| **Rush Hour**  Documentação |  |  |  |
| --- | --- | --- | --- |

Índice

[Introdução 2](#_Toc164031111)

[Keywords 2](#_Toc164031112)

[História 3](#_Toc164031113)

[Tabuleiro 3](#_Toc164031114)

[A Estrutura Essencial 3](#_Toc164031115)

[Objetivo 4](#_Toc164031116)

[Regras 4](#_Toc164031117)

[Movimentos e orientações 4](#_Toc164031118)

[Estado inicial 4](#_Toc164031119)

[Conclusão do jogo 5](#_Toc164031120)

[Formulação do problema 5](#_Toc164031121)

[Representação de estados 5](#_Toc164031122)

[Representação do tabuleiro 5](#_Toc164031123)

[Representação das viaturas 6](#_Toc164031124)

[Estado Inicial 6](#_Toc164031125)

[Estado Objetivo 7](#_Toc164031126)

[Operações: 7](#_Toc164031127)

[Função de Avaliação 9](#_Toc164031128)

[Heurísticas 9](#_Toc164031129)

[Heurística *blockers* 9](#_Toc164031130)

[Heurística *distance\_to\_goal* 10](#_Toc164031131)

[Heurística *camiões\_estacionados* 10](#_Toc164031132)

[Heurística *available\_to\_blockers* 10](#_Toc164031133)

[Função de Avaliação *(heurística\_final)* 11](#_Toc164031134)

[Métodos de pesquisa (pré-aplicação) 12](#_Toc164031135)

[Depth-First Search (DFS) 12](#_Toc164031136)

[Breadth-First Search (BFS) 12](#_Toc164031137)

[Iterative Deepening Search (IDS) 13](#_Toc164031138)

[Algoritmo A\* 14](#_Toc164031139)

[Bibliografia 14](#_Toc164031140)

# Introdução

O presente relatório tem como objetivo apresentar, analisar e implementar uma inteligência artificial (IA) capaz de resolver o jogo "Rush Hour", de acordo com vários algoritmos já conhecidos, avaliando métricas como o tempo de execução e a quantidade de nós expandidos, no sentido de definir qual o algoritmo mais ótimo para este jogo.

Foram aplicados, em específico, os seguintes algoritmos de pesquisa:

* *Depth-First Search* (DFS)
* *Breadth-First Search* (BFS)
* A\*
* Iterative Deepening Search (IDS)

# Keywords

* Rush Hour
* DFS
* BFS
* A\*
* IDS
* Heurística
* Matrix
* Veículo
* Nós
* Filhos

# História

O jogo *Rush Hour* foi inventado na década de 70 pelas mãos da designer japonesa **Nobuyuki Yoshigahara** sendo popularizado apenas em 1996 nos Estados Unidos. Inspirado pelo congestionamento urbano que presenciava diariamente em Tóquio, Nobuyuki idealizou um jogo que simulasse a experiência de navegar trânsito caótico, desafiando os jogadores a encontrar soluções criativas para escapar do engarrafamento.

# Tabuleiro

## ****A Estrutura Essencial****

* **Grade Quadriculada**: Uma matriz de 6 x 6 espaços, totalizando 36 posições, serve como palco para a movimentação das viaturas.
* **Veículos Obstrutores**: 11 carros e 4 camiões de diferentes cores assumem o papel de obstáculos.
* **Cores**: cada carro/automóvel tem uma cor única de forma a distinguir cada carro.
* **Carros e Camiões:** ocupam um espaço de (largura, altura) equivalente a (2 ,1) e (3,1)
* **Carro Vermelho:** O carro sobre o qual o jogo se desenvolve (1x2).
* **Saída**: presente na coluna quatro é a única parte do tabuleiro que aceita a saída do carro vermelho e sinaliza o objetivo.

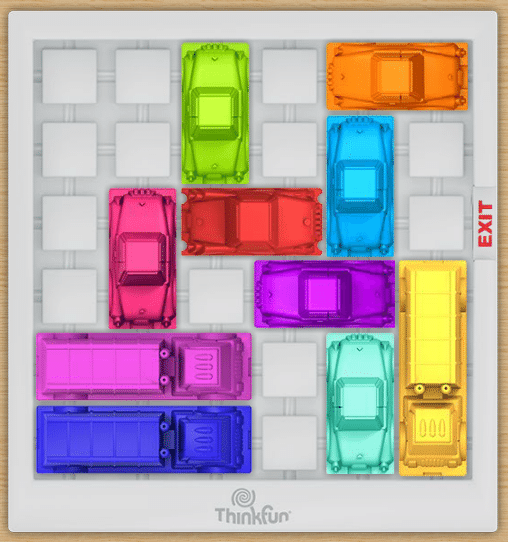


Figura 1-Exemplo de um jogo Rush Hour

# 

# Objetivo

O principal objetivo do *Rush Hour* é desobstruir o caminho para o carro vermelho poder passar pela saída. Para tal, os jogadores devem mover estrategicamente as demais viaturas no tabuleiro, criando um caminho livre para a fuga da viatura vermelha.

# Regras

## Movimentos e orientações

Cada carro no tabuleiro de jogo tem uma orientação, vertical ou horizontal (restritamente) e podem mover-se na sua orientação sem se intercalarem com outro carro ou saírem do tabuleiro, isto é até colidirem com um carro ou com uma borda do tabuleiro. O carro vermelho é o único carro que pode sair da pista.

## Estado inicial

O estado inicial é formado pelo carro vermelho na linha quatro (contando de baixo) orientado na horizontal. É necessário pelo menos um automóvel (horizontal ou vertical) a bloquear o caminho entre o carro vermelho e a saída. É necessário também haver sempre um ou mais lances possíveis.

## Conclusão do jogo

O jogo acaba quando o carro vermelho passar pela saída.

# Formulação do problema

## **Representação de estados**

### Representação do tabuleiro

Em Python, a representação de tabuleiros é frequentemente realizada através de uma estrutura de dados conhecida como "lista de listas". Esta abordagem consiste numa lista principal, onde cada elemento representa uma linha do tabuleiro, e cada elemento dessa lista principal é, por sua vez, uma sublista onde cada elemento dessa sublista representa uma coluna do tabuleiro. Para fins esclarecedores, trataremos a “lista de listas” por matriz e a indexação da mesma começa no seu canto superior esquerdo, onde esta célula representar-se-á pelas coordenadas (0,0). Somar um valor a estas coordenadas implica uma mudança de linha ou coluna.

A escolha por esta estrutura justifica-se pela facilidade de indexação e alteração do estado do tabuleiro em resposta aos movimentos das peças. Através da indexação, é possível aceder e modificar de forma precisa o estado de cada célula do tabuleiro.

Eis uma representação de um tabuleiro vazio:

[[“.”,“.”,“.”,“.”,“.”,“.”],

[“.”, “.”,“.”,“.”,“.”,“.”],

Cada viatura terá a sua única representação

[“.”,“.”,“.”,“.”,“.”,“.”],

[“.”,“.”,“.”,“.”,“.”,“.”],

[“.”,“.”,“.”,“.”,“.”,“.”],

[“.”,“.”,“.”,“.”,“.”,“.”]]

### Representação das viaturas

* “.” - Espaços vazios
* Carro vermelho (2) - tamanho: 2
* Carro rosa (3) – tamanho: 2
* Carro azul claro (4) – tamanho: 2
* Carro laranja (5) – tamanho: 2
* Carro cinzento (6) – tamanho: 2
* Carro verde escuro (7) – tamanho: 2
* Carro preto (8) – tamanho: 2
* Carro amarelo (9) – tamanho: 2
* Carro verde claro (10) – tamanho: 2
* Carro lilás (11) – tamanho: 2
* Carro castanho (12) – tamanho: 2
* Carro verde floresta (13) – tamanho: 2
* Camião azul escuro (14) – tamanho: 3
* Camião lilás (15) – tamanho: 3
* Camião amarelo (16) – tamanho: 3
* Camião azul claro (17) – tamanho: 3

Adicionalmente, de modo a facilitar o desenvolvimento, considera-se a “cabeça” da viatura a posição por ela ocupada mais próxima à origem da matriz, isto é, tabuleiro[0][0]. Assim, para um carro com orientação horizontal, será a posição mais à esquerda, e para um carro posicionado verticalmente, será a sua posição superior.

Estado Inicial

Qualquer estado inicial estará guardado numa biblioteca, sendo este representado pela matriz do tabuleiro, onde em cada célula da matriz estará contido o número de um determinado veículo, categorizado anteriormente pela cor. Todas as células não ocupadas ser-lhes-ão atribuídas a *string* “.”, representando assim um espaço vazio. Eis uma possível representação de um estado inicial:

Formalizando: Tabuleiro: list[list[int]] =

[[ 8 , 8 , “.”, 7 , “.”,“.”],

[ 3 , 3 , “.”, 7 , 16 , 17 ],

[“.”, 2 , 2 , 15, 16 , 17 ],

Representação de um estado inicial

[ 4 , 4 , 6 , 15 , 16 , 17 ],

[ 5 , “.”, 6 , 15 , “.”,“.”],

[ 5 , “.”,“.”, 14 , 14 ,14 ] ]

Estado Objetivo

Num tabuleiro físico, o carro vermelho, representado pelo número 2, teria o seu caminho desobstruído e assim o jogador conseguiria, em direção horizontal, deslizar o carro para fora do tabuleiro. Num ambiente computorizado, o jogo estará resolvido quando os números do carro vermelho se encontrarem no final da linha. Em Python, para que o jogo esteja concluído, os seguintes elementos serão do tipo:

tabuleiro[2][4] == 2 and tabuleiro[2][5] == 2

Assim, uma possível representação de uma matriz de um jogo findado é do tipo:

[[ 8 , 8 , 6 , 7 , “.”,“.”],

[“.”,“.”,“.”,“.”,“.”,“.”],

Representação de um estado objetivo

[“.”,“.”,“.”,“.” “.”, 2 , 2 ],

[ 5 , 4 , 4, 15, 16 , 17 ],

[ 5 , “.”,“.”, 15 , 16 , 17 ],

[14 , 14 , 14 , 15 , 16 , 17]]

### Operações:

As operações são independentes de cada carro ou camião. Ou seja, todos os objetos do tabuleiro podem ser alvos de uma qualquer operação, desde que correspondam às pré-condições das respetivas operações a executar. Considere-se para os exemplos seguintes as variáveis *coluna* e *linha*, ambas de tipo *int,* representando o número da coluna e da linha da “cabeça” da viatura do *tabuleiro*, respetivamente. Vejamos:

#### Mover para cima

##### Pré-condições

* 1 **≤** linha **≤** 5
* 0 **≤** coluna **≤** 5
* Tabuleiro[linha - 1][coluna] : list[list[int]] == “.”
* Orientação : str == ”vertical”

##### Efeitos

* tabuleiro[linha - 1][coluna], tabuleiro[linha + (tamanho - 1)][coluna] =

tabuleiro[linha + (tamanho - 1) ][coluna] , tabuleiro[linha – 1] [coluna]

##### Custo

* Depende dos métodos de pesquisa. Por defeito, custo = 1.

#### Mover para baixo

##### Pré-condições

* 0 **≤** linha **≤** 5 - tamanho
* 0 **≤** coluna **≤** 5
* tabuleiro[linha + tamanho][coluna] : list[list[int]] == “.”
* Orientação : str == “vertical”

##### Efeitos

* tabuleiro[linha + tamanho][coluna], tabuleiro[linha][coluna] =

tabuleiro [linha][coluna], tabuleiro [linha + tamanho][coluna]

##### Custo

* Depende dos métodos de pesquisa. Por defeito, custo = 1.

#### Mover para direita

##### Pré-condições

* 0 **≤** linha **≤** 5
* 0 **≤** coluna **≤** 5 - tamanho
* tabuleiro[linha][coluna + tamanho] : list[list[int]] == “.”
* Orientação : str == “horizontal”

##### Efeitos

* tabuleiro[linha][coluna], tabuleiro[linha][coluna + tamanho] =

tabuleiro [linha][coluna + tamanho], tabuleiro[linha][coluna]

##### Custo

* Depende dos métodos de pesquisa. Por defeito, custo = 1.

#### Mover para esquerda

##### Pré-condições

* 0 **≤** linha **≤** 5
* 1 **≤** coluna **≤** 5
* tabuleiro[linha][coluna-1] : list[list[int]] == “.”
* Orientação : str == “horizontal”

##### Efeitos

* tabuleiro[linha][coluna + (tamanho – 1)], tabuleiro[linha][coluna - 1] =

tabuleiro [linha][coluna], tabuleiro[linha + (tamanho - 1)][coluna - 1]

##### Custo

* Depende dos métodos de pesquisa. Por defeito, custo = 1.

# Função de Avaliação

## Heurísticas

A prática de boas heurísticas é importantíssima para o bom funcionamento do programa que, juntamente com a respetiva função de avaliação, interpretará o jogo. Assim, criamos as seguintes heurísticas com base na priorização da remoção de carros imediatamente à frente do carro vermelho. Vejamos:

### Heurística *blockers*

Uma das heurísticas mais simples que usaremos ao longo do projeto para dar robustez à Função de Avaliação é a heurística *blockers*. Esta tem como objetivo avaliar a quantidade de veículos obstrutores no percurso do carro vermelho, ou seja, quanto maior o número de carros do lado direito do carro vermelho, maior será o valor da heurística *blockers*.

Isto tem como cerne fundamental a avaliação da distância ao estado objetivo que representa, em termos leigos, quanto maior o valor de *blockers* mais distante está o algoritmo de encontrar a solução final.

### Heurística *distance\_to\_goal*

A heurística mais comum, e provavelmente a mais simples, é decerto o que representa a *distance\_to\_goal*. Esta função retorna apenas a numero de células entre o carro vermelho e a saída. Portanto, quanto mais baixo este valor, mais favorecido é o lance.

### Heurística *camiões\_estacionados*

Um dos fatores que contribui para o grau de resolução do problema é, sem dúvida, o correto posicionamento dos camiões colocados à frente do carro vermelho. Isto porque, para além de, para o problema ser solúvel, terem de ser posicionados verticalmente, são os únicos veículos que têm 3 células de tamanho. Tal significa que para um dado camião, existe apenas um estado em que o camião pode estar de forma ao problema estar resolvido: o mais abaixo. Apenas quando um camião está encostado ao limite inferior do tabuleiro (ou seja, quando a sua célula inferior encontra-se em tabuleiro[5]) é que esse camião não constitui um obstáculo ao carro vermelho.

Assim, considerando este facto, podemos assumir que, quanto maior o número de camiões “estacionados” num dado problema, então mais próximo estamos da solução e, como tal, esses lances devem ser priorizados.

### Heurística *available\_to\_blockers*

A heurística *available\_to\_blockers* calcula, essencialmente, a soma do número de lances de todos os *blockers* (carros a direita do vermelho que estejam na sua linha). Este cálculo é feito porque, quanto maior a “liberdade” dos *blockers*, mais facilitada está a solução, visto que mais provável é que um desses lances sejam o necessário para que tal carro deixe de representar um obstáculo ao vermelho.

Logo, quanto maior o número de lances possíveis aos *blockers*, mais priorizado será o lance em questão.

## Função de Avaliação *(heurística\_final)*

A Função de Avaliação é, provavelmente, tanto a mais subjetiva quanto a mais criativa parte do trabalho, pois representa nada mais que o que consideramos ser o método certo para encontrar a solução do problema. Assim, chegamos a uma função multivariável que acreditamos ser a indicada.

Antes de mais, é importante referir que, apesar de termos criado uma multitude de heurísticas, ao experimentar diferentes combinações destas heurísticas com variados coeficientes, acabamos por não utilizar todas elas para a função final, nomeadamente a *available\_to\_blockers*, de modo a reduzir o tempo médio de execução.

Acreditamos que esta heurística não se provou valiosa para a função final provavelmente porque, numa parte considerável dos problemas, um lance que maximize os lances possíveis a determinados carros não é compatível com o lance mais ótimo. Assim, apesar desta heurística poder ser um bom previsor, por exemplo, do grau de dificuldade de um problema, não contribui na sua resolução.

Tendo isto em conta, e de acordo com vários testes para ajuste de coeficientes, atingimos a seguinte fórmula:

**FA = 3 \* *blockers* + *distance\_to\_goal* / 3 + (1 / *camiões\_estacionados*) \* 0.1**

Note-se que, visto que é priorizado o lance com **menor** heurística, multiplicar um termo (por um número superior a 1) é diminuir o seu contributo, ao passo que dividi-lo (ou multiplicá-lo por um número inferior a 1) é aumentar o seu contributo. Vale também ressaltar que a heurística *camiões\_estacionados* é colocada sob 1 pois, ao contrário das demais heurísticas, esta é crescente com a solubilidade do problema. Assim, invertemos o seu resultado de forma a torná-lo consistente com a convenção de minimização da heurística.

# 

# Métodos de pesquisa (pré-aplicação)

## Depth-First Search (DFS)

A pesquisa em questão não possui heurísticas, sendo dependente unicamente da quantidade de lances possíveis. Para investigá-la, optou-se por iniciar a análise a partir das coordenadas (0,0). A análise progride, percorrendo a primeira linha, coluna a coluna, até encontrar uma viatura. Ao concluir a análise de tal linha, a linha seguinte é analisada, e assim sucessivamente.

Ao analisar as coordenadas, caso seja identificada uma viatura, são adicionados os lances possíveis a partir da referida viatura, à árvore (como “filhos” do nó atual), à *queue* e, por fim, à lista de posições visitadas (chamemos-lhe *histórico*). A ordem de priorização dos lances é a seguinte:

Lances verticais

1. Cima
2. Baixo

Lances horizontais

1. Direita
2. Esquerda

(A definir).

Após a finalização da análise, o programa segue a ordem da *queue* e escolhe o primeiro "filho" para análise, evitando posições já visitadas (por comparação com o *histórico*). Cada "filho" é então adicionado à *queue*, após o seu "pai", com prioridade para com os nós já presentes. O processo repete-se até que uma posição atinja o objetivo ou não seja mais possível progredir na árvore.

## Breadth-First Search (BFS)

Este método de pesquisa consiste em analisar todos os lances possíveis num dado estado do tabuleiro, verificando se alguma delas corresponde ao estado objetivo. Assim, o seu tempo de execução depende apenas da quantidade de lances disponíveis.

Consideremos, mais uma vez, que a análise do tabuleiro inicia-se pela origem, (0,0). Segundo a metodologia deste algoritmo, verificamos coluna a coluna a primeira linha, verificando a existência (ou não) de uma viatura. De seguida, avançamos para a linha seguinte do tabuleiro, repetindo o processo até que todas as linhas tenham sido analisadas.

Assim que seja encontrada uma viatura, avalia-se quais são os lances possíveis a ela associada, e, caso não seja um estado objetivo nem um estado já presente no *histórico*, esse lance é adicionado tais lances à árvore (como “filhos” do nó atual), à *queue*, e ao *histórico*.

Quando a análise de um tabuleiro estiver concluída, começa a análise do próximo na *queue*, ou seja, serão analisados todos os tabuleiros resultantes dos lances encontrados no estado anterior (sejam eles o conjunto ***A****),* e os tabuleiros resultantes dos lances encontrados em ***A***,precisamente por esta ordem, e assim sucessivamente.

A pesquisa termina quando se atinge um estado objetivo ou quando a *queue* estiver vazia.

## Iterative Deepening Search (IDS)

O algoritmo IDS é, em muitos aspetos, fruto de uma otimização do DFS. Isto porque, no seu cerne, funcionam exatamente da mesma forma, escolhendo sempre expandir nós com maior profundidade. Diferenciam-se, portanto, num único facto: o IDS está limitado na sua pesquisa em profundidade.

Ao passo que o Depth-First Search expande livre e indiscriminadamente os nós com a maior profundidade, o método IDS têm como parâmetro um inteiro que representa a profundidade máxima que pode alcançar nessa iteração. Tal significa que, quando explora um nó na profundidade máxima, caso não seja o estado objetivo esse nó é considerado o fim do ramo. Na ocasião de nenhum dos nós corresponder ao estado objetivo, o algoritmo falha, incrementando se a profundidade máxima em 1 e tentando de novo, até se descobrir o estado objetivo.

## Algoritmo A\*

O algoritmo A\* o primeiro (e único) método de pesquisa informado implementado neste projeto. Por definição, este algoritmo, após verificar que o nó em que se encontra não é o estado objetivo, expande esse nó, associando aos seus “filhos” um número de acordo com a soma de dois fatores: o custo da operação necessária para atingir o “filho”, e o valor do mesmo segundo uma função de avaliação heurística (neste caso, a Função de Avaliação **FA** previamente mencionada). Assim, sendo a fórmula clássica do A\* ***H = custo + FA***, e tendo em conta que o custo é constante ao longo de todos os operadores, pode-se simplesmente ter que ***H = FA***.

De seguida, tendo um valor associado a cada nó, escolhe aquele em que **H** seja maior ou menor, dependendo da convenção imposta de acordo com o problema, e repetindo este processo até identificar o nó em que se encontra como o estado objetivo. Assim sendo, o algoritmo A\* procura, essencialmente, o caminho mais rápido até ao ponto mais próximo da solução, sendo a sua implementação aconselhava quando o grau de ramificações na árvore é elevado, exatamente por ser um método de pesquisa informado.

# Bibliografia

(s.d.).

Fogleman, M. (s.d.). *Solving Rush Hour, the Puzzle*. Fonte: https://www.michaelfogleman.com/rush/

Liao, W.-C. (s.d.). *Rush Hour Puzzle Search Algorithms*. Fonte: GitHub: https://github.com/LiaoWC/rush\_hour\_puzzle\_AI/blob/main/README.md

ThinkFun. (s.d.). *Rush Hour Deluxe Edition.* Fonte: https://www.thinkfun.com/wp-content/uploads/2017/02/RushHDeluxe-5050-IN.pdf

Tromp, J., & Cilibrasi, R. (s.d.). *Limits of Rush Hour Logic Complexity.* Fonte: Arxiv: https://arxiv.org/pdf/cs/0502068v1.pdf

*Wikipedia*. (s.d.). Fonte: Rush Hour (puzzle): https://en.wikipedia.org/wiki/Rush\_Hour\_(puzzle)

Zarhuber, S., & Pirklbauer, F. (s.d.). *Rush Hour - An A\* Implementation*. Fonte: GitHub: https://github.com/saschazar21/rushhour/blob/master/README.md

Adicionar os seguintes a bibliografia (consultar AI Policy):

* Copilot
* Gemini
* Chat GPT