Universidade Federal de São Carlos
Campus São Carlos

A.I.A.D.P

Agente Inteligente de Aspirador de Pó

Projeto Inteligência Artificial – Prof. Murilo Naldi

Alunos:
Daniel Lucio Masselani de Moura - 743525
João Vitor Azevedo - 743554
Lucas Heidy Tuguimoto Garcia - 743565

São Carlos, SP
2019

1. Introdução

O objetivo deste projeto é construir um programa capaz de gerar um caminho a ser percorrido por um agente aspirador de pó. Tal agente possui características como número máximo de unidades de sujeira carregadas simultaneamente, a movimentação horizontal livre, e a impossibilidade de passar por paredes presente no ambiente. Para carregar-se de mais unidades de sujeiras, deve depositar as anteriores numa lixeira, e para movimentar-se verticalmente deve usar um elevador. No fim, deve localizar-se em um *dock station.*

2. Definição dos Estados

```
[Posicao,Sacola,Sujeiras]

 Posicao: p(X,Y) (Ex.: p(3,1))
 Sacola: Número Inteiro de 0 a 2
 Sujeiras: [P1,P2,P3,...]
```

Um estado neste projeto é constituído de 3 itens:

- Uma posição (p(X,Y)) que será a posição inicial do agente;
- Um número inteiro que representa a o número de unidades de sujeira armazenadas pelo agente;
- E uma lista de posições ([p(X,Y), p(X2,Y2),...]) que guarda a posição das unidades de sujeira ainda não colhidas pelo agente.

Ex: `[p(1,1),0,[p(1,2),p(3,1),p(4,2),p(5,5),p(3,4)]]`

3. Modelagem

Os objetos do ambiente são declarados logo no início do programa:

```
elevador(4).
lixeira(p(1,3).
powerstation(p(10,1)).
parede(p(7,1)).
```

No caso de elevador, lixeira e parede, o objeto define uma posição do ambiente. Já o elevador, define toda uma coluna na qual é permitida a movimentação vertical do agente.

4. Regras

4.1 Regras importadas

Para a construção do programa foram importadas algumas regras dos slides das aulas de Inteligência Artificial do prof. Murilo Naldi:

- Regra **pertence** que checa se um elemento pertence à uma lista:

```
pertence(Elem,[Elem|_]).
pertence(Elem,[_|Cauda]) :- pertence(Elem,Cauda).
```

- A regra **retirar_elemento** que retorna uma lista sem um elemento específico.

```
retirar_elemento(Elem,[Elem|Cauda],Cauda).
retirar_elemento(Elem,[Cabeca|Cauda],[Cabeca|Resultado]) :-
     retirar_elemento(Elem,Cauda,Resultado).
```

- A regra **concatena**, utilizada nas busca em largura

```
concatena([],L,L).
concatena([Cabeca|Cauda],L2,[Cabeca|Resultado]) :-
  concatena(Cauda,L2,Resultado).
```

- As buscas em largura e profundidade que são utilizadas para achar o caminho do agente.

```
% Busca em Largura
solucao_bl(Inicial,Solucao) :-
  bl([[Inicial]],Solucao).

bl([[Estado|Caminho]|_],[Estado|Caminho]) :-
  meta(Estado).

bl([Primeiro|Outros],Solucao) :-
  estende(Primeiro,Sucessores),
  concatena(Outros,Sucessores,NovaFronteira),
  bl(NovaFronteira,Solucao).
```

```
estende(_,[]).
estende([Estado|Caminho],ListaSucessores) :-
  bagof(
    [Sucessor,Estado|Caminho],
    (s(Estado,Sucessor),not(pertence(Sucessor,[Estado|Caminho]))),
    ListaSucessores),!.

% Busca em Profundidade
solucao_bp(Inicial,Solucao) :-
    bp([],Inicial,Solucao),writeln(Solucao).
bp(Caminho,Estado,[Estado|Caminho]) :-
    meta(Estado).
bp(Caminho,Estado,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    bp([Estado|Caminho],Sucessor,Solucao).
```

4.2 Regras base

Foram criadas para serem usadas nas verificações das sucessões de estado:

- **fora_do_mapa** é utilizada para checar se uma posição é válida dentro dos limites do ambiente:

```
fora_do_mapa(p(X,Y)) :-
  X = 0;
  Y = 0;
  X = 11;
  Y = 6.
```

- **pode_passar** define se uma movimentação para dada posição é possível (a posição não pode ser uma parede, e deve ser checada por **fora_do_mapa**):

```
pode_passar(_,Pos2) :-
  not(parede(Pos2)),
  not(fora_do_mapa(Pos2)).
```

4.3 Regras de Sucessão

Foram criadas múltiplas regras de sucessão diferentes, cada uma designada a uma ação do agente:

- A primeira trata a ação de recolher uma unidade de sujeira em uma posição, incrementar a quantidade na sacola e retirar a posição da lista de posições com sujeira.

```
s([Pos, Sacola, Sujeiras], [Pos,Sacola2,Sujeiras2]):-
pertence(Pos,Sujeiras),
          retirar_elemento(Pos,Sujeiras,Sujeiras2),
          Sacola < 2,
          Sacola2 is Sacola + 1.
```

- O estado (Pos,Sacola,Sujeiras) vem antes de (Pos,Sacola2,Sujeiras2) se:
    - Essa Pos pertence à lista de sujeiras e
    - Sujeiras 2 é Sujeira sem o 'elemento' Pos e
    - A sacola não está cheia e
    - Sacola2 tem 1 lixo a mais que Sacola

- Outra regra trata a ação de esvaziar a sacola na lixeira, zerando o contador de unidades de sujeira.

```
s([Pos,Sacola,Sujeiras],[Pos,Sacola2,Sujeiras]) :-
lixeira(Pos),    Sacola > 0,
Sacola2 is 0.
```

    - Se houver uma lixeira na posição atual e houver lixo na sacola, esvazia a sacola.

- Há também uma regra para tratar apenas o movimento horizontal do agente, realizando a validação por meio de uma das regras base, e atualizando a posição do agente.

```
s([Pos,Sacola,Sujeiras],[Pos,Sacola2,Sujeiras]) :-
lixeira(Pos),    Sacola > 0,
Sacola2 is 0.
```

- A posição p(SX,Y) vem logo depois de p(X,Y) se:
    - estiver à um “passo” de distância dela
    - é possivel passar de p(X,Y) para p(SX,Y).

- Por último, existe uma regra de sucessão de estados para o movimento vertical do agente. Validando a existência de um elevador na posição e atualizando a posição do agente.

```
s([p(X,Y),Sacola,Sujeiras],[p(X,SY),Sacola,Sujeiras]) :-
elevador(X),(SY is Y + 1; SY is Y - 1),
not(fora_do_mapa(p(X,SY))).
```

- O robô pode andar no Y apenas se:
    - houver um elevador na posição atual dele
    - O o novo Y estiver à um passo do antigo
    - a nova posição não estiver fora do mapa.

5. Meta

```
meta([Pos, 0, Lixos]) :-
  powerstation(Pos),
  Lixos = [].
```

O estado objetivo é um estado em que
- o robô está na mesma posição que o dockstation(powerstation)
- A sacola do robô está com 0 sujeiras
- A lista de Sujeiras está vazia

6. Desafios

Arquivo listas.pl:

```
pertence(Elem,[Elem|_]).              %pertence se é a cabeca
pertence(Elem,[_|Cauda]) :-           %pertence se pertence à
cauda
    pertence(Elem,Cauda).

retirar_elemento(Elem,[Elem|Cauda],Cauda).
retirar_elemento(Elem,[Cabeca|Cauda],[Cabeca|Resultado]) :-
  retirar_elemento(Elem,Cauda,Resultado).

concatena([],L,L).
concatena([Cabeca|Cauda],L2,[Cabeca|Resultado]) :-
  concatena(Cauda,L2,Resultado).
```

Arquivo busca_profundidade.pl:

```
:- [listas].

solucao_bp(Inicial,Solucao) :-
    bp([],Inicial,Solucao),writeln(Solucao).
bp(Caminho,Estado,[Estado|Caminho]) :-
    meta(Estado).
bp(Caminho,Estado,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    bp([Estado|Caminho],Sucessor,Solucao).
```

Arquivo busca_profundidade.pl:

```
:- [listas].

solucao_bp(Inicial,Solucao) :-
    bp([],Inicial,Solucao),writeln(Solucao).
bp(Caminho,Estado,[Estado|Caminho]) :-
    meta(Estado).
bp(Caminho,Estado,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    bp([Estado|Caminho],Sucessor,Solucao).
```

Arquivo caso_0.pl:

```
:- [listas].

% elevadores
elevador(7).
elevador(9).

%Definindo lixeiras
lixeira(p(6,1)).

%Definindo PowerStation
powerstation(p(10,1)).


parede(p(8,2)).
parede(p(8,3)).
parede(p(8,4)).
parede(p(8,5)).
parede(p(8,6)).
parede(p(8,7)).
parede(p(8,8)).
parede(p(8,9)).
parede(p(8,10)).
```

Arquivo caso_1.pl

```
:- [listas].

% elevadores
elevador(4).
elevador(9).

%Definindo lixeiras
lixeira(p(1,3)).
lixeira(p(10,5)).


%Definindo PowerStation
powerstation(p(10,3)).
```

```
parede(p(5,1)).
parede(p(6,1)).
parede(p(7,2)).
parede(p(7,5)).
```

Arquivo caso_1.pl

```
:- [listas].
:- [busca_profundidade].
:- [busca_largura].


% ?- solucao_bp([p(8,10),0,[p(3,3), p(3,5), p(2,8), p(7,2),
p(7,7)]], Meta).

% elevadores
elevador(4).
elevador(9).

%Definindo lixeiras
lixeira(p(1,3)).
lixeira(p(10,5)).
lixeira(p(6,6)).


%Definindo PowerStation
powerstation(p(8,10)).


parede(p(1,9)).
parede(p(2,6)).
parede(p(5,1)).
parede(p(5,2)).
parede(p(5,3)).
parede(p(5,4)).
parede(p(5,5)).
parede(p(5,6)).
```

```
parede(p(5,7)).
parede(p(5,8)).
parede(p(5,9)).
parede(p(7,4)).
parede(p(7,5)).
parede(p(8,1)).
parede(p(10,1)).
parede(p(10,3)).
parede(p(10,4)).
parede(p(10,8)).
parede(p(10,9)).
parede(p(10,10)).
```

Arquivo aadp.pl:

```
% Agente Aspirador de Pó
:- [listas].
:- [busca_profundidade].
:- [busca_largura].
:- [caso_0].   % trocar para testar outros casos

% verificando limites
fora_do_mapa(p(X,Y)) :-
  X = 0;
  Y = 0;
  X = 11;
  Y = 6.

pode_passar(_,Pos2) :-
  not(parede(Pos2)),
  not(fora_do_mapa(Pos2)).

% pegando sujeira
s([Pos, Sacola, Sujeiras], [Pos, Sacola2, Sujeiras2]) :-
%estado (P,Sa,Su) vem antes de (P,Sa2,Su2) se
    pertence(Pos,Sujeiras),
                            %Essa posicao pertence à lista de
```

```
sujeiras &
    retirar_elemento(Pos,Sujeiras,Sujeiras2),
              %Sujeiras 2 é sujeira sem o 'elemento' Pos &
    Sacola < 2,
                                        % A sacola não está cheia
                  &
    Sacola2 is Sacola + 1, writeln('limpou sujeira').
% Sa2 tem 1 lixo a mais que Sa

% esvaziando sacola na lixeira
s([Pos,Sacola,Sujeiras],[Pos,Sacola2,Sujeiras]) :-
%(P,Sa,Su) vem antes de (P,Sa2,Su2) se
  lixeira(Pos),
                                         %há uma lixeira em P &
  Sacola > 0,
                                       %Tem lixo na sacola  &
  Sacola2 is 0, writeln('esvaziou sacola').
          %A nova sacola está vazia

% andando em X
s([p(X, Y), Sacola, Sujeiras], [p(SX, Y), Sacola, Sujeiras]) :-
% (XY,Sa,Su) vem antes de (SXY,Sa,Su) se
    (SX is X + 1 ; SX is X - 1),
                              % o x novo está à um passo do
antigo, seja direita ou esquerda
    pode_passar(p(X,Y),p(SX,Y)).
                              % X -> SX é navegável

%subindo no elevador
s([p(X,Y), Sacola, Sujeiras],[p(X,SY),Sacola,Sujeiras]) :-
%Andar no Y ( apenas no elevador) se
  elevador(X),
                                           %Há um elevador no X
atual
  (SY is Y + 1; SY is Y - 1),
                                    %o Y novo está há um passo do
antigo, subida ou descida
  not(fora_do_mapa(p(X,SY))), writeln('elevador').
     %a nova posição não está fora do mapa

meta([Pos, 0, Lixos]) :-
```

```
                %A meta é um estado onde o robo está na
powerStation,
  powerstation(Pos),
                            %e a lista de lixos está vazia
  Lixos = [], writeln('pronto').
```

7. Desafios

Primeiramente, existiram dificuldades em iniciar o projeto. Foram feitas várias suposições e teorias de forma inadequada e limitada ao pensamento de uma linguagem de programação estruturada. Portanto, iniciou-se com um objetivo menor. A ideia foi criar um programa que retornasse apenas o caminho para o agente alcançar duas unidades de sujeira, com o intuito de entender por partes o que o projeto deveria realizar.

Ainda com um objetivo menor, houveram erros na execução e definição de mudança de estados. Foi seguida uma intuição inicial de tentar tratar a movimentação do agente juntamente da ação de pegar as sujeiras. E apenas foi possível progredir com o projeto quando foram separados as duas ações nas regras de sucessão de estados.

Além disso, durante a construção do projeto ainda houveram outros tipos de impasses como o tratamento da lista de sujeiras, a modelagem das paredes e na ordem de uso das regras. Porém, o progresso do projeto foi crescente e satisfatório.

8. Referências

Para produção desse projeto foram utilizados trechos de códigos disponibilizados nas aulas de Inteligência Artificial, Pelo Professor Murilo Naldi.

Também foi consultada a documentação do swi-prolog, disponível em: http://www.swi-prolog.org/pldoc/index.html