

# Trabalho 1 - Busy Police, Inteligência Artificial

Juliana K. Sousa 594997<sup>1</sup>, Leila A. Silva 628166<sup>1</sup>, Nicolas Prates 628212<sup>1</sup>

<sup>1</sup>Departamento de Computação - Universidade Federal de São Carlos (UFSCar)  
Caixa Postal 676 - 13565-905 - São Carlos - SP - Brasil

**Resumo.** *Este artigo descreve o desenvolvimento do projeto Busy Police com Prolog, na disciplina Inteligência Artificial ministrada pelo professor Murilo Naldi, no segundo semestre de 2018.*

## 1. Introdução

O jogo Busy Police foi lançado em 1983 para Atari. Nele o jogador controla um policial que deve recuperar os objetos que foram roubados pelo ladrão e, em seguida, prendê-lo. Há um timer com contagem regressiva, objetos que ajudam o policial a se locomover pelos andares (como escada e elevador) e objetos que atrapalham a movimentação (como carrinho e bola), que causam dano ao atingirem o jogador.



Figura 1. Imagem do jogo Busy Police para Atari

2. Descrição do Trabalho

Neste projeto, consideram-se cenários estáticos do jogo, em que o ladrão e os carrinhos ficam parados na mesma posição, não há objetos roubados a serem recuperados ou obstáculos como a bola. Foram fornecidos pelo professor alguns cenários de exemplo, como o da Figura 2.

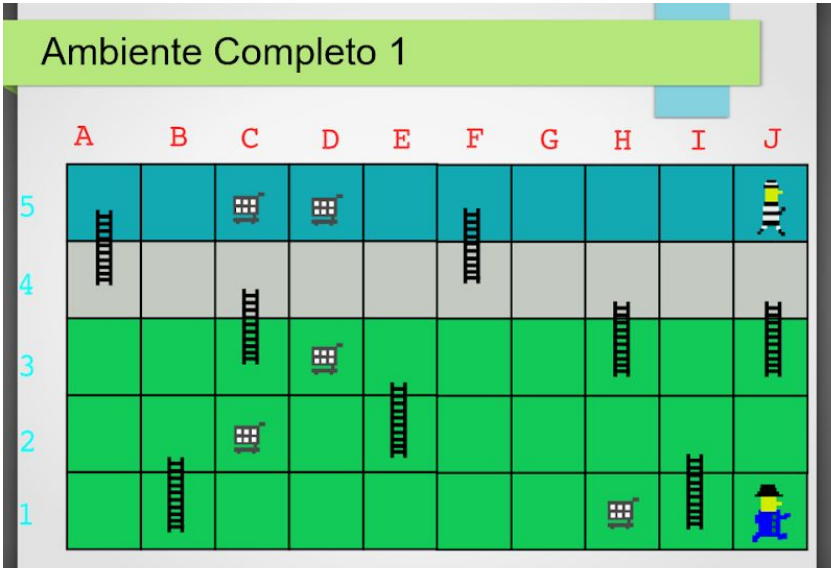


Figura 2. Ambiente Completo 1, fornecido pelo professor

O trabalho consiste em um problema de busca que mostra o caminho percorrido pelo policial desde uma posição inicial até a posição do ladrão.

O ambiente é modelado conceitualmente como uma matriz de posições 5x10 quadradas.

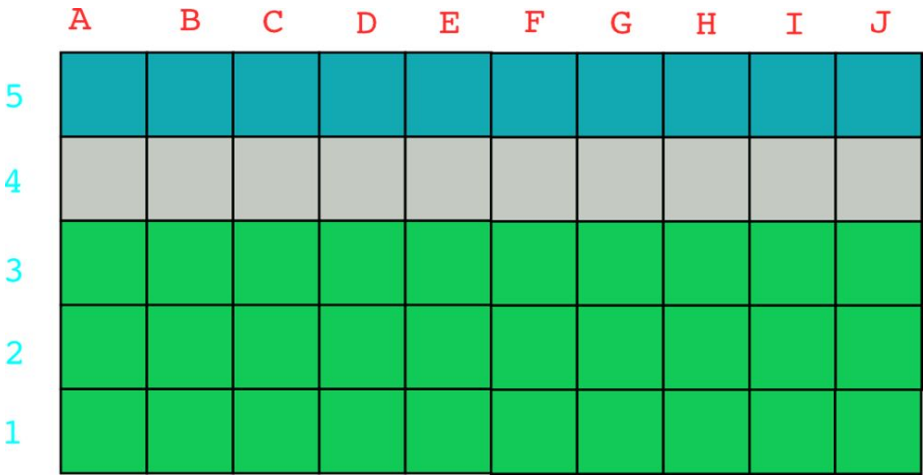


Figura 3. Representação conceitual da matriz de posições

O problema possui dois agentes: o policial, que define a posição inicial do problema de busca, podendo estar em qualquer posição do ambiente (porém preferencialmente no primeiro andar); e o ladrão, que define a posição final do problema de busca, também podendo estar em qualquer posição do ambiente, mas de preferência no último andar.

A movimentação do policial é livre em um mesmo andar (horizontalmente). A movimentação entre andares (vertical) é possível se houver uma escada, para subir ou descer um andar. O objeto carrinho faz com que o policial tenha que pular, porém só poderá ser pulado se as duas posições adjacentes ao carrinho não possuírem nenhum outro objeto dentro, podendo ser outro carrinho, uma escada, a parede e até mesmo o ladrão.

### 3. Regras e Fatos

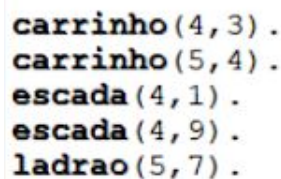
O primeiro objetivo do trabalho é implementar os objetos policial, ladrão, escadas e carrinhos de compra. Decidiu-se por representar os objetos por meio de sua posição, com base no ambiente definido anteriormente. Assim, cada objeto é definido no seguinte formato, sendo x a sua coordenada horizontal e y a coordenada vertical (andar).



```
objeto(x, y) .
```

Figura 4. Formato das regras para objetos

Dessa forma, as posições são programáveis, pois os objetos podem estar em qualquer posição da tabela. Alguns exemplos:



```
carrinho(4, 3) .  
carrinho(5, 4) .  
escada(4, 1) .  
escada(4, 9) .  
ladrao(5, 7) .
```

Figura 5. Exemplos de regras de objetos

O segundo objetivo do trabalho é a implementação da busca em si, que consiste em, dado um estado inicial (policial) e um estado final (ladrão), obter e imprimir um caminho entre os dois, sem ciclos, de forma que o estado inicial seja escolhido pelo usuário. A solução deve ser capaz de rodar a busca em qualquer cenário, incluindo cenários em que não há caminho até o ladrão.

O terceiro objetivo é a implementação de uma característica especial do grupo. Foi escolhido criar um vortex de teleporte, que manda o policial para uma posição aleatória do cenário assim que ele entra na posição ocupada pelo vortex. Caso a posição gerada já esteja ocupada, faz-se três tentativas e, se mesmo assim não for possível posicionar o

policial em uma posição livre, tenta-se as quatro extremidades do cenário (inferior esquerdo, superior esquerdo, inferior direito e, por fim, superior direito).

#### 4. Descrição do programa elaborado

No início do programa são definidos os objetos que fazem parte do ambiente, por exemplo, do ambiente 3 (com adição do vortex):

```
% Ambiente 3
%escada representada pela parte de baixo
escada(9,1) .
escada(1,2) .
escada(10,3) .
escada(5,4) .
carrinho(3,2) .
carrinho(5,2) .
carrinho(7,2) .
carrinho(7,3) .
carrinho(7,5) .
carrinho(8,4) .
vortex(3,3) .
ladrao(1,5) .
```

Figura 6. Objetos do Ambiente 3

Em seguida, a posição do ladrão é definida como meta para a busca.

```
%a meta é a posicao do ladrao
meta([X,Y]) :- ladrao(X,Y) .
```

Figura 7. Regra meta

E a regra ocupado(X,Y) indica se uma posição possui algum objeto.

```
%uma posição esta ocupada se tiver algum objeto
ocupado(X,Y) :- carrinho(X,Y) .
ocupado(X,Y) :- escada(X,Y) .
ocupado(X,Y) :- escada(X,Y2), Y2 is (Y+1) .
ocupado(X,Y) :- ladrao(X,Y) .
ocupado(X,Y) :- vortex(X,Y) .
```

Figura 8. Regra ocupado

Após, são definidas as regras de movimento, que determinam a forma como o policial pode se mover no ambiente, sempre verificando se a posição já foi visitada na busca.

```

%sempre verifica se nao esta no caminho que ja foi visitado
%move se tiver escada
move([X,Y1],[X,Y2], Caminho) :- escada(X,Y1), Y2 is (Y1+1), Y2<6,
    not(pertence([X,Y2],[X,Y1]|Caminho)).

%move para direita se nao for a ultima posicao
move([X1,Y],[X2,Y], Caminho) :- X2 is (X1+1), X2 < 11, not(carrinho(X2,Y)),
    not(pertence([X2,Y],[X1,Y]|Caminho)).

%move para esquerda se nao for a primeira posicao
move([X1,Y],[X2,Y], Caminho) :- X2 is (X1-1), X2 > 0, not(carrinho(X2,Y)),
    not(pertence([X2,Y],[X1,Y]|Caminho)).

%move dois para a direita se tiver um carrinho e nao tiver um objeto depois
move([X1,Y],[X3,Y], Caminho) :- X2 is (X1+1), X3 is (X1+2), X3<11,
    carrinho(X2,Y), not(ocupado(X3,Y)), not(pertence([X2,Y],[X1,Y]|Caminho)).

%move dois para a esquerda se tiver um carrinho e nao tiver um objeto depois
move([X1,Y],[X3,Y],Caminho) :- X2 is (X1-1), X3 is (X1-2), X3>0,
    carrinho(X2,Y), not(ocupado(X3,Y)), not(pertence([X2,Y],[X1,Y]|Caminho)).

%move para baixo se tiver escada
move([X,Y1],[X,Y2],Caminho) :- escada(X,Y2), Y2 is (Y1-1), Y2>0,
    not(pertence([X,Y2],[X,Y1]|Caminho)).

```

Figura 9. Regra move

As regras move também são utilizadas quando o policial entra no vortex. A posição destino é gerada aleatoriamente e, caso haja três tentativas falhas de posicionar o policial em uma posição livre, tenta-se posicioná-lo em um dos quatro cantos do cenário.

```

%move para um lugar aleatório caso encontre um vortex
move([X,Y],[X3,Y3],Caminho) :- vortex(X,Y), random_between(1,10,X3),
    random_between(1,5,Y3), not(ocupado(X3,Y3)),
    not(pertence([X,Y],[X3,Y3]|Caminho)).%tentativa 1
move([X,Y],[X4,Y4],Caminho) :- vortex(X,Y), random_between(1,10,X4),
    random_between(1,5,Y4), not(ocupado(X4,Y4)),
    not(pertence([X,Y],[X4,Y4]|Caminho)).%tentativa 2
move([X,Y],[X5,Y5],Caminho) :- vortex(X,Y), random_between(1,10,X5),
    random_between(1,5,Y5), not(ocupado(X5,Y5)),
    not(pertence([X,Y],[X5,Y5]|Caminho)).%tentativa 3
move([X,Y],[1,1],Caminho) :- vortex(X,Y), not(ocupado(1,1)),
    not(pertence([X,Y],[1,1]|Caminho)). %canto inferior esquerdo
move([X,Y],[1,5],Caminho) :- vortex(X,Y), not(ocupado(1,5)),
    not(pertence([X,Y],[1,5]|Caminho)).%canto superior esquerdo
move([X,Y],[10,1],Caminho) :- vortex(X,Y), not(ocupado(10,1)),
    not(pertence([X,Y],[10,1]|Caminho)).%canto inferior direito
move([X,Y],[10,5],Caminho) :- vortex(X,Y), not(ocupado(10,5)),
    not(pertence([X,Y],[10,5]|Caminho)).%canto superior direito

```

Figura 10. Regra move quando encontra o vortex



Pertence é uma regra auxiliar que define se um objeto pertence a uma lista.

```
pertence(X, [X|_]) .  
pertence(X, [_|L]) :- pertence(X, L) .
```


**Figura 11. Regra pertence**

Por fim, são definidas as regras para a chamada inicial ao rodar o programa, que por sua vez vai para a busca em profundidade (DFS), na qual são utilizadas as regras move definidas anteriormente.

```
%chamada inicial - passa para a dfs  
pega_ladrao(Inicio, Solucao) :- dfs([], Inicio, Solucao).  
  
dfs(Caminho, Estado, [Estado|Caminho]) :- nl, meta(Estado).  
dfs(Caminho, Estado, Solucao) :- write(Estado), move(Estado, Sucessor, Caminho),  
    not(pertence(Sucessor, [Estado|Caminho])), dfs([Estado|Caminho], Sucessor, Solucao).
```

**Figura 21. Regra dfs**

Ao executar o programa, tem-se como saída os estados percorridos, incluindo o retrocesso.

 SWI-Prolog (AMD64, Multi-threaded, version 7.6.4)

File Edit Settings Run Debug Help

Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.

For online help and background, visit <http://www.swi-prolog.org>  
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- pega\_ladrao([3,1],X).

```
[3,1]  
[4,1]  
[5,1]  
[6,1]  
[7,1]  
[8,1]  
[9,1]  
[9,2]  
[10,2]  
[8,2]  
[6,2]  
[4,2]  
[2,2]  
[1,2]  
[1,3]  
[2,3]  
[3,3]  
[6,5]  
[5,5]  
[4,5]  
[3,5]  
[2,5]  
X = [[1, 5], [2, 5], [3, 5], [4, 5], [5, 5], [6, 5], [3, 3], [2|...], [...|...]|...] ;
```

**Figura 12. Saída da execução do programa**

Ilustrando no ambiente, o caminho percorrido pelo policial é:

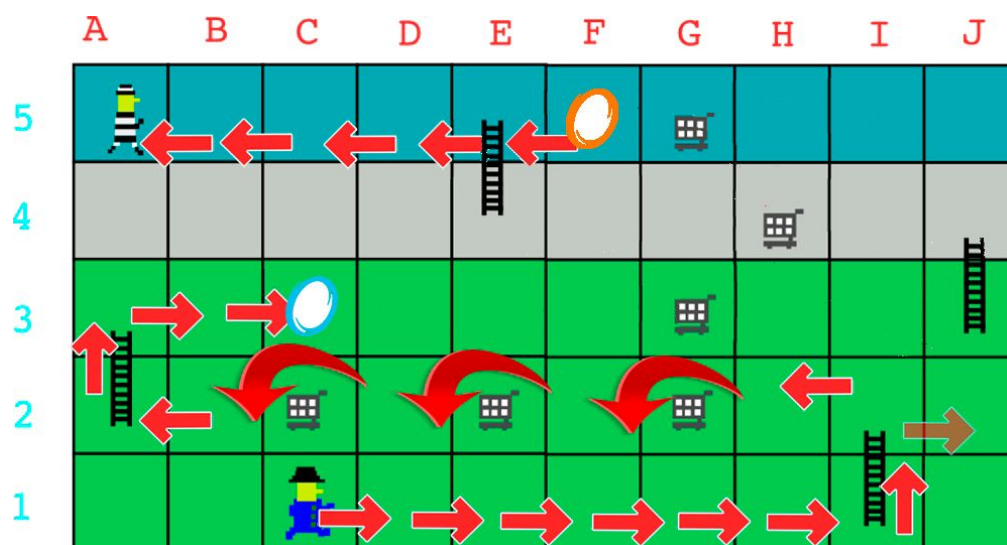


Figura 13. Saída ilustrada do programa

## Referências

Atari 2600, “Keystone Kapers”,  
[http://www.atari2600.com.br/Atari/Roms/01kz/Keystone\\_Kapers](http://www.atari2600.com.br/Atari/Roms/01kz/Keystone_Kapers)

YouTube, “Atari 2600: Busy Police.”,  
[https://www.youtube.com/watch?v=O1k\\_uGfA7rE](https://www.youtube.com/watch?v=O1k_uGfA7rE)