Inteligência Artificial

Projeto de um Agente Aspirador de Pó (AADP) Utilizando a Linguagem Prolog

Caio Ueno	743516
Gabriel Cheban	743535
João Augusto Leite	743551

Prof. Dr. Murilo Naldi
Departamento de Computação
Universidade Federal de São Carlos

1. Introdução

Neste projeto foi desenvolvido um programa na linguagem *Prolog*, que simula um Agente Aspirador de Pó que, através de uma busca cega, faz a limpeza de um prédio mapeado em uma matriz com as posições representadas por meio de índices.

Nesse cenário é possível que haja elevadores, utilizados para mudar de andar no prédio. Eles não necessariamente cobrem todos os andares, entretanto, todas as portas de elevadores são adjacentes verticalmente, ou seja, o robô pode utilizar o elevador para subir/descer somente um andar por vez. Há também as paredes que bloqueiam a passagem do agente pelo andar, e, eventualmente, não permitem o acesso de determinadas posições pelo agente.

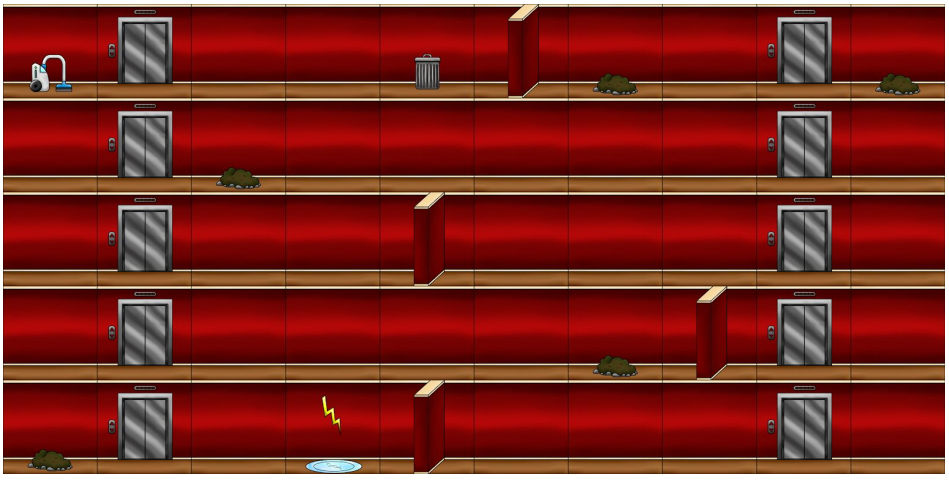

Figura 1: Exemplo de um cenário válido.

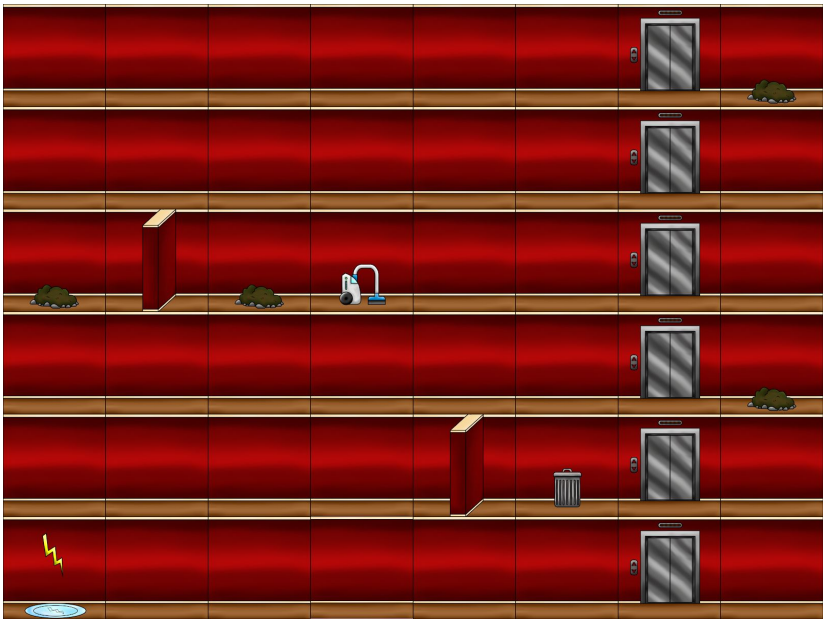

Figura 2: Exemplo de um cenário com sujeira bloqueada por parede.

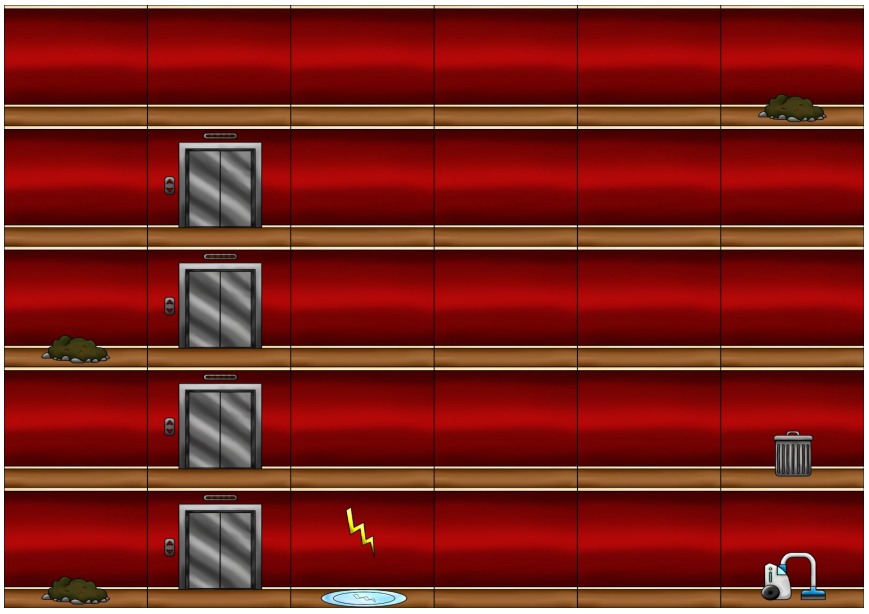

Figura 3: Exemplo de um cenário com sujeira inacessível.

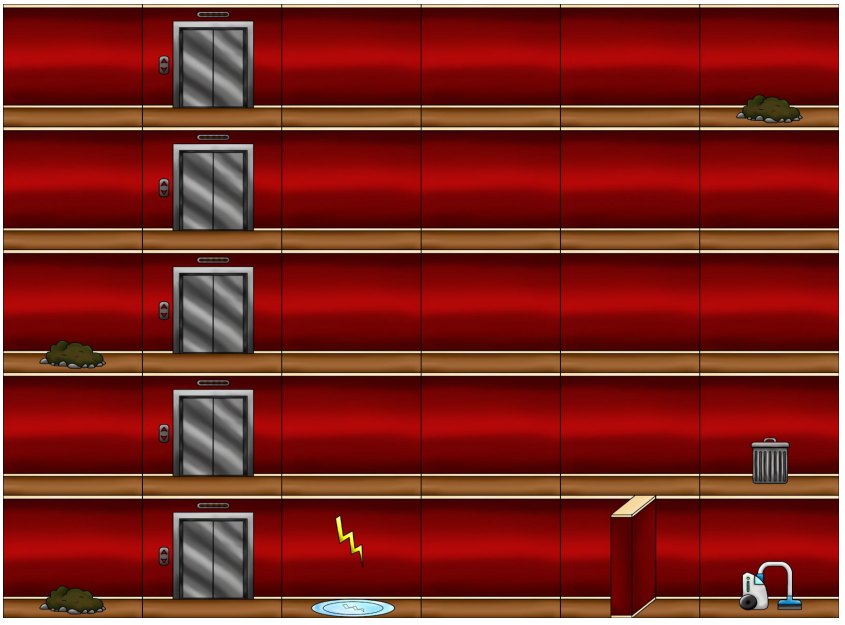

Figura 4: Exemplo de um cenário com aadp bloqueado.

O agente deve ser capaz de operar em todos esses cenários adversos e agir de maneira adequada, sinalizando a incapacidade de resolver o problema em casos específicos, esse sinal é feito utilizando a função *write* do *Prolog*.

2. Base de Conhecimento

Primordialmente serão apresentados os fatos da base de conhecimento. Para que o Agente Aspirador de Pó (AADP) realize a busca são necessárias: as coordenadas das sujeiras, se houverem, da lixeira e da *dockstation*. Além disso, a

base de conhecimento contém dois fatos sobre a capacidade, que representam quantas sujeiras o AADP está carregando atualmente. Isso determina qual será a próxima ação do agente.

A maior parte dos fatos é feita utilizando a estrutura: objeto(*coordenada*, *nome*), em que *coordenada* é uma lista com exatamente dois elementos, sendo o primeiro a coordenada X da matriz e o segundo a coordenada Y, e *nome* é o objeto em si, podendo ser: sujeira, elevador, lixeira, *dockstation* (posição final do aadp), parede, aadp (posição inicial deste) ou limite (tamanho da matriz, ou seja, quantos espaços em X e quantos andares em Y).

Como exemplo de cenário, a seguir são mostrados os fatos para o AADP. Estes por sua vez podem ser alterados para modelar qualquer outro cenário, inclusive aqueles que não possuem lixeira ou *dockstation*, ou ainda é possível aumentar a capacidade do agente.

```
% Caso 1: Modelo 1 da Especificação
% Fatos: Objetos em cada posição.
objeto([10,5], limite).
objeto([10,1], aadp).
objeto([3,4], sujeira).
objeto([7,2], sujeira).
objeto([7,5], sujeira).
objeto([10,5], sujeira).
objeto([1,1], sujeira).
objeto([5,5], lixeira).
objeto([4,1], dockstation).
objeto([2,1], elevador).
objeto([2,2], elevador).
objeto([2,3], elevador).
objeto([2,4], elevador).
objeto([2,5], elevador).
objeto([9,1], elevador).
objeto([9,2], elevador).
objeto([9,3], elevador).
objeto([9,4], elevador).
objeto([9,5], elevador).
objeto([6,5], parede).
objeto([5,3], parede).
objeto([5,1], parede).
objeto([8,2], parede).
% Carga suportada pelo robô
capacidade(0).
capacidade(1).
```

3. Conjunto de Regras

A seguir estão dispostas as regras utilizadas pelo AADP na resolução do problema. Uma breve explicação é colocada logo abaixo da regra para melhor entendimento, além de um exemplo de como invocá-la no interpretador do *Prolog*.

```
:- dynamic objeto/2.
:- style_check(-singleton).
```

Trecho de código para poder remover da base de dados as posições das sujeiras uma vez aspiradas, utilizando a regra *retract* e eliminar os *warnings* sobre variáveis não unificadas.

```
% Verificar se um elemento pertence a uma lista (retirado do material de apoio)
pertence(Elem, [Elem|_]).
pertence(Elem, [_|Cauda]) :- pertence(Elem, Cauda).
```

Verifica se um elemento pertence a uma lista. Ou o elemento é a cabeça da lista, ou ele pertence a cauda.

Exemplo: ?- pertence(2, [1,2,3,4]).

```
% Concatenar listas (retirado do material de apoio)
concatena([ ], L, L).
concatena([Cabeca|Cauda], L2, [Cabeca|Resultado]) :-
  concatena(Cauda, L2, Resultado).
```

Concatenação de duas listas. É necessário remover todos os elementos da primeira lista, um a um, e colocá-los na segunda, respeitando a ordem que foram tirados.

Exemplo: ?- concatena([1,2,3], [4,5,6], L).

```
% Inverter uma listas (retirado do material de apoio -A Cartilha Prolog-)
inverter(X, Y) :- reverso([], X, Y).
reverso(L, [], L).
reverso(L, [X|Y], Z) :- reverso([X|L], Y, Z).
```

Inverte uma lista. Retira a cabeça da lista passada e insere na cabeça da lista nova.
Exemplo: ?- inverter([3,2,1], L).

```
% Remover elementos duplicados (caso forem vizinhos -Código cedido-)
removerDuplicata([], []).
removerDuplicata([X], [X]).
removerDuplicata([X, X|T], [X|R]) :-  removerDuplicata([X|T], [X|R]).
removerDuplicata([X, Y|T], [X|R]) :-
  X \== Y,
  removerDuplicata([Y|T], R).
```

Dada uma lista, remove elementos repetidos que são adjacentes. Usado para corrigir as posições iniciais que a busca retorna.

```
% Verificar se ou qual é o objeto
verifica([PosX, PosY], Objeto) :-
  bagof(PosObjeto, objeto(PosObjeto, Objeto), ListaObjetos),
  pertence([PosX,PosY], ListaObjetos).
```

Verifica quais objetos encontram-se em dada posição, ou ainda, em quais posições encontram-se um tipo de objeto.

Exemplo: ?- verifica([2,1], Objetos).
?- verifica(Pos, sujeira).

```
% Obter as casas adjacentes à casa atual
adjacente([X, Y], [NextX, NextY]) :-
  % Válido para todos
  ((NextX is X - 1;
  NextX is X + 1),
  NextY is Y);
  % Apenas para elevadores
  ((NextY is Y - 1;
  NextY is Y + 1),
  NextX is X,
  verifica([X, Y], elevador),
  verifica([NextX, NextY], elevador)).
```

Obtenção dos adjacentes de uma posição. Todas as posições possuem como adjacentes seus vizinhos, caso seus vizinhos ultrapassem os limites da matriz, esses não são considerados durante a movimentação. Para os elevadores é necessário acrescentar os adjacentes superior e inferior, que também são descartados caso ultrapassem os limites do cenário.
Exemplo: ?- adjacente([1,1], X).

```
% Verificar se uma casa é inválida
invalida([X, Y], [MaxX, MaxY]) :-
  X is 0;
  Y is 0;
  X is MaxX + 1;
  Y is MaxY + 1;
  verifica([X, Y], parede).
```

Determina se uma dada posição é inválida, ou seja, não será considerada dentro do cenário - ultrapassa algum limite superior ou inferior do índice X ou Y da matriz ou é uma parede.
Exemplo: ?- invalida(Pos, Limites).

```
incrementa(Qtde, Qtde_novo) :- Qtde_novo is Qtde + 1.
```

Regra para incrementar um dado parâmetro.
Exemplo: ?- incrementa(2, X).

```
% Movimento do robô
movimento([X, Y], [NextX, NextY], [MaxX, MaxY]) :-
  adjacente([X, Y], [NextX, NextY]),
  not(invalida([NextX, NextY], [MaxX, MaxY])).
```

Regra de movimento do agente, onde ele pode se movimentar para as posições adjacentes desde que elas não sejam inválidas.
Exemplo: ?- movimento([1,1], X, [5,10]).

```
% Meta: se a posição passada é a posição do objeto buscado
meta(Estado, Objeto) :- verifica(Estado, Objeto).
```

Generalização do predicado meta, onde é possível escolher qual é o objeto de interesse que indica se um estado - posição na matriz - é uma meta ou não.

```
% Busca em profundidade (retirado do material de apoio -adaptado-)
buscaProfundidade(Estado, EstadoAux, Rota, [EstadoAux|Rota], Objeto) :-
  meta(Estado, Objeto).
buscaProfundidade(Estado, EstadoAux, Rota, Solucao, Objeto) :-
  objeto(Limite, limite),
  movimento(Estado, Sucessor, Limite),
  not(pertence(Sucessor, [EstadoAux|Rota])),
  buscaProfundidade(Sucessor, Sucessor, [EstadoAux|Rota], Solucao, Objeto).
```

Busca em profundidade, através de uma busca cega. Passa pelos estados em busca do estado meta, adicionando todos os estados intermediários necessários para se chegar ao estado meta.

```
% Regra para encontrar o lixo
encontraObjeto(EstadoInicio, EstadoFim, [EstadoFim|Rota], Objeto) :-
  buscaProfundidade(EstadoInicio, EstadoInicio, [], [EstadoFim|Rota], Objeto).
```

Aplica uma busca em Profundidade para encontrar determinado objeto, como sujeiras, lixeiras e a *dockstation*.

```
% Regra para procurar até 2 sujeiras (Caso base)
procuraLixo(EstadoInicio, EstadoFim, Rota, Objeto, Qtde) :-
  not(capacidade(Qtde));
  (not(encontraObjeto(EstadoInicio, EstadoFim, Aux, sujeira)),
  Rota = []).
```

Caso base da busca por sujeiras, se não houver mais capacidade ou se não houver mais sujeiras, então não é possível procurar por sujeiras.

```
% Regra para procurar até 2 sujeiras (Caso recursivo)
procuraLixo(EstadoInicio, EstadoFim, Rota, Objeto, Qtde) :-
  incrementa(Qtde, Qtde_novo),
  encontraObjeto(EstadoInicio, EstadoMedio, RotaLixo, Objeto),
  retract(objeto(EstadoMedio, sujeira)),
  procuraLixo(EstadoMedio, EstadoFim, RotaOutra, Objeto, Qtde_novo),
  concatena(RotaOutra, RotaLixo, Rota),
  pertence(EstadoFim, Rota). % Unifica EstadoFim com a cabeça de Rota
```

Encontra uma sujeira, remove-a da base de conhecimento, incrementa a capacidade, gera uma lista com o trajeto do AADP e chama novamente a regra, até que a capacidade seja 2 ou não encontre uma sujeira.

Exemplo: ?- procuraLixo([1,1], X, R, sujeira, 0).

```
% Regra para procurar até 2 sujeiras e ir à lixeira (Caso base)
limpaPredio(EstadoInicio, EstadoFim, Rota) :-
  not(encontraObjeto(EstadoInicio, EstadoFim, Rota, sujeira)).
```

Caso base da regra, caso não haja mais sujeiras então é necessário parar a busca.

```
% Regra para procurar até 2 sujeiras e ir à lixeira (Caso recursivo)
limpaPredio(EstadoInicio, EstadoFim, Rota) :-
  procuraLixo(EstadoInicio, EstadoMedio, RotaSujeira, sujeira, 0),
  encontraObjeto(EstadoMedio, EstadoFim, RotaLixeira, lixeira),
  concatena(RotaLixeira, RotaSujeira, Rota1),
  limpaPredio(EstadoFim, EstadoFinal, Rota2),
  concatena(Rota2, Rota1, Rota).
```

Caso recursivo da regra, repete a busca de até duas sujeiras, vai até a lixeira, concatena as rotas, isso é feito até que não haja mais sujeiras no prédio, todas as rotas que foram concatenadas antes, serão concatenadas em uma única rota.

```
% Regra para determinar se é possível limpar o prédio
testaMapa(EstadoInicio) :-
  encontraObjeto(EstadoInicio, EstadoFim1, RotaLixeira, lixeira),
  encontraObjeto(EstadoInicio, EstadoFim2, RotaDock, dockstation).
```

Regra utilizada antes da busca pelas sujeiras, ela faz uma busca pela lixeira e pelo *dockstation*, caso esses estejam inacessíveis então o mapa não é válido.

```
% Regra para verificar se irá limpar o prédio
executa(EstadoInicio, EstadoFim, Rota) :-
  not(testaMapa(EstadoInicio)),
  write('Lixeira ou Dock Station inexistentes ou bloqueadas'), nl, !.
```

Regra de um dos casos bases, se o mapa não passou pela verificação da existência e alcançabilidade de seus elementos então mostra uma mensagem para o usuário.

```
% Regra para verificar se o prédio está limpo
executa(EstadoInicio, EstadoFim, Rota) :-
  not(encontraObjeto(EstadoInicio, EstadoFim3, RotaLixo, sujeira)),
  encontraObjeto(EstadoInicio, EstadoFim, RotaAux, dockstation),
  inverter(RotaAux, Rota),
  write('Caminho: '),
  write(Rota), nl, !.
```

Trata a possibilidade do prédio já estar limpo, nessa situação o AADP se encaminha para o *dockstation* e termina a execução.

```
% Regra para procurar as sujeiras possíveis, ir à lixeira e ir à Dock Station
executa(EstadoInicio, EstadoFim, Rota) :-
  limpaPredio(EstadoInicio, EstadoMedio, RotaPredio),
  encontraObjeto(EstadoMedio, EstadoFim, RotaDS, dockstation),
  concatena(RotaDS, RotaPredio, RotaAux),
  inverter(RotaAux, RotaCerta),
  removerDuplicata(RotaCerta, Rota),
  write('Caminho: '),
  write(Rota), nl, !.
```

Após feitas verificações em relação ao mapa, então é necessário executar a limpeza das sujeiras e por fim ir até o *dockstation*.

```
% Chamada para resolver o problema
resolvePredio(EstadoFim, Rota) :-
  objeto(EstadoInicio, aadp),
  executa(EstadoInicio, EstadoFim, Rota), !.
```

Regra invocada para a resolução do cenário pelo AADP.

Exemplo: ? resolvePredio(X, Rota).

4. Código Completo

O código desenvolvido, junto com um cenário válido, será mostrado a seguir em formato texto e estilizado para utilização posterior.

```
:- dynamic objeto/2.
:- style_check(-singleton).

% Fatos: Objetos em cada posição.
objeto([6,1],aadp).
objeto([6,5], limite).
objeto([1,1], sujeira).
objeto([1,3], sujeira).
objeto([6,5], sujeira).
objeto([6,2], lixeira).
objeto([6,4], dockstation).
objeto([2,1], elevador).
objeto([2,2], elevador).
objeto([2,3], elevador).
objeto([2,4], elevador).
```

```
objeto([2,5], elevador).
objeto([5,4], parede).
% Carga suportada pelo robô
capacidade(0).
capacidade(1).

  % Regras
  % Verificar se um elemento pertence a uma lista (retirado do
material de apoio)
  pertence(Elem, [Elem|_]).
  pertence(Elem, [_|Cauda]) :- pertence(Elem, Cauda).

  % Concatenar listas (retirado do material de apoio)
  concatena([ ], L, L).
  concatena([Cabeca|Cauda], L2, [Cabeca|Resultado]) :-
concatena(Cauda, L2, Resultado).

  % Inverter uma listas (retirado do material de apoio -A Cartilha
Prolog-)
  inverter(X, Y) :- reverso([], X, Y).
  reverso(L, [], L).
  reverso(L, [X|Y], Z) :- reverso([X|L], Y, Z).

  % Remover elementos duplicados (caso forem vizinhos -Código
cedido-)
  removerDuplicata([], []).
  removerDuplicata([X], [X]).
  removerDuplicata([X, X|T], [X|R]) :-  removerDuplicata([X|T],
[X|R]).
  removerDuplicata([X, Y|T], [X|R]) :-
      X \== Y,
      removerDuplicata([Y|T], R).

  % Verificar se ou qual é o objeto
  verifica([PosX, PosY], Objeto) :-
      bagof(PosObjeto, objeto(PosObjeto, Objeto), ListaObjetos),
      pertence([PosX,PosY], ListaObjetos).

  % Obter as casas adjacentes à casa atual
  adjacente([X, Y], [NextX, NextY]) :-
      % Válido para todos
```

```
    ((NextX is X - 1;
    NextX is X + 1),
    NextY is Y);
    % Apenas para elevadores
    ((NextY is Y - 1;
    NextY is Y + 1),
    NextX is X,
    verifica([X, Y], elevador),
    verifica([NextX, NextY], elevador)).

  % Verificar se uma casa é inválida
  invalida([X, Y], [MaxX, MaxY]) :-
    X is 0;
    Y is 0;
    X is MaxX + 1;
    Y is MaxY + 1;
    verifica([X, Y], parede).

  incrementa(Qtde, Qtde_novo) :- Qtde_novo is Qtde + 1.

  % Movimento do robô
  movimento([X, Y], [NextX, NextY], [MaxX, MaxY]) :-
    adjacente([X, Y], [NextX, NextY]),
    not(invalida([NextX, NextY], [MaxX, MaxY])).

  % Meta: se a posição passada é a posição do objeto buscado
  meta(Estado, Objeto) :- verifica(Estado, Objeto).

  % Busca em profundidade (retirado do material de apoio
-adaptado-)
  buscaProfundidade(Estado, EstadoAux, Rota, [EstadoAux|Rota],
Objeto) :-
    meta(Estado, Objeto).
  buscaProfundidade(Estado, EstadoAux, Rota, Solucao, Objeto) :-
    objeto(Limite, limite),
    movimento(Estado, Sucessor, Limite),
    not(pertence(Sucessor, [EstadoAux|Rota])),
    buscaProfundidade(Sucessor, Sucessor, [EstadoAux|Rota],
Solucao, Objeto).

  % Regra para encontrar o lixo
```

```
  encontraObjeto(EstadoInicio, EstadoFim, [EstadoFim|Rota],
Objeto) :-
      buscaProfundidade(EstadoInicio, EstadoInicio, [],
[EstadoFim|Rota], Objeto).

  % Regra para procurar até 2 sujeiras (Caso base)
  procuraLixo(EstadoInicio, EstadoFim, Rota, Objeto, Qtde) :-
      not(capacidade(Qtde));
      (not(encontraObjeto(EstadoInicio, EstadoFim, Aux, sujeira)),
      Rota = []).

  % Regra para procurar até 2 sujeiras (Caso recursivo)
  procuraLixo(EstadoInicio, EstadoFim, Rota, Objeto, Qtde) :-
      incrementa(Qtde, Qtde_novo),
      encontraObjeto(EstadoInicio, EstadoMedio, RotaLixo, Objeto),
      retract(objeto(EstadoMedio, sujeira)),
      procuraLixo(EstadoMedio, EstadoFim, RotaOutra, Objeto,
Qtde_novo),
      concatena(RotaOutra, RotaLixo, Rota),
      pertence(EstadoFim, Rota). % Unifica EstadoFim com a cabeça
de Rota

  % Regra para procurar até 2 sujeiras e ir à lixeira (Caso base)
  limpaPredio(EstadoInicio, EstadoFim, Rota) :-
      not(encontraObjeto(EstadoInicio, EstadoFim, Rota, sujeira)).

  % Regra para procurar até 2 sujeiras e ir à lixeira (Caso
recursivo)
  limpaPredio(EstadoInicio, EstadoFim, Rota) :-
      procuraLixo(EstadoInicio, EstadoMedio, RotaSujeira, sujeira,
0),
      encontraObjeto(EstadoMedio, EstadoFim, RotaLixeira, lixeira),
      concatena(RotaLixeira, RotaSujeira, Rota1),
      limpaPredio(EstadoFim, EstadoFinal, Rota2),
      concatena(Rota2, Rota1, Rota).

% Regra para determinar se é possível limpar o prédio
  testaMapa(EstadoInicio) :-
      encontraObjeto(EstadoInicio, EstadoFim1, RotaLixeira,
lixeira),
      encontraObjeto(EstadoInicio, EstadoFim2, RotaDock,
```

```
dockstation).

  % Regra para verificar se irá limpar o prédio
  executa(EstadoInicio, EstadoFim, Rota) :-
      not(testaMapa(EstadoInicio)),
      write('Lixeira ou Dock Station inexistentes ou bloqueadas'),
nl, !.

  % Regra para verificar se o prédio está limpo
  executa(EstadoInicio, EstadoFim, Rota) :-
      not(encontraObjeto(EstadoInicio, EstadoFim3, RotaLixo,
sujeira)),
      encontraObjeto(EstadoInicio, EstadoFim, RotaAux,
dockstation),
      inverter(RotaAux, Rota),
      write('Caminho: '),
      write(Rota), nl, !.

  % Regra para procurar as sujeiras possíveis, ir à lixeira e ir à
Dock Station
  executa(EstadoInicio, EstadoFim, Rota) :-
      limpaPredio(EstadoInicio, EstadoMedio, RotaPredio),
      encontraObjeto(EstadoMedio, EstadoFim, RotaDS, dockstation),
      concatena(RotaDS, RotaPredio, RotaAux),
      inverter(RotaAux, RotaCerta),
      removerDuplicata(RotaCerta, Rota),
      write('Caminho: '),
      write(Rota), nl, !.

  % Chamada para resolver o problema
  resolvePredio(EstadoFim, Rota) :-
      objeto(EstadoInicio, aadp),
      executa(EstadoInicio, EstadoFim, Rota), !.
```

5. Referências

http://www.dei.isep.ipp.pt/~jtavares/ALGAV/downloads/ALGAV_TP_aula4.pdf;

http://www.ppgia.pucpr.br/~fabricio/ftp/Aulas/Programa%E7%E3o%20L%F3gica%20

-%20Tecpar/listas-prolog.pdf;

https://stackoverflow.com/questions/38141764/removing-consecutive-duplicates-from-a-list-in-prolog;

Nicoletti, M. C. A Cartilha Prolog, 2003;

Mateial de apoio cedido pelo professor Dr. Murilo Naldi.