# AADP

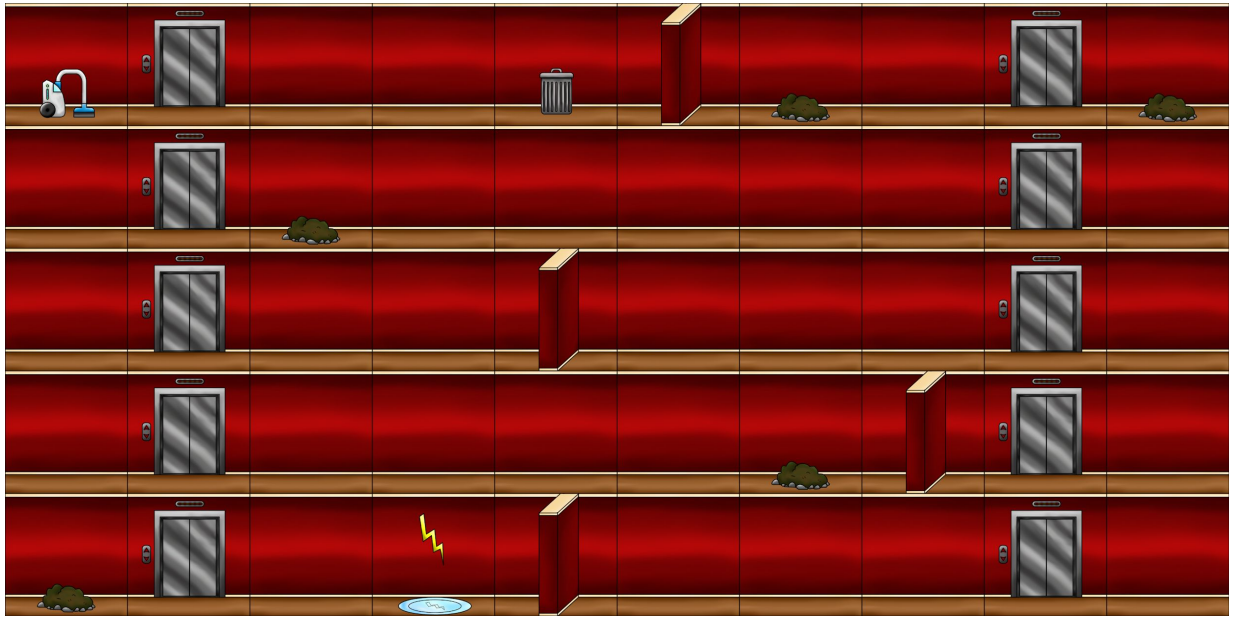
Caio Ueno 743516
Gabriel Cheban 743535
João Augusto Leite 743551

# Apresentação do Cenário

Foram considerados 6 cenários que descrevem possíveis situações para o AADP. A localização de cada objeto é definida como um fato na base de conhecimento. Os possíveis objetos são: paredes, elevadores, sujeiras, lixeira e dockstation.

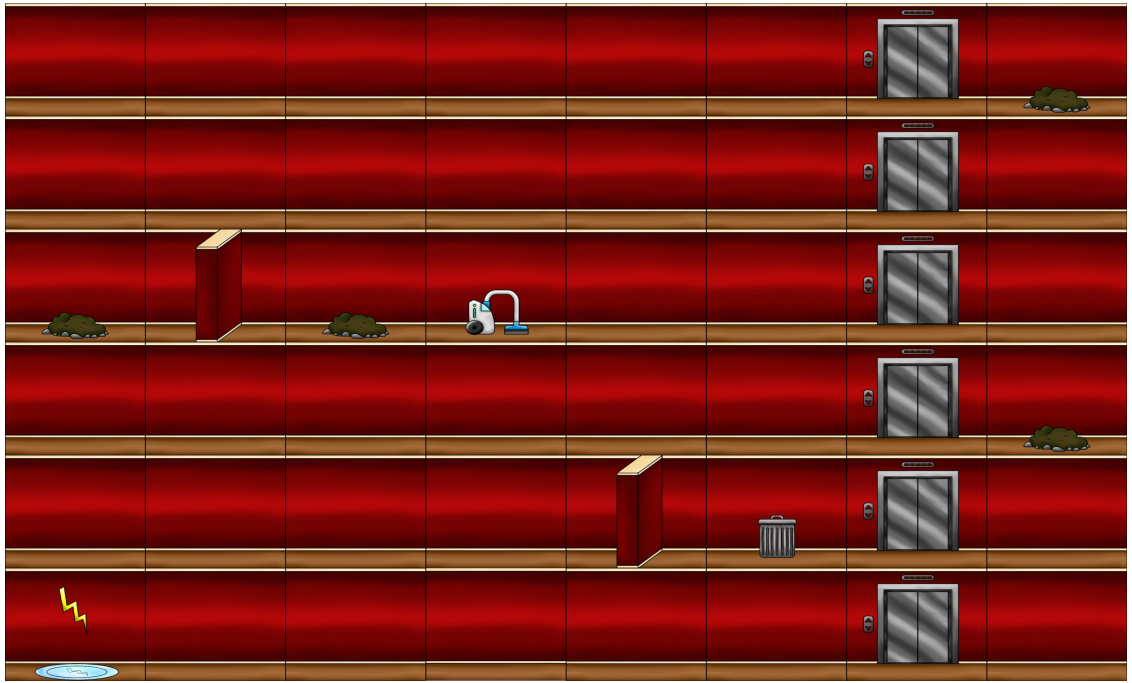
```
objeto([6,5], sujeira).
objeto([6,2], lixeira).
objeto([3,1], dockstation).
objeto([5,1], parede).
objeto([2,1], elevador).
```

# Cenário 1



Todos os objetos acessíveis (referência: especificação).

# Cenário 2

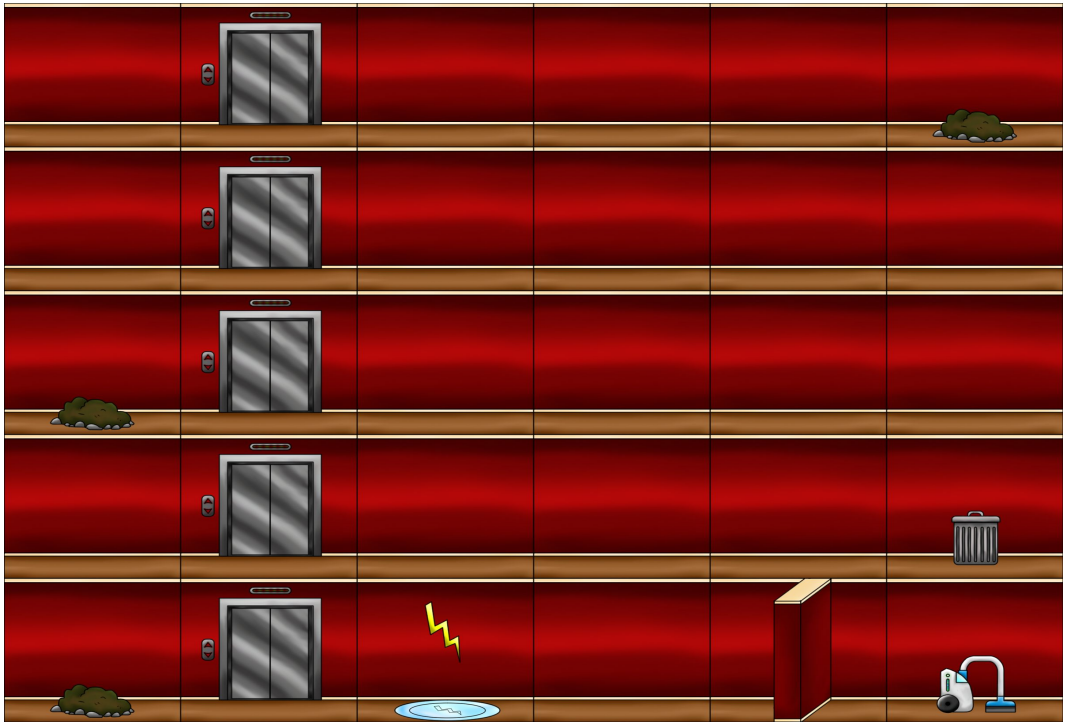


Parede bloqueando uma sujeira.

# Cenário 3

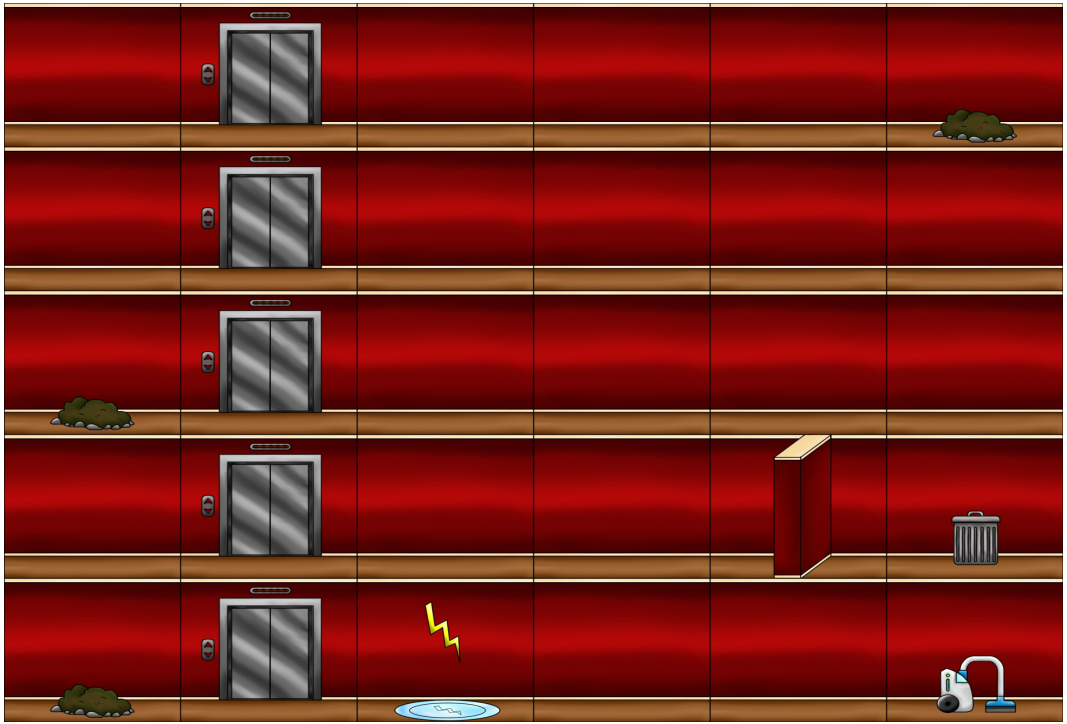
Sujeira em andar sem acesso por elevador.

# Cenário 4



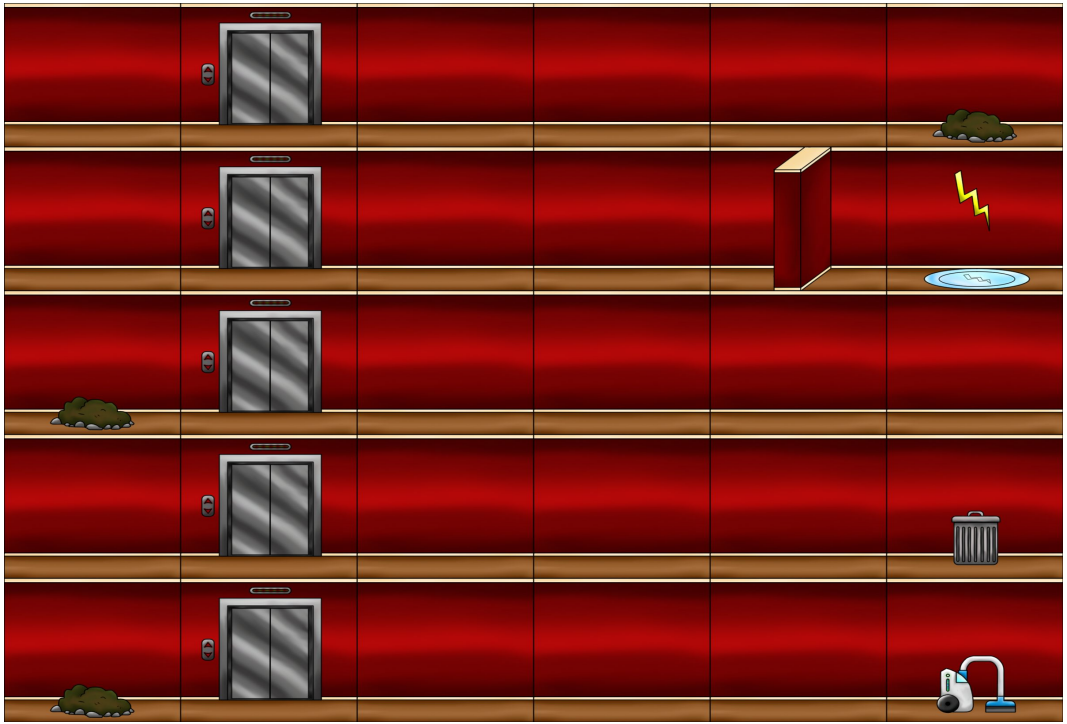
AADP bloqueado por parede.

# Cenário 5



Lixeira bloqueada por parede.

# Cenário 6



*Dockstation* bloqueada por parede.

# Listas

- **Pertence**(Elemento, Lista) [1]
- **Concatena**(Lista1, Lista2, ListasConcatenadas) [1]
- **Inverter**(Lista, ListaReversa) [2]

[1] Material de apoio do Prof. Dr. Murilo Naldi.
[2] A cartilha do Prolog.

# Utilitários

- **Verifica**: Checa se a coordenada possui um objeto, ou qual objeto está naquela coordenada.
- **Adjacente**: “Conecta” uma coordenada com suas adjacentes, as laterais, e caso for um elevador a de cima e a de baixo, respeitando os limites.
- **Invalida**: Verifica se se dada coordenada é inválida.
- **removerDuplicata**[3]: remoção de elementos duplicados adjacentes.

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

**Movimento:** encontra os adjacentes de uma dada posição, verificando a validade da posição.

```
% 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).
```

- **buscaProfundidade**: Realiza uma busca por um Objeto desde uma posição inicial - Estado - até a posição do Objeto, retornando uma Rota - lista de coordenadas navegadas.
- **encontraObjeto**: Predicado que age como uma interface para deixar a busca mais inteligível.

- **procuraLixo**(Caso Base): se não existe mais sujeiras alcançáveis ou a capacidade máxima foi atingida.
- **procuraLixo**(Caso Recursivo): incrementa a capacidade, verifica, caso seja possível, procura uma sujeira, retira a sujeira dos fatos e chama recursivamente ela mesma.

```
% 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).
```

- **limpaPredio**(Caso Base): não há sujeiras alcançáveis.
- **limpaPredio**(Caso Recursivo): procura lixos e uma lixeira. Concatena e chama recursivamente

- **testaMapa**: verifica se uma determinada configuração de cenário é possível.
- **executa**(Caso Base 1): chama testaMapa.
- **executa**(Caso Base 2): verifica se o prédio já está limpo, se sim vai para o  dockstation e termina.

```
% 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, !.
```

- **executa**(Caso Recursivo): Chama o limpa prédio, encontra o caminho até o dockstation, concatena os caminhos, inverte o resultado e remove duplicatas adjacentes.
- **resolvePredio**: Predicado invocado para resolução do problema.

```
% 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), !.
```

# Referências


- http://www.dei.isep.ipp.pt/~jtavares/ALGAV/downloads/ALGAV_TP_aula4.pdf;
- Apostila 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;
- A Cartilha Prolog - Maria do Carmo Nicoletti;
- Material de apoio cedido pelo professor Dr. Murilo Naldi.

# Perguntas?