

---

# Busca em Espaço de Estados

Fabrício Barth

Inspere

Março de 2022

---

---

# Introdução

## Agente orientado a **meta**

- O projetista não determina um mapeamento entre percepções e ações, mas determina que objetivo o agente deve alcançar
- É necessário que o próprio agente construa um plano de ações que atinjam seu objetivo  
(como se o próprio agente construísse seu programa)
- Exemplos: o agente aspirador de pó, um agente motorista de táxi, uma sonda espacial, ...

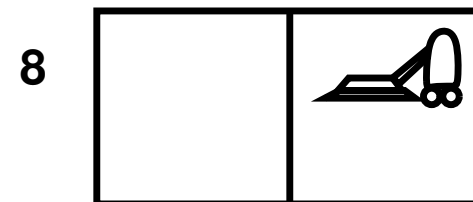
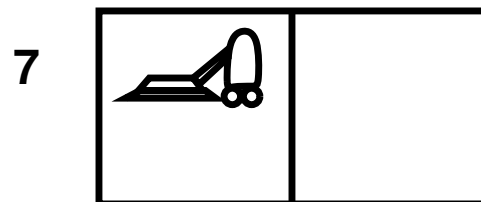
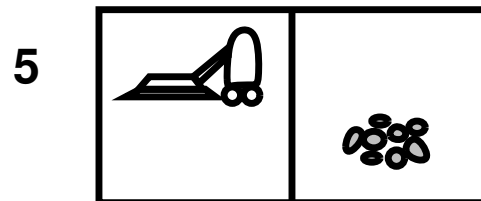
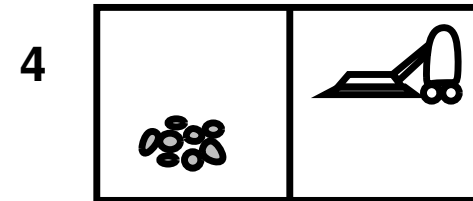
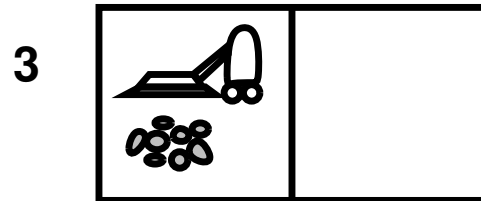
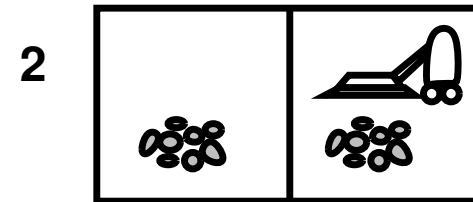
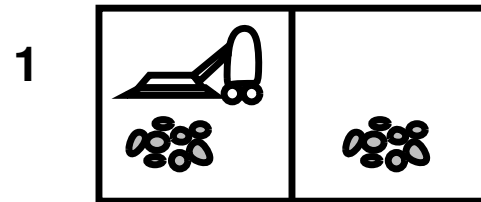
## O que é **busca**?

- O mundo do agente tem um conjunto de **estados** possíveis (muitas vezes este conjunto é infinito)
- Existem **transições** entre os estados do mundo, formando um grafo.
- São utilizados **algoritmos** para encontrar um caminho neste grafo
  - ★ partindo do estado inicial (atual)
  - ★ até o estado objetivo

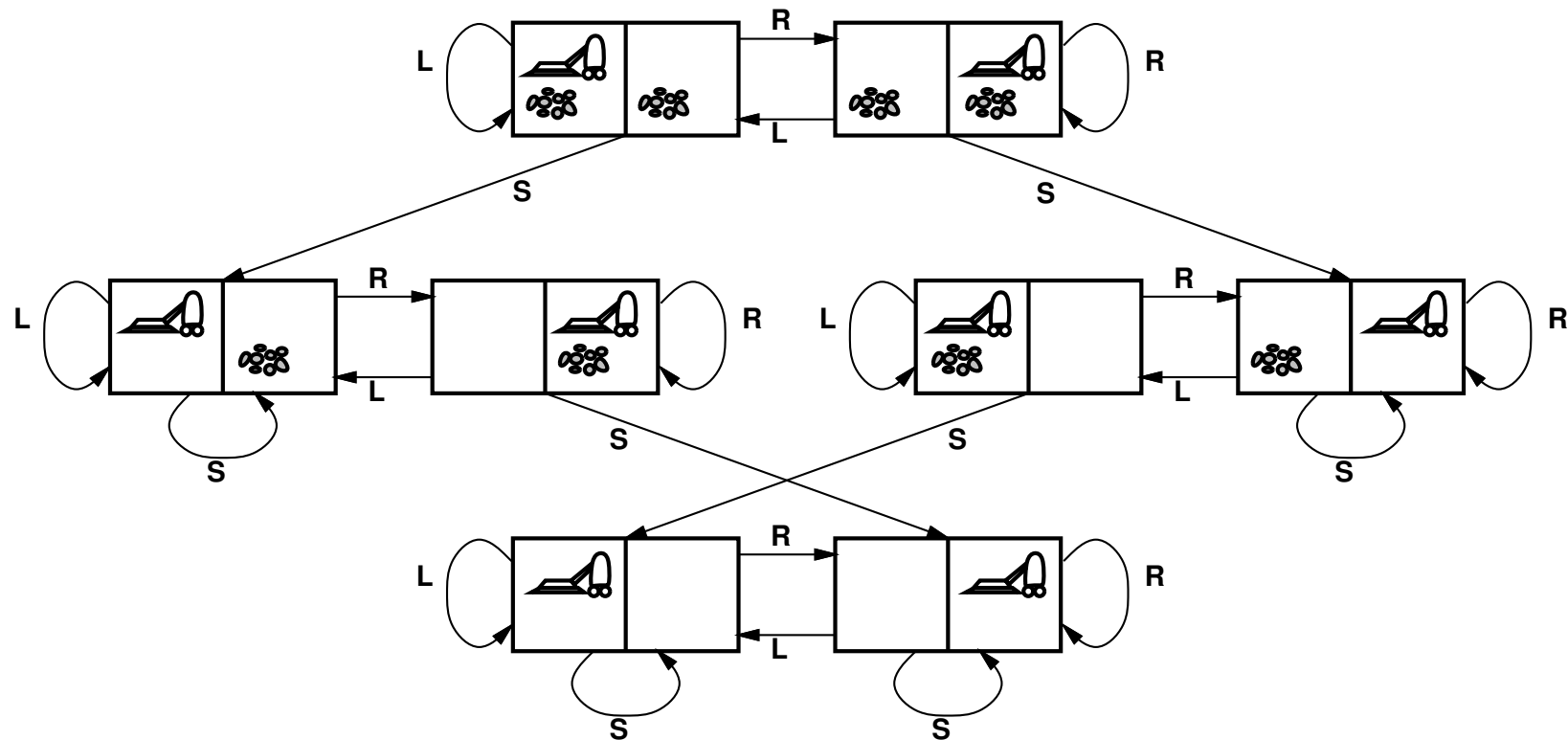
# Exemplo do aspirador de pó

- Um robô aspirador de pó deve limpar uma casa com duas posições. As operações que ele sabe executar são:
  - ★ sugar
  - ★ ir para a posição da esquerda
  - ★ ir para a posição da direita
- Como o aspirador pode montar um plano para limpar a casa se inicialmente ele está na posição direita e as duas posições têm sujeira?
  - ★ Quais os estados possíveis do mundo do aspirador e as transições?

## Estados possíveis:



## Espaço de busca



# Por que **estados**?

- As informações do mundo real são absurdamente complexas, é praticamente impossível modelá-las todas
  - ★ No exemplo do aspirador, o mundo dele tem várias outras informações: a cor do tapete, se é dia, de que material o aspirador é feito, quanto ele tem de energia, como é o nome do/a proprietário/a, ....
- A noção de estado é utilizada para **abstrair** esses detalhes e considerar somente o que é relevante para a solução do problema
- O mesmo se dá com as operações modeladas: são abstrações das operações reais (ir para a posição da direita implica em várias outras operações).



## Exemplo do **homem, o lobo, o carneiro e o cesto de alface.**

- Uma pessoa, um lobo, um carneiro e um cesto de alface estão à beira de um rio. Dispondo de um barco no qual pode carregar apenas um dos outros três, a pessoa deve transportar tudo para a outra margem. Determine uma série de travessias que respeite a seguinte condição: em nenhum momento devem ser deixados juntos e sozinhos o lobo e o carneiro ou o carneiro e o cesto de alface.

# Busca como **desenvolvimento** de software

- No desenvolvimento de um software para resolver um problema, o projetista pode optar por várias paradigmas de modelagem do problema:
  - ★ O sistema é modelado por procedimentos que alteram os dados de entrada
  - ★ O sistema é modelado por funções
  - ★ O sistema é modelado por predicados
  - ★ O sistema é modelado por objetos
  - ★ ...

- Busca é mais uma forma de modelar um problema:
  - ★ Definir os estados
  - ★ Definir as transições
  - ★ Escolher um algoritmo de busca

# Exercício

O que é

- estado
- transição
- estado meta e
- custo da solução encontrada

para os seguintes problemas

- 8-Puzzle

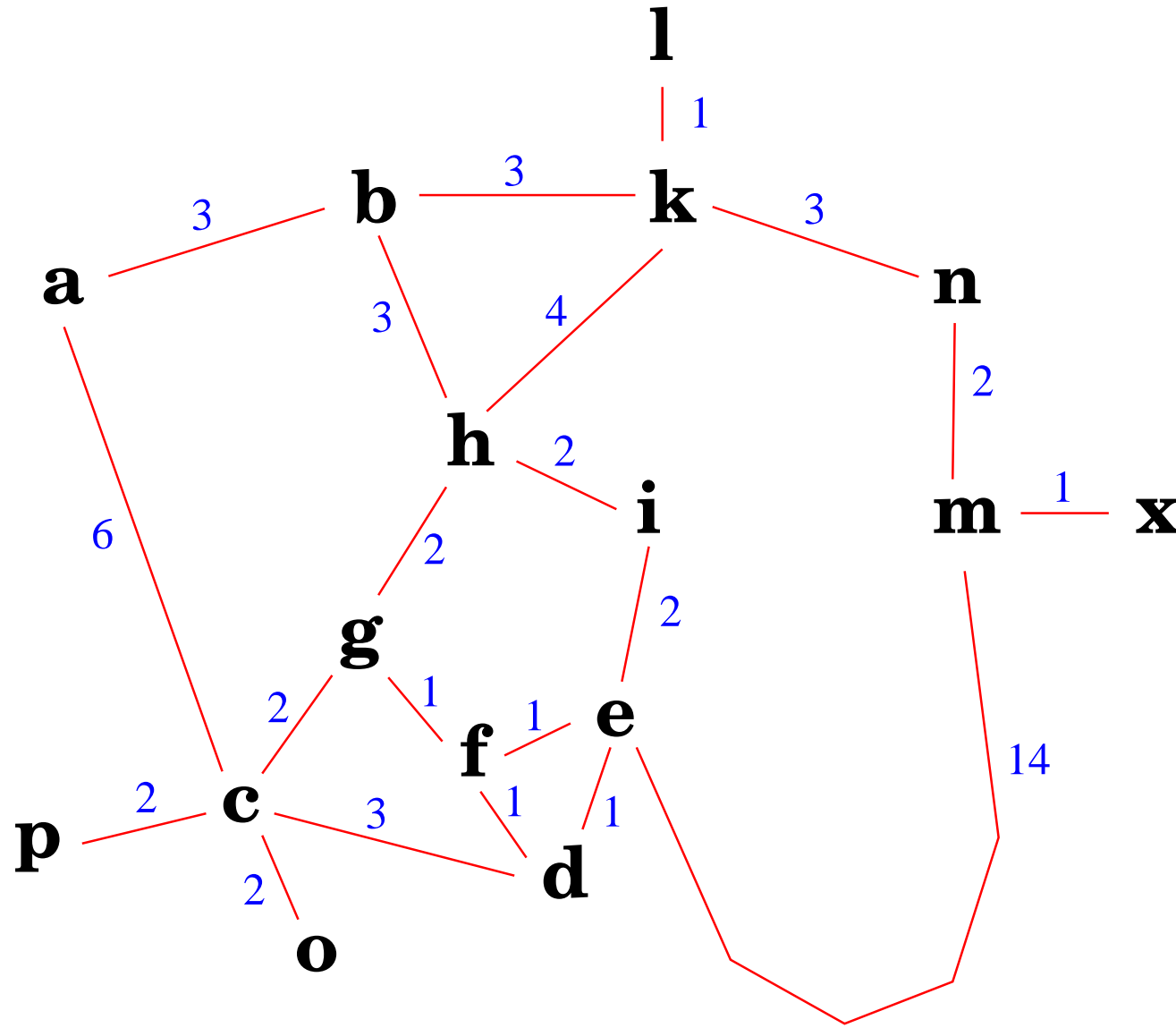
5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

- Encontrar um caminho da cidade “i” até “x”



---

# Algoritmos de Busca

## “Cega”

# Árvore de busca

- Coloca-se o estado inicial como nodo raiz
- Cada operação sobre o estado raiz gera um novo nodo (chamado de **sucessor**)
- Repete-se este processo para os novos nodos até gerar o nodo que representa o estado meta
- **Estratégia** de busca: que nodo escolher para expandir
- Exemplo: (fazer as árvores para o exemplo do aspirador e do homem no rio)



## Estratégias de busca

- Busca em **largura**: o nodo de **menor** profundidade mais a esquerda é escolhido para gerar sucessores
- Busca em **profundidade**: o nodo de **maior** profundidade mais a esquerda é escolhido para gerar sucessores

# Algoritmo de busca em **largura**

```
function BL(Estado inicial): Nodo  
  Fila abertos  
  abertos.add(new Nodo(inicial))  
  while abertos.size() > 0 do  
    Nodo n  $\leftarrow$  abertos.removeFirst()  
    if n.getEstado().éMeta() then  
      return n  
    end if  
    abertos.append(n.sucessores())  
  end while  
  return null
```

# Critérios de comparação entre os algoritmos

- **Completo**: o algoritmo encontra a solução se ela existir
- **Ótimo**: o algoritmo encontra a solução de menor custo
- **Tempo**: quanto tempo o algoritmo leva para encontrar a solução no pior caso
- **Espaço**: quanto de memória o algoritmo ocupa

# Análise do algoritmo BL

- Completo: sim
- Ótimo: sim, para problemas de custo uniforme
- Tempo: explorar  $O(b^d)$  nodos
  - ★  $b$  = fator de ramificação
  - ★  $d$  = profundidade do estado meta
  - ★  $O(b^d) = 1 + b + b^2 + b^3 + \dots + b^d$
- Espaço: guardar  $O(b^d)$  nodos.

## Exemplo de complexidade

Prof.	Nodos	Tempo	Memória
0	1	1ms	100 bytes
2	111	0,1 seg	11 Kbytes
4	11.111	11 seg	1 Mbyte
6	$10^6$	18 min	111 Mbytes
8	$10^8$	31 horas	11 Gbytes
12	$10^{12}$	35 anos	111 Tbytes
14	$10^{14}$	3500 anos	11.111 Tbytes

( $b = 10$ , 1000 nodos por segundo, 100 bytes por nodo)

# Nodos da árvore

- Cada nodo tem
  - ★ o estado que representa
  - ★ o nodo pai
  - ★ o operador que o gerou
  - ★ sua profundidade na árvore de busca
  - ★ o custo de ter sido gerado (denotado por  $g$ )
  - ★ opcionalmente, os nodos sucessores
- (fazer a figura)

## Estratégias de poda da árvore de busca

- Um nodo não gera um sucessor igual a seu pai
- Um nodo não gera um sucessor igual a um de seus ascendentes
- Um nodo não gera um sucessor que já exista na árvore de busca

- Detalhes de implementação:
  - ★ Verificar se um estado já está na árvore pode levar muito tempo
    - \* imagine uma árvore com milhares de estados do jogo de xadrez, cada novo estado deve ser comparado com outros milhares de estados!
  - ★ Ter uma tabela **hash** (que tem tempo ótimo de consulta) para saber se determinado nodo existe na árvore



# Algoritmo de busca em **profundidade**

```
function BP(Estado inicial, int m): Nodo
  Pilha abertos
  abertos.add(new Nodo(inicial))
  while abertos.size() > 0 do
    Nodo n  $\leftarrow$  abertos.removeTopo()
    if n.getEstado().éMeta() then
      return n
    end if
    if n.getProfundidade() < m then
      abertos.insert(n.sucessores())
    end if
  end while
  return null
```

# Análise do algoritmo BP

- Completo: não (caso a meta esteja em profundidade maior que  $m$ )

Se  $m = \infty$ , é completo se o espaço de estados é finito e existe poda para não haver loops entre as operações

- Ótimo: não
- Tempo: explorar  $O(b^m)$  nodos (ruim se  $m$  é muito maior que  $d$ )
- Espaço: guardar  $O(bm)$  nodos. (em profundidade 12, ocupa 12 Kbytes!)

# Algoritmo de busca em profundidade iterativo

```
function BPI(Estado inicial): Nodo  
  int  $p \leftarrow 1$   
  loop  
    Nodo  $n \leftarrow$  BP(inicial,  $p$ )  
    if  $n \neq \text{null}$  then  
      return  $n$   
    end if  
     $p \leftarrow p + 1$   
  end loop
```

## Análise do algoritmo BPI

- Completo: sim
- Ótimo: sim, se todas as ações tem o mesmo custo
- Tempo: explorar  $O(b^d)$  nodos
- Espaço: guardar  $O(bd)$  nodos.

# Algoritmo de busca de **custo uniforme**

```
function BCU(Estado inicial): Nodo
Set abertos ordenados por custo
abertos.add(new Nodo(inicial))
while abertos.size() > 0 do
    Nodo n  $\leftarrow$  abertos.removeFirst()
    if n.getEstado().éMeta() then
        return n
    end if
    abertos.append(n.sucessores())
end while
return null
```

## Algoritmo de busca de **custo uniforme**

- Expande nós de acordo com o custo.
- Se  $\text{custo} = \text{profundidade do nó}$  então temos uma busca em largura.

## Resumo

	BL	BP	BPI	BCU
<b>Completo</b>	sim	não	sim	sim
<b>Ótimo</b>	sim	não	sim	sim
<b>Tempo</b>	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^d)$
<b>Espaço</b>	$O(b^d)$	$O(bm)$	$O(bd)$	$O(b^d)$

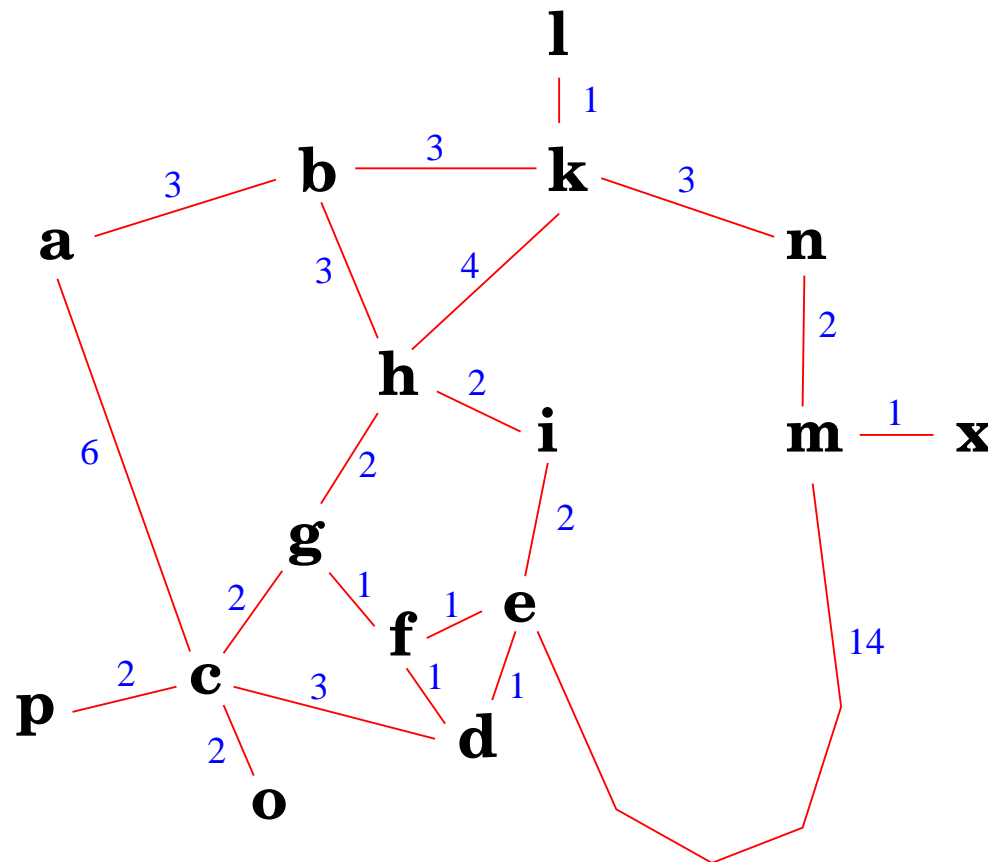
---

# Algoritmos de Busca

## **“Inteligente”**



## Exemplo: ir de “h” para “o” (com BL)



A árvore de busca gerada é “inteligente”?

# Heurística

- Heurística: **Estimativa** de custo até a meta.  
(denotado pela função  $h : Estados \rightarrow Reais$ )
- No exemplo das cidades, poderia ser a distância em linha reta.
- Algoritmo de **busca gananciosa**: retira de abertos sempre o nodo com menor estimativa de custo (menor  $h$ ).
  - ★ Refazer a busca de um caminho entre “h” e “o”.
  - ★ Refazer a busca de um caminho entre “i” e “x”.
  - ★ **Fazer a tabela de  $h$  para os dois casos.**

- ★ Refazer a busca de um caminho entre “h” e “o”.  
**ótimo!**
- ★ Refazer a busca de um caminho entre “i” e “x”.  
**não ótimo!**

# Busca $A^*$

- **Idéia:** Evitar expandir caminhos que **já** estão muito caros mas também considerar os que têm menor expectativa de custo.
- Utilizar na escolha de um nodo da lista de abertos
  - ★ tanto a estimativa de custo de um nodo ( $h(n)$ )
  - ★ quanto o custo acumulado para chegar no nodo ( $g(n)$ )
$$f(n) = g(n) + h(n)$$
- Refazer a busca de um caminho entre “i” e “x” utilizando  $f$ .

## Algoritmo de busca A\*

```
function BA*(Estado inicial): Nodo
  PriorityList(f) abertos {lista ordenada por f}
  abertos.add(new Nodo(inicial))
  while abertos.size() > 0 do
    Nodo n ← abertos.removeFirst()
    if n.getEstado().éMeta() then
      return n
    end if
    abertos.append(n.sucessores())
  end while
  return null
```

## Propriedades da função $h$

- Supondo que o valor de  $h$ , no exemplo das cidades, é dado por  $10 \times$  a distância em linha reta.
- O algoritmo  $A^*$  ainda é ótimo?
- $h(n)$ : estimativa de custo de  $n$  até a meta
- $h^*(n)$ : custo real de  $n$  até a meta
- Se  $h(n) \leq h^*(n)$ , então  $h$  é **admissível**.
- Se  $h$  é admissível, o algoritmo  $A^*$  é ótimo!

## Análise do algoritmo A\*

- Completo: **sim**
- Ótimo: **sim** (se  $h$  é admissível)
- Tempo: explorar  $O(b^d)$  nodos no pior caso (quando a heurística é “do contra”)
- Espaço: guardar  $O(b^d)$  nodos no pior caso.

## Exercício

- Determine uma heurística para o problema 8-Puzzle e verifique se é admissível.

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

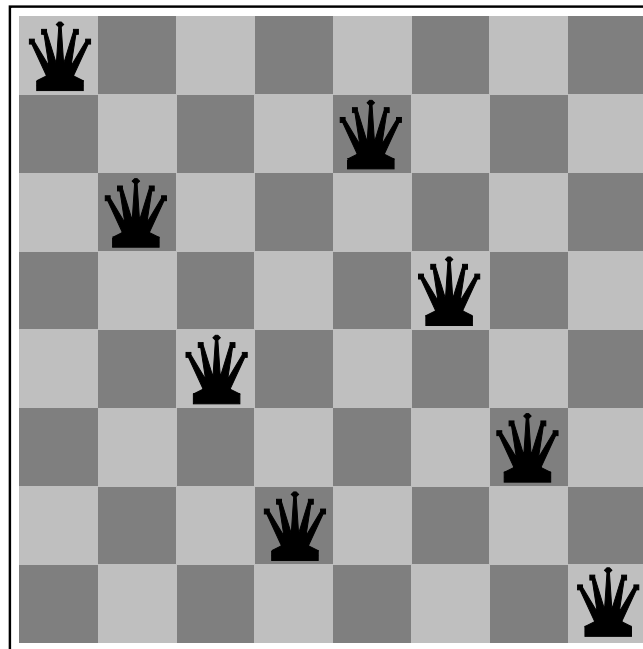
Goal State



- ★  $h_1$ : número de peças fora do lugar
- ★  $h_2$ : distância de cada peça de seu lugar
- ★  $h_3$ : peças fora da formação de caracol

# Exercício

- Determine uma heurística para o problema das 8-raíñas e verifique se é admissível.



★  $h$ : soma do número de ataques

# Algoritmo Subida da Montanha-1

**Idéia:** escolher sempre um sucessor melhor

( “*subir sempre*” ).

**function** BSM-1(Estado *inicial*): Estado

Estado *atual*  $\leftarrow$  *inicial*

**loop**

*prox*  $\leftarrow$  melhor sucessor de *atual* (segundo *h*)

**if**  $h(\textit{prox}) \geq h(\textit{atual})$  **then** {sem sucessor melhor}

**return** *atual*

**end if**

*atual*  $\leftarrow$  *prox*

**end loop**

## Análise do algoritmo BSM-1

- Não mantém a árvore (logo, não pode retornar o caminho que usou para chegar à meta).
- Completo: **não** (problema de **máximos locais**)
- Ótimo: não se aplica
- Tempo: ?
- Espaço: **nada!**

# Algoritmo Subida da Montanha-2

```
function BSM-2(Estado inicial): Estado
Estado atual  $\leftarrow$  inicial
loop
    prox  $\leftarrow$  melhor sucessor de atual (segundo h)
    if  $h(\textit{prox}) \geq h(\textit{atual})$  then {sem sucessor melhor}
        if atual.éMeta() then
            return atual
        else
            atual  $\leftarrow$  estado gerado aleatoriamente
        end if
    else
        atual  $\leftarrow$  prox
    end if
end loop
```

## Análise do algoritmo BSM-2

- Completo: **sim** (se a geração de estados aleatórios distribuir normalmente os estados gerados)
- Ótimo: não se aplica
- Tempo: ?
- Espaço: **nada!**

## Material de **consulta**

- Capítulos 3 e 4 do livro do Russell & Norvig