Universidade Federal de São Carlos
Departamento de Computação

Trabalho de Inteligência Artificial
Agente Aspirador de Pó

Professor Dr. Murilo Coelho Naldi

Alisson Hayasi da Costa (RA: 726494)
Bruna Zamith Santos (RA: 628093)

05/2019
São Carlos - SP, Brasil

Conteúdo

1 Introdução

Este relatório visa descrever a solução do desafio “Agente Aspirador de Pó” em Prolog (Linguagem de Programação Lógica [1]) e a lógica por trás da proposta de solução pelos autores deste documento.

O documento está organizado como segue: Seção 2 introduz o problema do “Agente Aspirador de Pó”; a Seção 3 expõe os principais conceitos necessários para resolução geral do problema; a Seção 4 apresenta a base de conhecimento aplicada no projeto; por fim, a Seção 5 apresenta como executar o programa e os resultados das simulações. Nos apêndices estão os códigos completos usados para os exemplos de execução da Seção 5.

2 Problema

O problema do “Agente Aspirador de Pó” é composto pelos seguintes elementos:

1 cenário matricial	*N* sujeiras
1 agente aspirador de pó (AADP)	*M* paredes
1 *Dock Station*	*O* elevadores
1 Lixeira	

Sendo *N*, *M* e *O* quantidades variáveis. Além disso, o problema também contém as seguintes restrições:

- Posição inicial do AADP é definida pelo usuário;
- A movimentação do AADP é livre na horizontal e na vertical, exceto se:
  - Houver uma parede bloqueando;
  - For margem do cenário;
- A movimentação do AADP na vertical só se dá por intermédio dos elevadores;
- O AADP deve coletar uma sujeira quando passar por ela;
- A cadas duas sujeiras coletadas, o AADP deve esvaziar-se na lixeira.

O objetivo final é que o AADP saia da posição inicial, colete todas as *N* sujeiras do cenário e termine na *Dock Station*.

Os elementos do cenário estão representados na Figura 1.

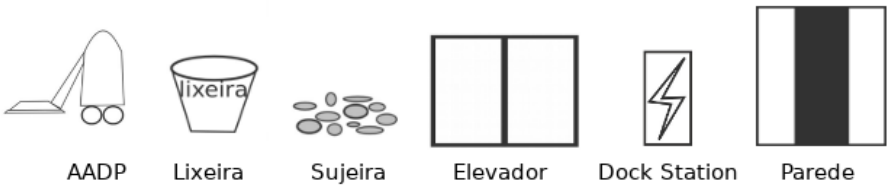

Figura 1: Representação dos elementos do cenário. Fonte: Material de Apoio, professor Dr. Murilo Naldi Coelho.

3 Conceitos

De acordo com a descrição do problema apresentada na Seção 2, identificamos alguns tópicos cujo entendimento era essencial para a solução do problema. Eles são descritos a seguir, resumidamente, junto com como eles foram aplicados na modelagem do problema.

3.1 Modelagem de Estados e Transição

Um problema de busca cega pode ser representado de diversas formas. Uma delas é baseada em seus estados. Um estado conceitualmente deve contemplar toda a situação do ambiente. As transições, por sua vez, definem a troca entre estados definida por ações específicas.

Apesar da definição do estado referir-se a um modelo abrangente, posto que a solução final é definir um caminho, podemos simplificadamente representar os estados como sendo apenas as posições deste caminho e tratar as ações na própria regra de busca.

A Figura 2 exibe a representação do cenário como posições num eixo cartesiano. Um estado é representado pela posição (X,Y) onde o AADP se encontra.

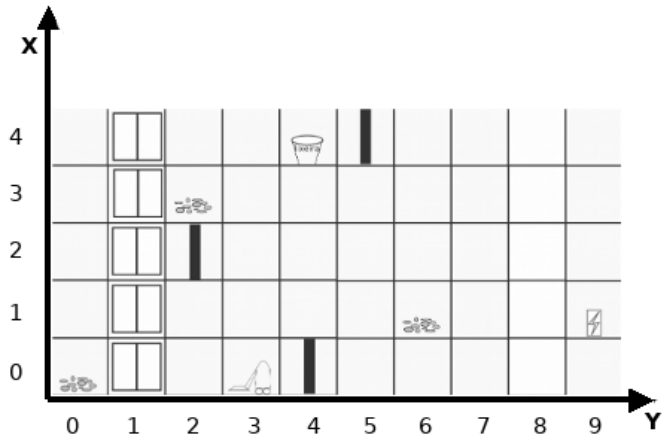

Figura 2: Representação do cenário como um eixo cartesiano. Fonte: Baseado no Material de Apoio, professor Dr. Murilo Naldi Coelho.

3.2 Busca em Profundidade

O algoritmo de Busca em Profundidade busca o caminho mais "profundo"possível em um grafo. Em outras palavras, isso significa que o processo de descoberta de estados ocorre explorando todas as ações inexploradas do estado **e** mais recentemente descoberto. Depois que todas as ações que têm origem em **e** foram descobertas, a busca regressa pelo mesmo caminho e passa a explorar todas as ações de possíveis de serem exploradas pelo estado que deu origem a **e**. Esse processo é feito continuamente até que um estado meta seja alcançado.

O problema do AADP é modelado como uma combinação de diferentes buscas consecutivas:

- Buscar sujeira (se a capacidade máxima não tiver sido atingida e ainda existirem sujeiras no caminho);
- Buscar lixeira (se a capacidade máxima tiver sido atingida);
- Buscar *Dock Station* (se não houverem mais sujeiras no caminho).

4 Base de Conhecimento

4.1 Definição do Ambiente

O cenário na base de conhecimento é definido pelo tamanho da matriz (X horizontal e Y vertical). Além disso, é também definido pelos fatos que indicam as posições das paredes, elevadores, *dock station*, lixeira e sujeiras, ainda considerando um plano cartesiano. Não obstante, é definida também a capacidade máxima do AADP.

Exemplos de definições são ilustrados a seguir:

```
cenario(10,5). % Cenario tem tamanho 10x5
parede(4,0). % Parede na posicao (4,0)
elevador(1,0). % Elevador na posicao (1,0)
dockStation(0,0). % Dock Station na posicao (0,0)
lixeira(4,4). % Lixeira na posicao (4,4)
sujeiras([[2,3],[6,1],[6,4]]). % Lista com posicoes das sujeiras
capacidadeMax(2). % Capacidade maxima igual a 2 sujeiras
```

A definição do ambiente desta forma torna a solução escalável: Funciona para diferentes definições de cenários.

4.2 Regras Gerais

Nessa seção serão descritas as quatro regras gerais para manipulação de listas que serão utilizadas na solução do problema: ***pertence***, ***concatena***, ***pop*** e ***reverse***.

A regra ***pertence*** recebe como entrada um elemento e uma lista e retorna verdadeiro se o elemento pertence àquela lista. Caso contrário, retorna falso. Ela é formada por uma regra base - elemento está na cabeça da lista - e uma regra recursiva - elemento pertence à cauda da lista.

```
% pertence(Entrada1,Entrada2)
% Verifica se elemento pertence a lista
% Entrada1: Elem
% Entrada2: Lista
pertence(Elem,[Elem|_]).
pertence(Elem,[_|Cauda]) :- pertence(Elem,Cauda).
```

A regra ***concatena*** recebe como entrada duas listas e retorna a concatenação dessas listas. Ela tem a mesma implementação que a função nativa “append/3” do Prolog.

```
% concatena(Entrada1,Entrada2,Saida)
% Concatena duas listas
% Entrada1: Lista 1
% Entrada2: Lista 2
% Saida: Concatenacao das entradas
concatena([X|Y],Z,[X|W]) :- concatena(Y,Z,W).
concatena([],X,X).
```

A regra ***pop*** recebe como entrada uma lista e retorna duas saídas: o primeiro elemento da lista e a lista restante (após a remoção do primeiro elemento).

```
% pop(Entrada,Saida1,Saida2)
% Faz um pop na lista
% Entrada: Lista
% Saida1: Primeiro elemento da lista
% Saida2: Lista sem o elemento
pop([Elem|Cauda],Elem,Cauda).
```

A regra ***reverse*** recebe como entrada uma lista e retorna a lista invertida. Isto é necessário para inverter o sentido do caminho final.

```
% reverse(Entrada,Saida)
% Inverte elementos na lista
% Entrada: Lista
% Saida: Lista invertida
reverse([],[]).
reverse([Cab|Cauda],R) :- reverse(Cauda,RCauda), concatena(RCauda,[Cab],R).
```

As quatro regras supracitas serão utilizadas no corpo de outras regras a serem definidas nas próximas seções.

4.3 Regras de Movimentação

É preciso definir as regras que checam a possibilidade de realização do movimento pelo AADP. Essas são quatro: ***moveDir***, ***sobeElev***, ***moveEsq*** e ***desceElev***.

A regra ***moveDir*** define que só é possível que o AADP se mova para a direita se não exceder o limite horizontal do cenário e se não houver uma parede impedindo a passagem. Assim, se o movimento for permitido, com uma entrada X e Y, a saída é X + 1 e Y, já que a posição vertical se mantém.

```
% moveDir(Entrada1,Entrada2,Saida1,Saida2)
% Move para direita
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveDir(Xin,Y,Xout,Y) :-
    cenario(Lim,_),
    LimDir is Lim -1, % Eixo comeca no 0
    Xin < LimDir,
    X is Xin + 1,
    not(parede(X,Y)),
    Xout is X.
```

A regra ***sobeElev*** define que só é possível que o AADP se mova para cima não exceder o limite vertical do cenário e se ele estiver na mesma posição que um elevador. Assim, se o movimento for permitido, com uma entrada X e Y, a saída é X e Y + 1, já que a posição horizontal se mantém.

```
% sobeElev(Entrada1,Entrada2,Saida1,Saida2)
% Sobe pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
sobeElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
    cenario(_,Lim),
    LimSup is Lim - 1, % Eixo comeca no 0
    Yin < LimSup,
    Yout is Yin + 1.
```

A regra ***moveEsq*** define que só é possível que o AADP se mova para a esquerda se não exceder o limite horizontal do cenário e se não houver uma parede impedindo a passagem. Assim, se o movimento for permitido, com uma entrada X e Y, a saída é X - 1 e Y, já que a posição vertical se mantém.

```
% moveEsq(Entrada1,Entrada2,Saida1,Saida2)
% Move para esquerda
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveEsq(Xin,Y,Xout,Y) :-
    Xin > 0,
    X is Xin - 1,
    not(parede(X,Y)),
    Xout is X.
```

A regra ***desceElev*** define que só é possível que o AADP se mova para baixo não exceder o limite vertical do cenário e se ele estiver na mesma posição que um elevador. Assim, se o movimento for permitido, com uma entrada X e Y, a saída é X e Y - 1, já que a posição horizontal se mantém.

```
% desceElev(Entrada1,Entrada2,Saida1,Saida2)
% Desce pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
desceElev(X,Yin,X,Yout) :-
```

```
    elevador(X,Yin),
    Yin > 0,
    Yout is Yin - 1.
```

4.4 Regras para Transição de Estados

As regras de transição de estado são definidas pelas várias definições de ***s***. Essas regras são responsáveis por produzir um estado sucessor de acordo com a posição do AADP no estado atual. Elas serão usadas dentro do contexto de busca.

```
% s(Entrada,Saida)
% Transicao de estado
% Entrada: Estado inicial
% Saida: Estado final (sucessor)
s([X,Y],[X1,Y]) :-
    moveDir(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    sobeElev(X,Y,X,Y1).
s([X,Y],[X1,Y]) :-
    moveEsq(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    desceElev(X,Y,X,Y1).
```

4.5 Regras para Buscas

A regra ***meta*** verfica se o estado atual do AADP é um estado meta. Isso é necessário para lidar com a busca, como explicado anteriormente na Seção 3.2.

```
% meta(Entrada1,Entrada2)
% Checa se o estado atual eh o estado meta
% Entrada1: Estado atual
% Entrada2: Estado meta
meta(Estado,Meta) :- Meta == Estado.
```

A regra ***busca*** implementa a busca em profundidade. Ela tem como entrada o caminho inicial (vazio), o estado atual e o estado meta. Com isso, ela retorna o caminho percorrido do estado atual ao estado meta (se houver). A regra é baseada na implementação do Material de Apoio, professor Dr. Murilo Coelho Naldi. A regra base verifica se o estado atual é estado meta e então retorna a concatenação do estado atual com o caminho inicial. A regra recursiva ocorre se não for encontrada a meta, então insere-se o estado atual no caminho e procura-se nos estados sucessores.

```
% busca(Entrada1,Entrada2,Entrada3,Saida)
% Busca em Profundidade
% Entrada1: Caminho inicial
% Entrada2: Estado atual
```

```
% Entrada3: Estado meta
% Saida: Caminho da solucao
% Fonte: Material de Apoio, professor Dr. Murilo Coelho Naldi
busca(Caminho,Estado,Final,[Estado|Caminho]) :- meta(Estado,Final).
busca(Caminho,Estado,Final,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    busca([Estado|Caminho],Sucessor,Final,Solucao).
```

A regra ***solucao*** é aquela que deverá ser chamada pelo usuário (como explicitado na Seção 5). Com base num estado inicial (posição inicial do AADP), o caminho de solução é retornado. Para tanto, ela verifica as sujeiras do cenário e chama a regra limpa.

```
% solucao(Entrada,Saida)
% Recebe estado inicial e retorna o caminho de solucao
% Entrada: Posicao inicial do AADP
% Saida: Caminho percorrido
solucao(Inicial,Solucao) :- sujeiras(Sujeiras),
    limpa(Inicial,0,Sujeiras,[],Solucao).
```

A regra ***limpa*** recebe o estado atual, a capacidade do AADP (quantidade de sujeiras recolhidas até aquele momento), a lista de sujeiras presentes no cenário naquele momento e o caminho percorrido até então. Ela retorna a concatenação do caminho atual com o caminho percorrido até encontrar a meta.

São três metas possíveis, dependendo do cenário:

Cenário	**Meta**
Capacidade máxima não foi atingida, ainda existem sujeiras no cenário	Encontrar sujeira
Capacidade máxima foi atingida, ainda existem sujeiras no cenário	Encontrar lixeira
Não existem sujeiras no cenário	Encontrar *Dock Station*. Encerra.

Exceto pelo terceiro caso (não existem sujeiras no cenário), os outros casos devem chamar busca recursivamente. Os caminhos entre as diversas buscas são concatenados. Para a regra que busca a sujeira, cada meta será um elemento da lista de sujeiras - a regra pop definida anteriormente (Seção 4.2) garante que a lista de sujeiras vai sendo sempre reduzida. Ao encontrar a sujeira, a capacidade do AADP também é incrementada.

```
% limpa(Entrada1,Entrada2,Entrada3,Entrada4,Saida)
% Define regras para limpeza
% Entrada1: Estado atual
% Entrada2: Capacidade do AADP
% Entrada3: Lista de sujeiras a serem encontradas
% Entrada4: Caminho atual
% Saida: Caminho final
```

```
% Quando nao tiver mais sujeira,vai para Dock Station
limpa(Estado,_,[],Caminho,SolucaoFinal) :-
    dockStation(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,SolucaoSaida),
    reverse(SolucaoSaida,SolucaoFinal).
% Busca sujeira se capacidade menor que capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade < CapacidadeMax,
    pop(Sujeiras,Alvo,NovaSujeiras),
    busca([],Estado,Alvo,Solucao),
    NovaCapacidade is Capacidade + 1,
    concatena(Solucao,Caminho,NovoCaminho),
    limpa(Alvo,NovaCapacidade,NovaSujeiras,NovoCaminho,CaminhoResposta).
% Esvazia na lixeira se capacidade igual a capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade >= CapacidadeMax,
    lixeira(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,NovoCaminho),
    limpa([X,Y],0,Sujeiras,NovoCaminho,CaminhoResposta).
```

5 Execução

Para executar a solução explicitada neste código, é preciso carregar a base de conhecimentos no Prolog, por meio do comando:

```
['aadp.pl'].
```

Então, podemos buscar a Solução a partir de uma posição inicial ([0,0] no exemplo abaixo):

```
solucao([0,0],Solucao).
```

Para efeitos de simulação, testamos nossa solução em 3 diferentes cenários.

5.1 Cenário 1

O primeiro cenário é representado na Figura 3.

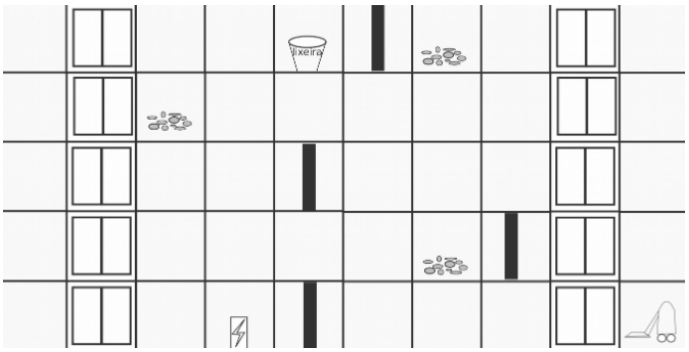

Figura 3: Representação do Cenário 1. Fonte: Material de Apoio, professor Dr. Murilo Naldi Coelho.

As definições do ambiente na base do conhecimento são:

```
% Tamanho Cenario
cenario(10,5).
% Paredes
parede(4,0).
parede(7,1).
parede(4,2).
parede(5,4).
% Elevadores
elevador(1,0).
elevador(1,1).
elevador(1,2).
elevador(1,3).
elevador(1,4).
elevador(8,0).
elevador(8,1).
```

```
elevador(8,2).
elevador(8,3).
elevador(8,4).
% Dock Station
dockStation(3,0).
% Lixeira
lixeira(4,4).
% Sujeiras
sujeiras([[2,3],[6,1],[6,4]]).
% Capacidade AADP
capacidadeMax(2).
```

As demais regras não devem ser modificadas. O resultado obtido está ilustrado na Figura 4.

```
?- solucao([9,0],Solucao).
Solucao = [[9, 0], [8, 0], [8, 1], [8, 2], [8, 3], [7, 3], [6, 3], [5, 3], [4, 3], [3, 3], [2, 3], [2, 3]
, [1, 3], [1, 2], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1], [6, 1], [6, 1], [5, 1], [4, 1], [3, 1], [2, 1],
 [1, 1], [1, 2], [1, 3], [1, 4], [2, 4], [3, 4], [4, 4], [4, 4], [3, 4], [2, 4], [1, 4], [1, 3], [2, 3],
[3, 3], [4, 3], [5, 3], [6, 3], [7, 3], [8, 3], [8, 4], [7, 4], [6, 4], [6, 4], [7, 4], [8, 4], [8, 3], [
7, 3], [6, 3], [5, 3], [4, 3], [3, 3], [2, 3], [1, 3], [1, 2], [1, 1], [1, 0], [2, 0], [3, 0]] .
```

Figura 4: Resultado da Execução para o Cenário 1.

5.2 Cenário 2

O segundo cenário é representado na Figura 5.

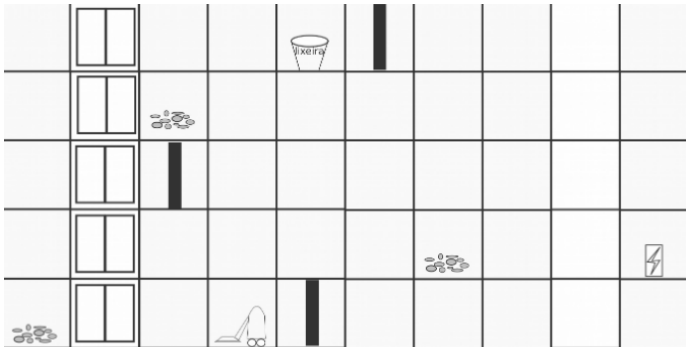

Figura 5: Representação do Cenário 2. Fonte: Material de Apoio, professor Dr. Murilo Naldi Coelho.

As definições do ambiente na base do conhecimento são:

```
% Tamanho Cenario
cenario(10,5).
% Paredes
parede(4,0).
parede(2,2).
parede(4,2).
parede(5,4).
```

```
% Elevadores
elevador(1,0).
elevador(1,1).
elevador(1,2).
elevador(1,3).
elevador(1,4).
% Dock Station
dockStation(9,1).
% Lixeira
lixeira(4,4).
% Sujeiras
sujeiras([[0,0],[6,1],[2,3]]).
% Capacidade AADP
capacidadeMax(2).
```

O resultado obtido está ilustrado na Figura 6.

```
?- solucao([3,0],Solucao).
Solucao = [[3, 0], [2, 0], [1, 0], [0, 0], [0, 0], [1, 0], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1], [6, 1]
, [6, 1], [5, 1], [4, 1], [3, 1], [2, 1], [1, 1], [1, 2], [1, 3], [1, 4], [2, 4], [3, 4], [4, 4], [4, 4],
 [3, 4], [2, 4], [1, 4], [1, 3], [2, 3], [2, 3], [1, 3], [1, 2], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1],
[6, 1], [7, 1], [8, 1], [9, 1]] .
```

Figura 6: Resultado da Execução para o Cenário 2.

5.3 Cenário 3

O segundo cenário é representado na Figura 7.

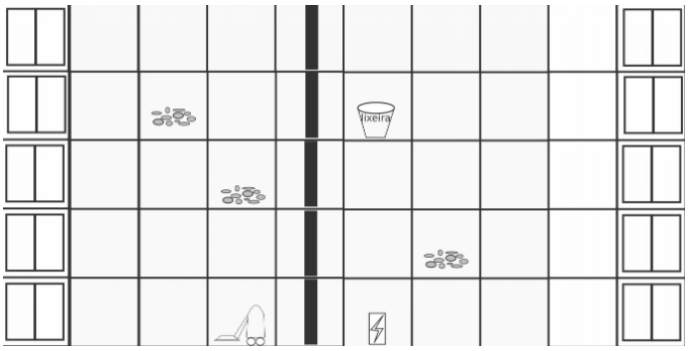

Figura 7: Representação do Cenário 3. Fonte: Material de Apoio, professor Dr. Murilo Naldi Coelho.

As definições do ambiente na base do conhecimento são:

```
% Tamanho Cenario
cenario(10,5).
% Paredes
parede(4,0).
parede(4,1).
```

```
parede(4,2).
parede(4,3).
parede(4,4).
% Elevadores
elevador(0,0).
elevador(0,1).
elevador(0,2).
elevador(0,3).
elevador(0,4).
elevador(9,0).
elevador(9,1).
elevador(9,2).
elevador(9,3).
elevador(9,4).
% Dock Station
dockStation(5,0).
% Lixeira
lixeira(5,3).
% Sujeiras
sujeiras([[6,1],[3,2],[2,3]]).
% Capacidade AADP
capacidadeMax(2).
```

O resultado obtido está ilustrado na Figura 8.

```
?- ['aspirador_ambiente3.pl'].
true.

?- solucao([3,0],Solucao).
false.
```

Figura 8: Resultado da Execução para o Cenário 3.

A Código - Cenário 1

```
% Alunos:
%       Alisson Hayasi da Costa (RA: 726494)
%       Bruna Zamith Santos (RA: 628093)

% ------------ Definicao de Ambiente
% Tamanho Cenario
cenario(10,5).

% Paredes
parede(4,0).
parede(7,1).
parede(4,2).
parede(5,4).

% Elevadores
elevador(1,0).
elevador(1,1).
elevador(1,2).
elevador(1,3).
elevador(1,4).
elevador(8,0).
elevador(8,1).
elevador(8,2).
elevador(8,3).
elevador(8,4).

% Dock Station
dockStation(3,0).

% Lixeira
lixeira(4,4).

% Sujeiras
sujeiras([[2,3],[6,1],[6,4]]).

% Capacidade AADP
capacidadeMax(2).

% ------------ Definicao das Regras Gerais
% pertence(Entrada1,Entrada2)
% Verifica se elemento pertence a lista
% Entrada1: Elem
% Entrada2: Lista
pertence(Elem,[Elem|_]).
pertence(Elem,[_|Cauda]) :- pertence(Elem,Cauda).
```

```
% concatena(Entrada1,Entrada2,Saida)
% Concatena duas listas
% Entrada1: Lista 1
% Entrada2: Lista 2
% Saida: Concatenacao das entradas
concatena([X|Y],Z,[X|W]) :- concatena(Y,Z,W).
concatena([],X,X).

% pop(Entrada,Saida1,Saida2)
% Faz um pop na lista
% Entrada: Lista
% Saida1: Primeiro elemento da lista
% Saida2: Lista sem o elemento
pop([Elem|Cauda],Elem,Cauda).

% reverse(Entrada,Saida)
% Inverte elementos na lista
% Entrada: Lista
% Saida: Lista invertida
reverse([],[]).
reverse([Cab|Cauda],R) :- reverse(Cauda,RCauda), concatena(RCauda,[Cab],R).

% ------------ Definicao das Regras de Movimentacao
% moveDir(Entrada1,Entrada2,Saida1,Saida2)
% Move para direita
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveDir(Xin,Y,Xout,Y) :-
    cenario(Lim,_),
    LimDir is Lim -1, % Eixo comeca no 0
    Xin < LimDir,
    X is Xin + 1,
    not(parede(X,Y)),
    Xout is X.

% sobeElev(Entrada1,Entrada2,Saida1,Saida2)
% Sobe pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
sobeElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
    cenario(_,Lim),
```

```
    LimSup is Lim - 1, % Eixo comeca no 0
    Yin < LimSup,
    Yout is Yin + 1.

% moveEsq(Entrada1,Entrada2,Saida1,Saida2)
% Move para esquerda
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveEsq(Xin,Y,Xout,Y) :-
    Xin > 0,
    X is Xin - 1,
    not(parede(X,Y)),
    Xout is X.

% desceElev(Entrada1,Entrada2,Saida1,Saida2)
% Desce pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
desceElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
    Yin > 0,
    Yout is Yin - 1.

% ------------ Definicao das Regras de Transicao de Estado
% s(Entrada,Saida)
% Transicao de estado
% Entrada: Estado inicial
% Saida: Estado final (sucessor)
s([X,Y],[X1,Y]) :-
    moveDir(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    sobeElev(X,Y,X,Y1).
s([X,Y],[X1,Y]) :-
    moveEsq(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    desceElev(X,Y,X,Y1).

% ------------ Definicao das Regras para Buscas
% meta(Entrada1,Entrada2)
% Checa se o estado atual eh o estado meta
% Entrada1: Estado atual
% Entrada2: Estado meta
meta(Estado,Meta) :- Meta == Estado.
```

```
% busca(Entrada1,Entrada2,Entrada3,Saida)
% Busca em Profundidade
% Entrada1: Caminho inicial
% Entrada2: Estado atual
% Entrada3: Estado meta
% Saida: Caminho da solucao
% Fonte: Material de Apoio, professor Dr. Murilo Coelho Naldi
busca(Caminho,Estado,Final,[Estado|Caminho]) :- meta(Estado,Final).
busca(Caminho,Estado,Final,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    busca([Estado|Caminho],Sucessor,Final,Solucao).

% solucao(Entrada,Saida)
% Recebe estado inicial e retorna o caminho de solucao
% Entrada: Posicao inicial do AADP
% Saida: Caminho percorrido
solucao(Inicial,Solucao) :- sujeiras(Sujeiras),limpa(Inicial,0,Sujeiras,[],Solucao).

% limpa(Entrada1,Entrada2,Entrada3,Entrada4,Saida)
% Define regras para limpeza
% Entrada1: Estado atual
% Entrada2: Capacidade do AADP
% Entrada3: Lista de sujeiras a serem encontradas
% Entrada4: Caminho atual
% Saida: Caminho final

% Quando nao tiver mais sujeira,vai para Dock Station
limpa(Estado,_,[],Caminho,SolucaoFinal) :-
    dockStation(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,SolucaoSaida),
    reverse(SolucaoSaida,SolucaoFinal).
% Busca sujeira se capacidade menor que capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade < CapacidadeMax,
    pop(Sujeiras,Alvo,NovaSujeiras),
    busca([],Estado,Alvo,Solucao),
    NovaCapacidade is Capacidade + 1,
    concatena(Solucao,Caminho,NovoCaminho),
    limpa(Alvo,NovaCapacidade,NovaSujeiras,NovoCaminho,CaminhoResposta).
% Esvazia na lixeira se capacidade igual a capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade >= CapacidadeMax,
```

```
lixeira(X,Y),
busca([],Estado,[X,Y],Solucao),
concatena(Solucao,Caminho,NovoCaminho),
limpa([X,Y],0,Sujeiras,NovoCaminho,CaminhoResposta).
```

B Código - Cenário 2

```
% Alunos:
%   Alisson Hayasi da Costa (RA: 726494)
%   Bruna Zamith Santos (RA: 628093)

% ------------ Definicao de Ambiente
% Tamanho Cenario
% Alunos:
%   Alisson Hayasi da Costa (RA: 726494)
%   Bruna Zamith Santos (RA: 628093)

% ------------ Definicao de Ambiente
% Tamanho Cenario
cenario(10,5).

% Paredes
parede(4,0).
parede(2,2).
parede(4,2).
parede(5,4).

% Elevadores
elevador(1,0).
elevador(1,1).
elevador(1,2).
elevador(1,3).
elevador(1,4).

% Dock Station
dockStation(9,1).

% Lixeira
lixeira(4,4).

% Sujeiras
sujeiras([[0,0],[6,1],[2,3]]).

% Capacidade AADP
capacidadeMax(2).

% ------------ Definicao das Regras Gerais
% pertence(Entrada1,Entrada2)
% Verifica se elemento pertence a lista
% Entrada1: Elem
% Entrada2: Lista
pertence(Elem,[Elem|_]).
```

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

% concatena(Entrada1,Entrada2,Saida)
% Concatena duas listas
% Entrada1: Lista 1
% Entrada2: Lista 2
% Saida: Concatenacao das entradas
concatena([X|Y],Z,[X|W]) :- concatena(Y,Z,W).
concatena([],X,X).

% pop(Entrada,Saida1,Saida2)
% Faz um pop na lista
% Entrada: Lista
% Saida1: Primeiro elemento da lista
% Saida2: Lista sem o elemento
pop([Elem|Cauda],Elem,Cauda).

% reverse(Entrada,Saida)
% Inverte elementos na lista
% Entrada: Lista
% Saida: Lista invertida
reverse([],[]).
reverse([Cab|Cauda],R) :- reverse(Cauda,RCauda), concatena(RCauda,[Cab],R).

% ------------ Definicao das Regras de Movimentacao
% moveDir(Entrada1,Entrada2,Saida1,Saida2)
% Move para direita
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveDir(Xin,Y,Xout,Y) :-
    cenario(Lim,_),
    LimDir is Lim -1, % Eixo comeca no 0
    Xin < LimDir,
    X is Xin + 1,
    not(parede(X,Y)),
    Xout is X.

% sobeElev(Entrada1,Entrada2,Saida1,Saida2)
% Sobe pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
sobeElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
```

```
    cenario(_,Lim),
    LimSup is Lim - 1, % Eixo comeca no 0
    Yin < LimSup,
    Yout is Yin + 1.

% moveEsq(Entrada1,Entrada2,Saida1,Saida2)
% Move para esquerda
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveEsq(Xin,Y,Xout,Y) :-
    Xin > 0,
    X is Xin - 1,
    not(parede(X,Y)),
    Xout is X.

% desceElev(Entrada1,Entrada2,Saida1,Saida2)
% Desce pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
desceElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
    Yin > 0,
    Yout is Yin - 1.

% ------------ Definicao das Regras de Transicao de Estado
% s(Entrada,Saida)
% Transicao de estado
% Entrada: Estado inicial
% Saida: Estado final (sucessor)
s([X,Y],[X1,Y]) :-
    moveDir(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    sobeElev(X,Y,X,Y1).
s([X,Y],[X1,Y]) :-
    moveEsq(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    desceElev(X,Y,X,Y1).

% ------------ Definicao das Regras para Buscas
% meta(Entrada1,Entrada2)
% Checa se o estado atual eh o estado meta
% Entrada1: Estado atual
% Entrada2: Estado meta
```

```
meta(Estado,Meta) :- Meta == Estado.

% busca(Entrada1,Entrada2,Entrada3,Saida)
% Busca em Profundidade
% Entrada1: Caminho inicial
% Entrada2: Estado atual
% Entrada3: Estado meta
% Saida: Caminho da solucao
% Fonte: Material de Apoio, professor Dr. Murilo Coelho Naldi
busca(Caminho,Estado,Final,[Estado|Caminho]) :- meta(Estado,Final).
busca(Caminho,Estado,Final,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    busca([Estado|Caminho],Sucessor,Final,Solucao).

% solucao(Entrada,Saida)
% Recebe estado inicial e retorna o caminho de solucao
% Entrada: Posicao inicial do AADP
% Saida: Caminho percorrido
solucao(Inicial,Solucao) :- sujeiras(Sujeiras),limpa(Inicial,0,Sujeiras,[],Solucao).

% limpa(Entrada1,Entrada2,Entrada3,Entrada4,Saida)
% Define regras para limpeza
% Entrada1: Estado atual
% Entrada2: Capacidade do AADP
% Entrada3: Lista de sujeiras a serem encontradas
% Entrada4: Caminho atual
% Saida: Caminho final

% Quando nao tiver mais sujeira,vai para Dock Station
limpa(Estado,_,[],Caminho,SolucaoFinal) :-
    dockStation(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,SolucaoSaida),
    reverse(SolucaoSaida,SolucaoFinal).
% Busca sujeira se capacidade menor que capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade < CapacidadeMax,
    pop(Sujeiras,Alvo,NovaSujeiras),
    busca([],Estado,Alvo,Solucao),
    NovaCapacidade is Capacidade + 1,
    concatena(Solucao,Caminho,NovoCaminho),
    limpa(Alvo,NovaCapacidade,NovaSujeiras,NovoCaminho,CaminhoResposta).
% Esvazia na lixeira se capacidade igual a capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
```

```
Capacidade >= CapacidadeMax,
lixeira(X,Y),
busca([],Estado,[X,Y],Solucao),
concatena(Solucao,Caminho,NovoCaminho),
limpa([X,Y],0,Sujeiras,NovoCaminho,CaminhoResposta).
```

C Código - Cenário 3

```
% Alunos:
%   Alisson Hayasi da Costa (RA: 726494)
%   Bruna Zamith Santos (RA: 628093)

% ------------ Definicao de Ambiente
% Tamanho Cenario
cenario(10,5).

% Paredes
parede(4,0).
parede(4,1).
parede(4,2).
parede(4,3).
parede(4,4).

% Elevadores
elevador(0,0).
elevador(0,1).
elevador(0,2).
elevador(0,3).
elevador(0,4).
elevador(9,0).
elevador(9,1).
elevador(9,2).
elevador(9,3).
elevador(9,4).

% Dock Station
dockStation(5,0).

% Lixeira
lixeira(5,3).

% Sujeiras
sujeiras([[6,1],[3,2],[2,3]]).

% Capacidade AADP
capacidadeMax(2).

% ------------ Definicao das Regras Gerais
% pertence(Entrada1,Entrada2)
% Verifica se elemento pertence a lista
% Entrada1: Elem
% Entrada2: Lista
pertence(Elem,[Elem|_]).
```

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

% concatena(Entrada1,Entrada2,Saida)
% Concatena duas listas
% Entrada1: Lista 1
% Entrada2: Lista 2
% Saida: Concatenacao das entradas
concatena([X|Y],Z,[X|W]) :- concatena(Y,Z,W).
concatena([],X,X).

% pop(Entrada,Saida1,Saida2)
% Faz um pop na lista
% Entrada: Lista
% Saida1: Primeiro elemento da lista
% Saida2: Lista sem o elemento
pop([Elem|Cauda],Elem,Cauda).

% reverse(Entrada,Saida)
% Inverte elementos na lista
% Entrada: Lista
% Saida: Lista invertida
reverse([],[]).
reverse([Cab|Cauda],R) :- reverse(Cauda,RCauda), concatena(RCauda,[Cab],R).

% ------------ Definicao das Regras de Movimentacao
% moveDir(Entrada1,Entrada2,Saida1,Saida2)
% Move para direita
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveDir(Xin,Y,Xout,Y) :-
    cenario(Lim,_),
    LimDir is Lim -1, % Eixo comeca no 0
    Xin < LimDir,
    X is Xin + 1,
    not(parede(X,Y)),
    Xout is X.

% sobeElev(Entrada1,Entrada2,Saida1,Saida2)
% Sobe pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
sobeElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
```

```
    cenario(_,Lim),
    LimSup is Lim - 1, % Eixo comeca no 0
    Yin < LimSup,
    Yout is Yin + 1.

% moveEsq(Entrada1,Entrada2,Saida1,Saida2)
% Move para esquerda
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
moveEsq(Xin,Y,Xout,Y) :-
    Xin > 0,
    X is Xin - 1,
    not(parede(X,Y)),
    Xout is X.

% desceElev(Entrada1,Entrada2,Saida1,Saida2)
% Desce pelo elevador
% Entrada1: X inicial
% Entrada2: Y inicial
% Saida1: X final
% Saida2: Y final
desceElev(X,Yin,X,Yout) :-
    elevador(X,Yin),
    Yin > 0,
    Yout is Yin - 1.

% ------------ Definicao das Regras de Transicao de Estado
% s(Entrada,Saida)
% Transicao de estado
% Entrada: Estado inicial
% Saida: Estado final (sucessor)
s([X,Y],[X1,Y]) :-
    moveDir(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    sobeElev(X,Y,X,Y1).
s([X,Y],[X1,Y]) :-
    moveEsq(X,Y,X1,Y).
s([X,Y],[X,Y1]) :-
    desceElev(X,Y,X,Y1).

% ------------ Definicao das Regras para Buscas
% meta(Entrada1,Entrada2)
% Checa se o estado atual eh o estado meta
% Entrada1: Estado atual
% Entrada2: Estado meta
```

```
meta(Estado,Meta) :- Meta == Estado.

% busca(Entrada1,Entrada2,Entrada3,Saida)
% Busca em Profundidade
% Entrada1: Caminho inicial
% Entrada2: Estado atual
% Entrada3: Estado meta
% Saida: Caminho da solucao
% Fonte: Material de Apoio, professor Dr. Murilo Coelho Naldi
busca(Caminho,Estado,Final,[Estado|Caminho]) :- meta(Estado,Final).
busca(Caminho,Estado,Final,Solucao) :-
    s(Estado,Sucessor),
    not(pertence(Sucessor,[Estado|Caminho])),
    busca([Estado|Caminho],Sucessor,Final,Solucao).

% solucao(Entrada,Saida)
% Recebe estado inicial e retorna o caminho de solucao
% Entrada: Posicao inicial do AADP
% Saida: Caminho percorrido
solucao(Inicial,Solucao) :- sujeiras(Sujeiras),limpa(Inicial,0,Sujeiras,[],Solucao).

% limpa(Entrada1,Entrada2,Entrada3,Entrada4,Saida)
% Define regras para limpeza
% Entrada1: Estado atual
% Entrada2: Capacidade do AADP
% Entrada3: Lista de sujeiras a serem encontradas
% Entrada4: Caminho atual
% Saida: Caminho final

% Quando nao tiver mais sujeira,vai para Dock Station
limpa(Estado,_,[],Caminho,SolucaoFinal) :-
    dockStation(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,SolucaoSaida),
    reverse(SolucaoSaida,SolucaoFinal).
% Busca sujeira se capacidade menor que capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
    Capacidade < CapacidadeMax,
    pop(Sujeiras,Alvo,NovaSujeiras),
    busca([],Estado,Alvo,Solucao),
    NovaCapacidade is Capacidade + 1,
    concatena(Solucao,Caminho,NovoCaminho),
    limpa(Alvo,NovaCapacidade,NovaSujeiras,NovoCaminho,CaminhoResposta).
% Esvazia na lixeira se capacidade igual a capacidade maxima
limpa(Estado,Capacidade,Sujeiras,Caminho,CaminhoResposta) :-
    capacidadeMax(CapacidadeMax),
```

```
    Capacidade >= CapacidadeMax,
    lixeira(X,Y),
    busca([],Estado,[X,Y],Solucao),
    concatena(Solucao,Caminho,NovoCaminho),
    limpa([X,Y],0,Sujeiras,NovoCaminho,CaminhoResposta).
```

Referências

[1] Max Bramer. *Logic Programming with Prolog*. Springer Publishing Company, Incorporated, 2nd edition, 2014.