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

Índice

[Introdução 2](#_Toc162476042)

[História 2](#_Toc162476043)

[Tabuleiro 3](#_Toc162476044)

[A Estrutura Essencial 3](#_Toc162476045)

[Objetivo 3](#_Toc162476046)

[Regras 4](#_Toc162476047)

[Movimentos e orientações 4](#_Toc162476048)

[Estado inicial 4](#_Toc162476049)

[Conclusão do jogo 4](#_Toc162476050)

[Formulação do problema 4](#_Toc162476051)

[Representação de estados 4](#_Toc162476052)

[Representação do tabuleiro 4](#_Toc162476053)

[Representação das viaturas 5](#_Toc162476054)

[Estado Inicial 6](#_Toc162476055)

[Estado Objetivo 6](#_Toc162476056)

[Operações: 7](#_Toc162476057)

[Função de Avaliação 9](#_Toc162476058)

[Heurísticas 9](#_Toc162476059)

[Heurística *moves* 9](#_Toc162476060)

[Heurística *blockers* 10](#_Toc162476061)

[Função de Avaliação (???) 10](#_Toc162476062)

[Constante *media* 11](#_Toc162476063)

[Métodos de pesquisa (pré-aplicação) 11](#_Toc162476064)

[Depth-First Search 11](#_Toc162476065)

[Breadth-First Search 12](#_Toc162476066)

[Pré-avaliações/expectativas 13](#_Toc162476067)

[TRABALHO A DESENVOLVER 14](#_Toc162476068)

# 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\*

# 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. *(seriam 12 carros incluindo o vermelho)*
* **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.

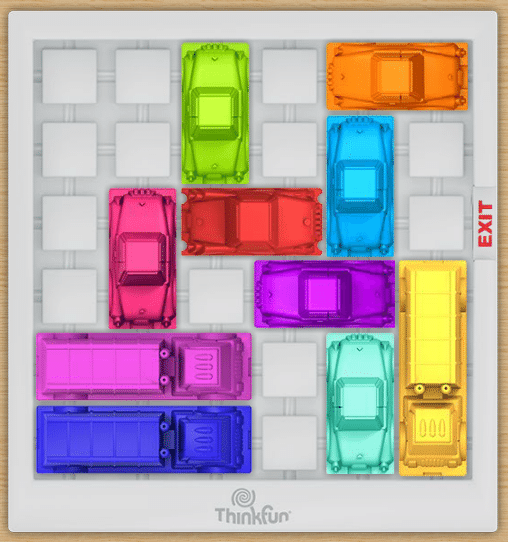


Figura 1-Exemplo de um jogo Rush Hour

# 

# 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:

[[ 0 , 0 , 0 , 0 , 0 , 0 ],

[ 0 , 0 , 0 , 0 , 0 , 0 ],

Cada viatura terá a sua única representação

[ 0 , 0 , 0 , 0 , 0 , 0 ],

[ 0 , 0 , 0 , 0 , 0 , 0 ],

[ 0 , 0 , 0 , 0 , 0 , 0 ],

[ 0 , 0 , 0 , 0 , 0 , 0 ]]

### Representação das viaturas

* 0 - 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 APLICAR FUNÇAO HASH

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 o valor 0, representando assim um espaço vazio. Eis uma possível representação de um estado inicial:

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

[[ 8 , 8 , 0 , 7 , 0 , 0 ],

[ 3 , 3 , 0 , 7 , 16 , 17 ],

[ 0 , 2 , 2 , 15, 16 , 17 ],

Representação de um estado inicial

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

[ 5 , 0 , 6 , 15 , 0 , 0 ],

[ 5 , 0 , 0 , 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 , 0 , 0 ],

[ 3 , 3 , 6 , 7 , 0 , 0 ],

Representação de um estado objetivo

[ 0 , 0 , 0 , 0 , 2 , 2 ],

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

[ 5 , 0 , 0 , 15 , 16 , 17 ],

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

USAR FUNÇAO HASH

### 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]] == 0
* 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]] == 0
* 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]] == 0
* 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]] == 0
* 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 *moves*

Esta heurística medirá o número de lances (em inglês, *moves*) necessários para retirar um determinado carro da frente do vermelho. Atente no seguinte exemplo:

Uma imagem com diagrama, Retângulo, design

Descrição gerada automaticamente

O conjunto de todas as setas, verdes e brancas, representam todos os lances possíveis desta posição. No entanto, como o objetivo da heurística é a remoção de obstáculos entre o carro vermelho e a saída (em Python, o elemento tabuleiro[2][5]) incluso, o conjunto de lances representados pelas setas brancas mantêm o problema do mesmo modo, ou seja, não promovem qualquer remoção de obstáculos. Analisaremos a seguir a remoção de todos os carros imediatamente à direita do vermelho.

A reação em cadeia representada pelas setas azuis, pela ordem proposta, representa com clareza a desobstrução do caminho do carro vermelho segundo o veículo castanho. Assim, segundo a heurística *moves*, o carro castanho tem *moves* = 2.

Do mesmo modo, reação em cadeia representada pelas setas verde, pela ordem proposta, representa também a desobstrução do caminho do carro vermelho segundo o veículo roxo. Assim, segundo a heurística *moves*, o camião roxo tem *moves* = 5.

### Heurística *blockers*

Embora igualmente eficaz, uma heurística mais simples que usaremos ao longo do projeto para dar robustez à Função de Avaliação é a heurística *blockers*. Esta heurística 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.

## Função de Avaliação (???)

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.

Apesar da utilização exclusiva da heurística *blockers* ser aceitável, não é de todo a mais correta. Isto porque, na realidade, o que se quer neste exercício não é minimizar o número de carros à frente do carro vermelho, mas sim minimizar o número de lances necessários para este chegar ao estado objetivo. Tendo isso em conta, torna-se evidente que é crucial a criação de uma função que essencialmente represente a quantidade de lances necessários até à resolução do problema.

Sabemos, no entanto, que deduzir esse número é categoricamente impossível (podendo-se, no máximo, estimar esse número). Como tal, inspiramo-nos nas funções tipicamente utilizadas nos algoritmos A\*. Estas funções resultam da soma de uma heurística *h*, com uma estimativa da proximidade do estado atual ao estado objetivo (seja essa a função *g).* Assim, utilizando as heurísticas *blockers* e *moves*, e considerando os lances necessários como a métrica desta função obtém-se:

***F* = *moves + (blockers – 1) \* media***

Logo, do modo como está construída, a função ***F*** verifica quantos lances são necessários para mover um determinado carro à direita do vermelho, e acrescenta o número de lances necessários, em média, para mover os restantes, resultando, portanto, numa estimativa do número de lances restantes ate ao estado objetivo.

### Constante *media*

A constante *media* alude à média de lances necessários para mover um carro da frente do carro vermelho.

# 

# Métodos de pesquisa (pré-aplicação)

## Depth-First Search

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

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.

### 

### Pré-avaliações/expectativas

Em contraste com jogos mais complexos como o Xadrez, onde o número de posições únicas é consideravelmente extenso, no jogo *Rush Hour*, a repetição de posições na lista de "netos" (filhos dos filhos) durante a busca em árvore é um evento significativamente mais frequente. Essa particularidade decorre de duas principais razões:

**Movimentação do Carro**: O movimento de um carro pode obstruir a realização de um lance específico por um dos seus "filhos".

**Repetição de Posições**: A posição do carro pode se repetir, impedindo que o código a analise novamente, resultando em economia de tempo de processamento.

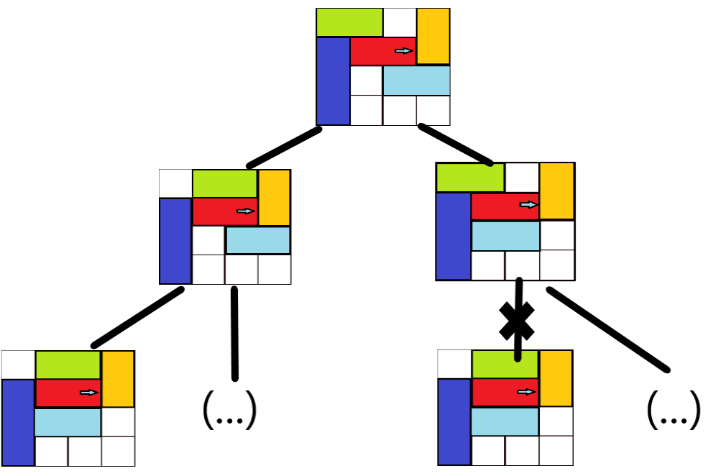


Figura 2- exemplo de repetição em depth 2

# 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

# Referências

(s.d.).

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

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

ThinkFun. (s.d.). *Rush Hour Deluxe Edition.* Obtido de 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.* Obtido de Arxiv: https://arxiv.org/pdf/cs/0502068v1.pdf

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

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