 0).
convencao(A, A).
```

Essas listas igualam esses valores possíveis de angulação para facilitar o código.

```
addvisitadas(X) :-
    visitadas(A),
    W = [X],
    union(W,A,C),
    retractall(visitadas(_)),
    assert(visitadas(C)).
```

Essa cláusula trata das casas visitadas pelo agente. A cada mudança de coordenada do agente é necessária a atualização dessa lista.

```
verifica(Ac) :-
    Ac == goforward,
    salvapos,
    aposicao(X),
    addvisitadas(X).

verifica(Ac) :-
    Ac == turnleft,
    salvaangulo(Ac).

verifica(Ac) :-
    Ac == turnright,
    salvaangulo(Ac).

verifica(_).
```

As cláusulas acima verificam as ultimas ações feitas pelo agente; se a ação foi um goforward então existe mudança de coordenadas, mas se a ação for um turnleft/turnright então é preciso atualizar a angulação do agente.

```
cria_casasvi:-
    aposicao(Posicao),
    casas_visitadas(Cvi),
    append(Posicao, Cvi, Lista1),
    list_to_set(Lista1, Lista),
    retractall(casas_visitadas(_)),
    assert(casas_visitadas(Lista)),
    write('Casas seguras: '),
    writeln(Lista).
```

Esse predicado cria uma lista com as casas visitadas pelo agente. Essas casas são portanto seguras, pois não possuem nenhum dos obstáculos, assim, as coordenadas das casas são usadas para criar uma lista de casas seguras.

```
%acao(_, Acao):-
%   casas_visitadas(CasasVisitadas),
%    minhacasa(Posicao),
%   angulo(Sentido),
%   casa_frente(Posicao, 0, Frente), %qual a casa da frente com sentido 0
%   member(Frente, casas_visitadas),
%   acao(Sentido, 0, Acao).

%acao(_, Acao):-
%   casas_visitadas(CasasVisitadas),
%    minhacasa(Posicao),
%   angulo(Sentido),
%   casa_frente(Posicao, 90, Frente), %qual a casa da frente com sentido 90
%   member(Frente, casas_visitadas),
%   acao(Sentido, 90, Acao).
```

```
%acao(_, Acao):-
%   casas_visitadas(CasasVisitadas),
%    minhacasa(Posicao),
%   angulo(Sentido),
%   casa_frente(Posicao, 180, Frente), %qual a casa da frente com sentido 180
%   member(Frente, casas_visitadas),
%   acao(Sentido, 180, Acao).

%acao(_, Acao):-
%   casas_visitadas(CasasVisitadas),
%    minhacasa(Posicao),
%   angulo(Sentido),
%   casa_frente(Posicao, 270, Frente), %qual a casa da frente com sentido 270
%   member(Frente, casas_visitadas),
%   acao(Sentido, 270, Acao).
```

Essas cláusulas de ações são necessárias para o agente voltar pelas casas visitadas, dependendo de seu ângulo. O predicado member checa se a casa da frente faz parte de casas visitadas, se sim o agente continua seu caminho por ela.

```
salvaseguras :-
    aposicao(X,Y),  %Vejo a posição onde ele se encontra
    Sx is X + 1,  %Somo 1 a X para salvar a casa do lado direito
    Dx is X - 1,  %Diminuo 1 a X para salvar a casa do lado esquerdo
    Sy is Y + 1,  %Somo 1 a Y para salvar a casa de cima
    Dy is Y - 1,  %Diminuo 1 a Y para salvar a casa de baixo
    visitadas(E),
    seguraw(W),
    union([[X,Dy]],W,P),
    union([[Dx,Y]],P,Q),
    union([[X,Sy]],Q,R),
    union([[Sx,Y]],R,S),
    intersection(S,E,I),
    subtract(S,I,T),
    retractall(seguras(_)), %limpa o que tiver em seguras
    assert(seguras(T)).
```

Essa cláusula salva as casas seguras, primeiramente ele vê a posição em que o agente se encontra, depois acha Sx, Dx, Sy, Dy que são coordenadas das casas adjacentes, após isso é achado as interseções entre seguras e visitadas e a lista de coordenadas. Por fim ele subtrai a lista das interseções de S e atribui isso a T, dando T como casas seguras.