

# INF623

2024/1



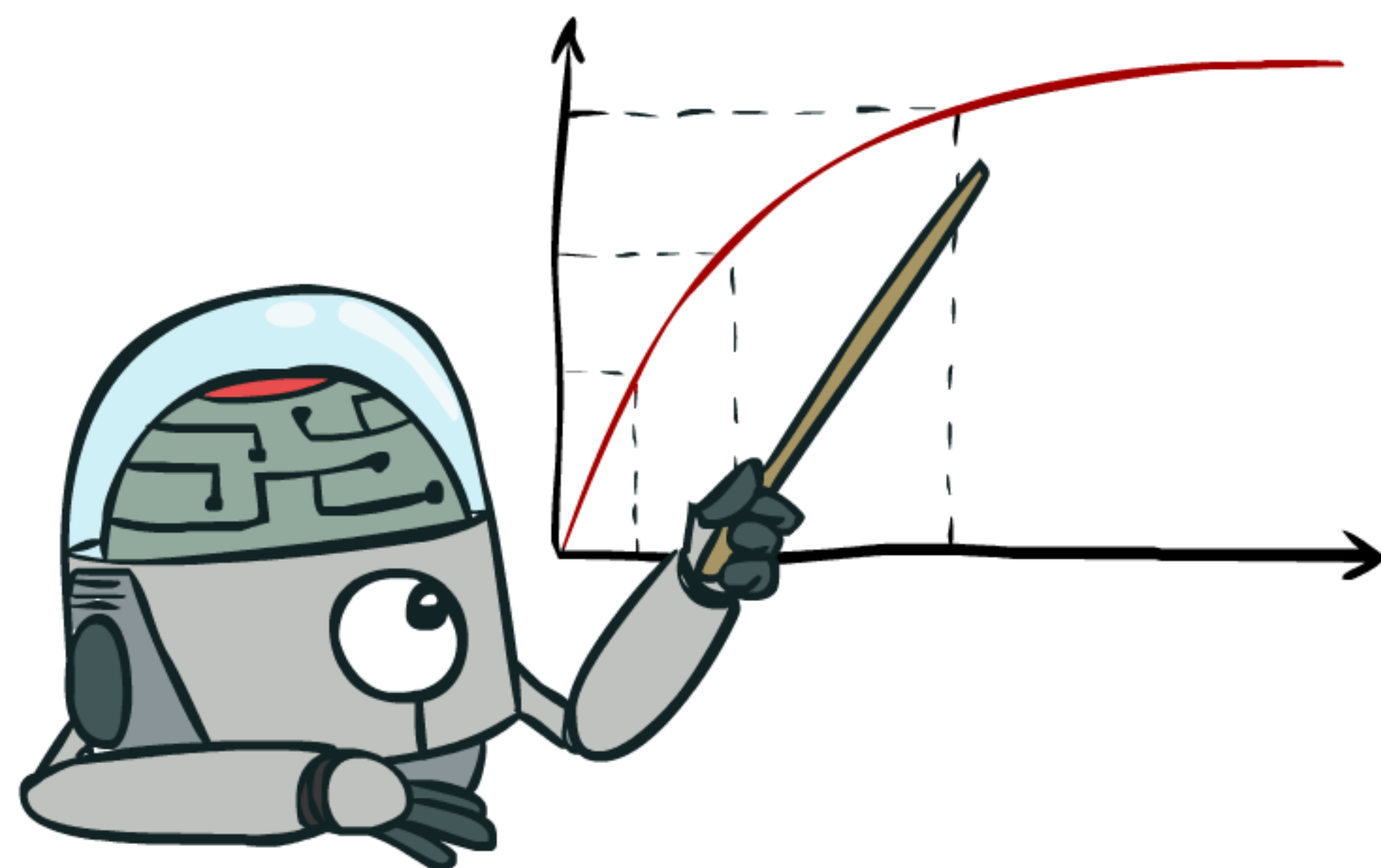
# Inteligência Artificial

## A2: Busca no espaço de estados I

# Plano de aula

- ▶ Formalização de problemas de busca no espaço de estados
- ▶ Espaço de estados
- ▶ Exemplos de problemas
- ▶ Árvore de busca
- ▶ Algoritmos de busca sem informação
  - ▶ Busca em largura
  - ▶ Busca em profundidade
    - ▶ Iterativa vs. recursiva
  - ▶ Busca de custo uniforme

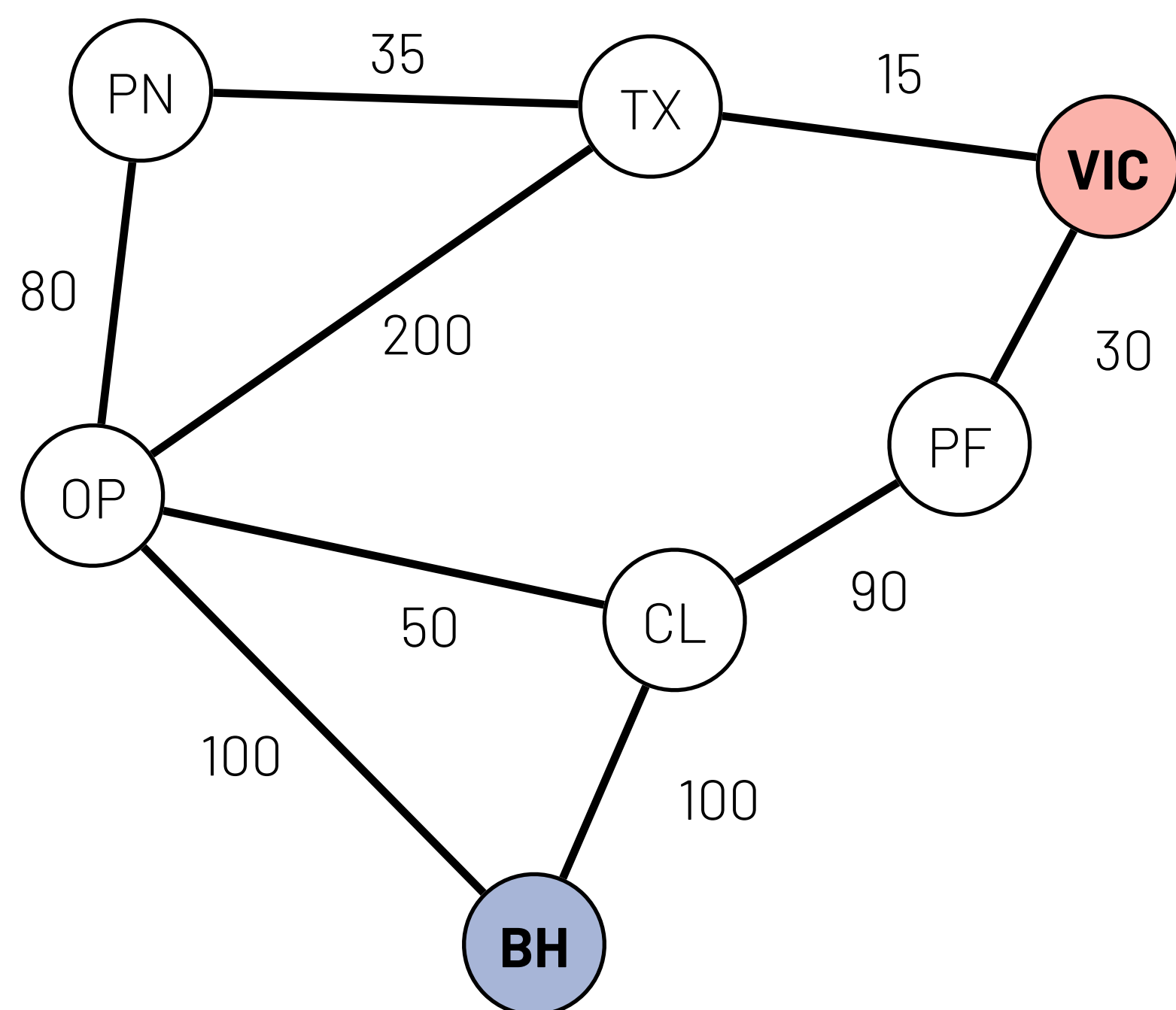
# Agentes racionais



**Agir de maneira autônoma**  
visando atingir o **melhor resultado**  
ou, em caso de incerteza, o melhor  
resultado esperado.

# Exemplo 1: caminho mais curto

Considere o mapa abaixo representando cidades de Minas Gerais e as estradas que as conectam. Os números entre cidades representam o custo de viajar entre uma cidade e outra.



Qual o caminho de menor custo entre **BH** e **Viçosa (VIC)**?

► Solução 1

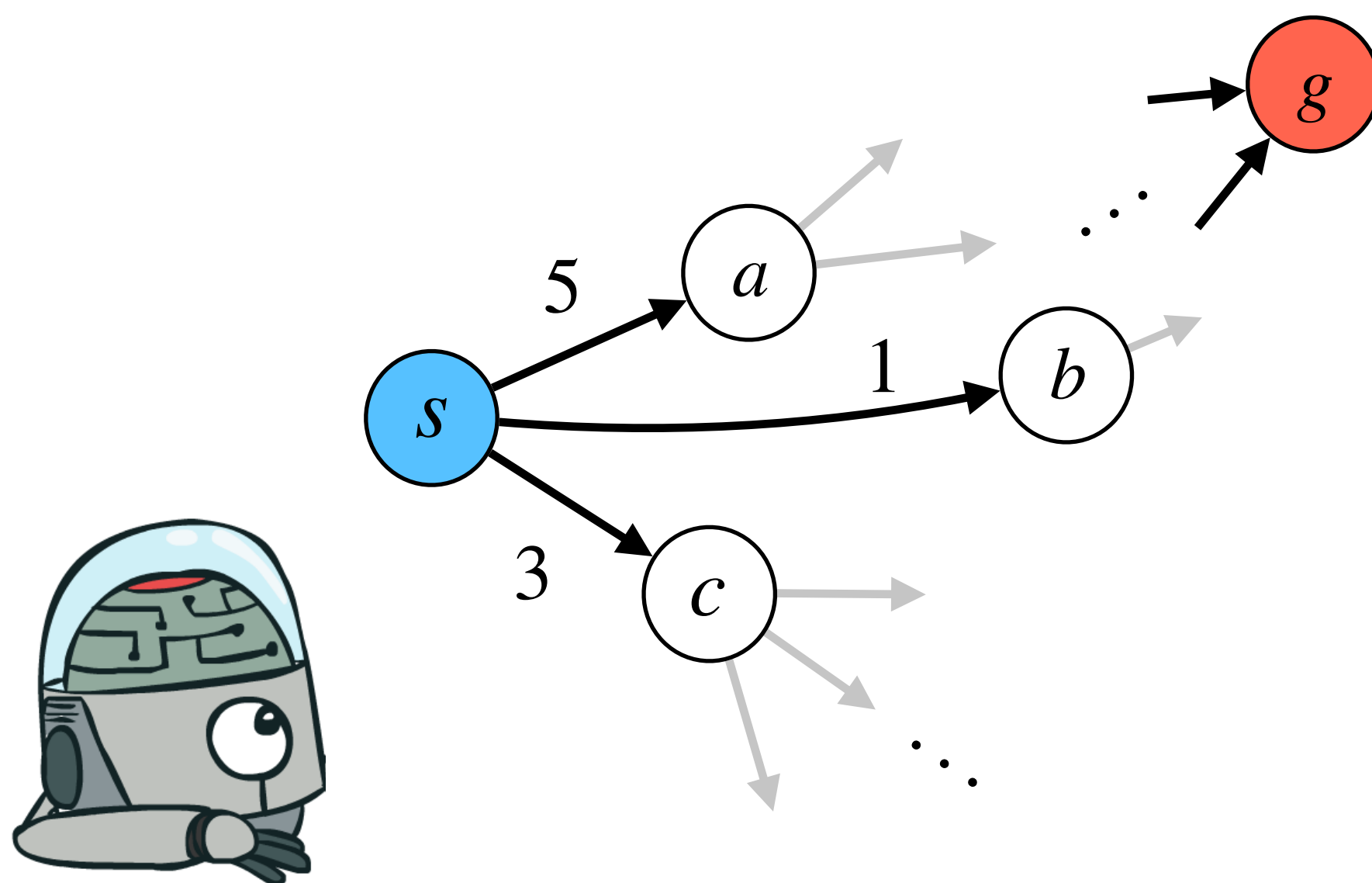
BH, OP, PN, TX, VIC – Custo  $100 + 80 + 35 + 15 = 230$

► Solução 2

BH, CL, PF, VIC – Custo  $100 + 90 + 30 = 220$

# Agentes racionais baseados em busca

Para resolver problemas desse tipo, chamados de **problema de busca**, um agente assume que o mundo é representado por **estados** e que objetivo é chegar em um **estado final  $g$**  a partir de um **estado inicial  $s$** :



- ▶ O agente utiliza **ações** ( $\rightarrow$ ) que possuem **custos** associados para modificar o estado corrente
- ▶ Uma **solução** é uma sequência de ações (um caminho) que leva o agente de  $s$  a  $g$

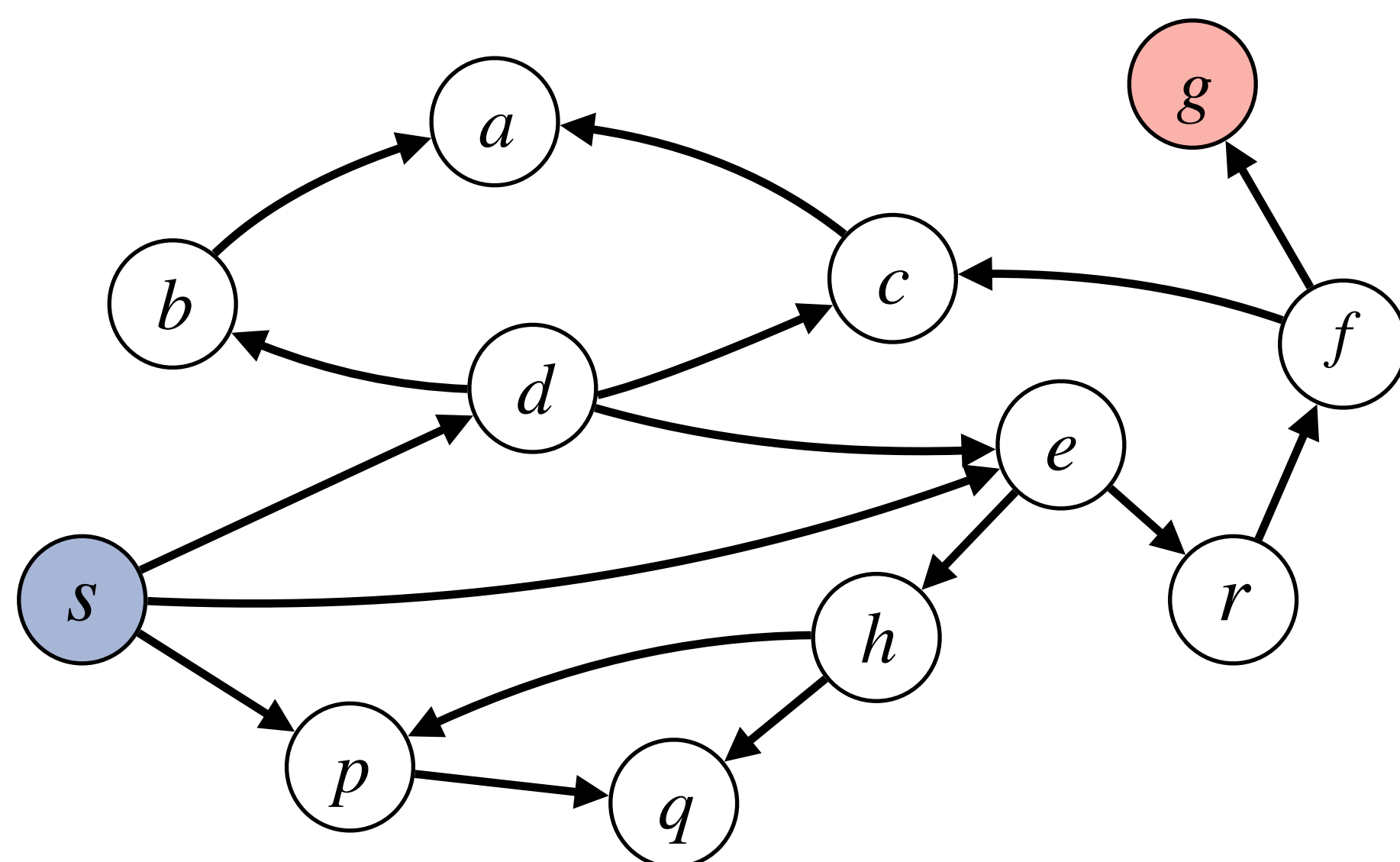
# Problemas de busca

Um problema de busca é definido por:

- ▶ Conjunto de estados  $S$ , chamado de **espaço de estados**
- ▶ **Estado inicial**  $s \in S$
- ▶ **Estado final**  $g \in S$
- ▶ **Função de ações**  $A(s)$  que retorna o conjunto finito de ações possíveis em  $s$
- ▶ **Modelo de transição**  $T(s, a)$ , uma função que retorna um novo estado  $s'$  resultado da aplicação da ação  $a$  no estado  $s$
- ▶ **Função custo de ação**  $C(s, a, s')$  que retorna o custo numérico da aplicação da ação  $a$  no estado  $s$  para alcançar o estado  $s'$

# Espaço de estados como um grafo

O **espaço de estados** pode ser representado como um grafo, onde os **vértices são os estados** e as **arestas são as ações**:

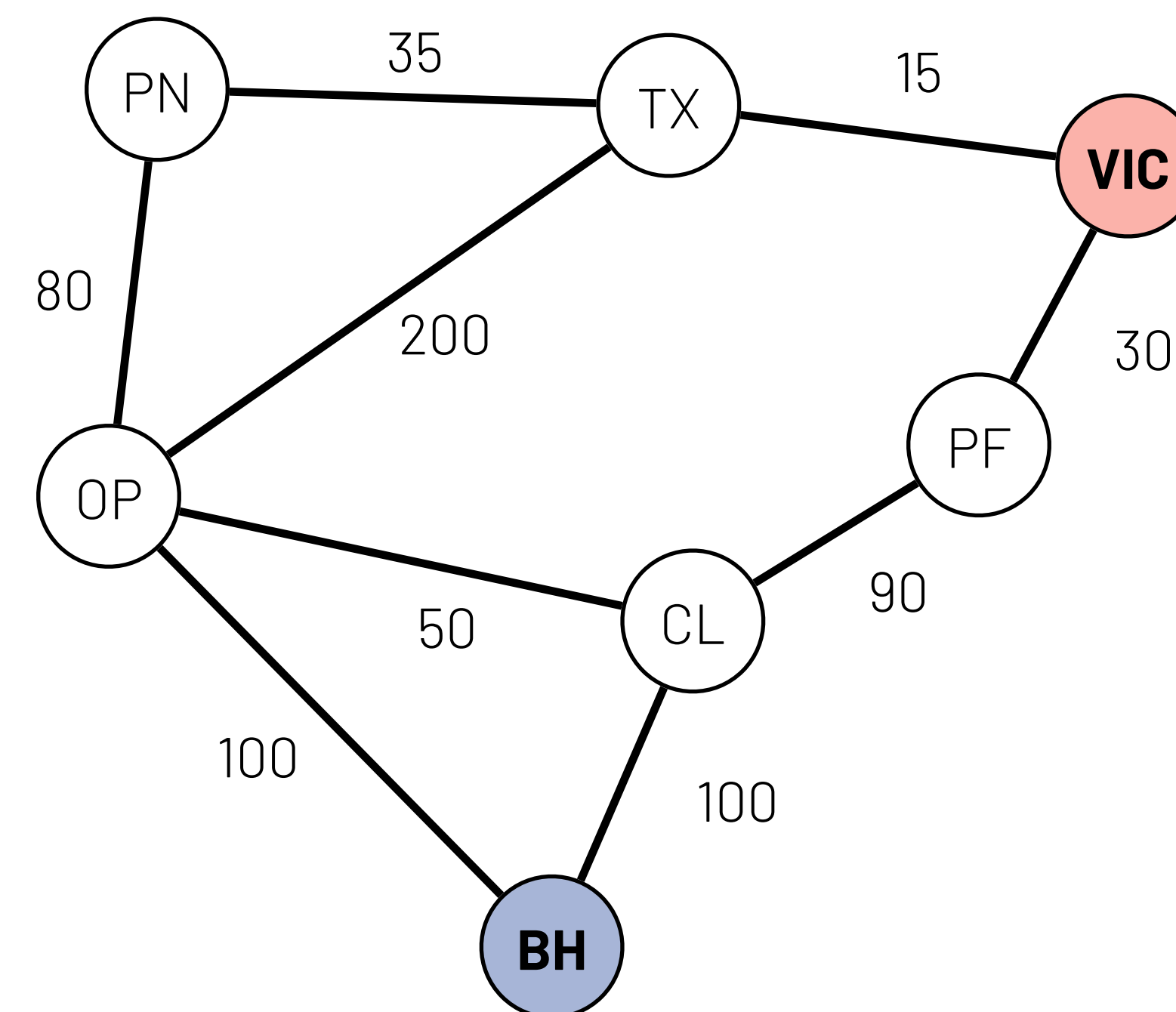


- ▶ O conjunto de vértices é igual ao de estados  $S$
- ▶ O conjunto de arestas  $A$  contém as ações
- ▶ Uma **solução** para um problema de busca é uma sequência de estados (caminho)  $s, s_1, s_2, \dots, g$  tal que  $(s_i, s_{i+1}) \in A$ , para todo  $s_i$  na sequência
- ▶ Um **caminho  $C$  é ótimo**, também denotado como  $C^*$ , se não existe nenhum outro caminho entre  $s$  e  $g$  com custo menor que  $C$



# Exemplo 1: caminho mais curto

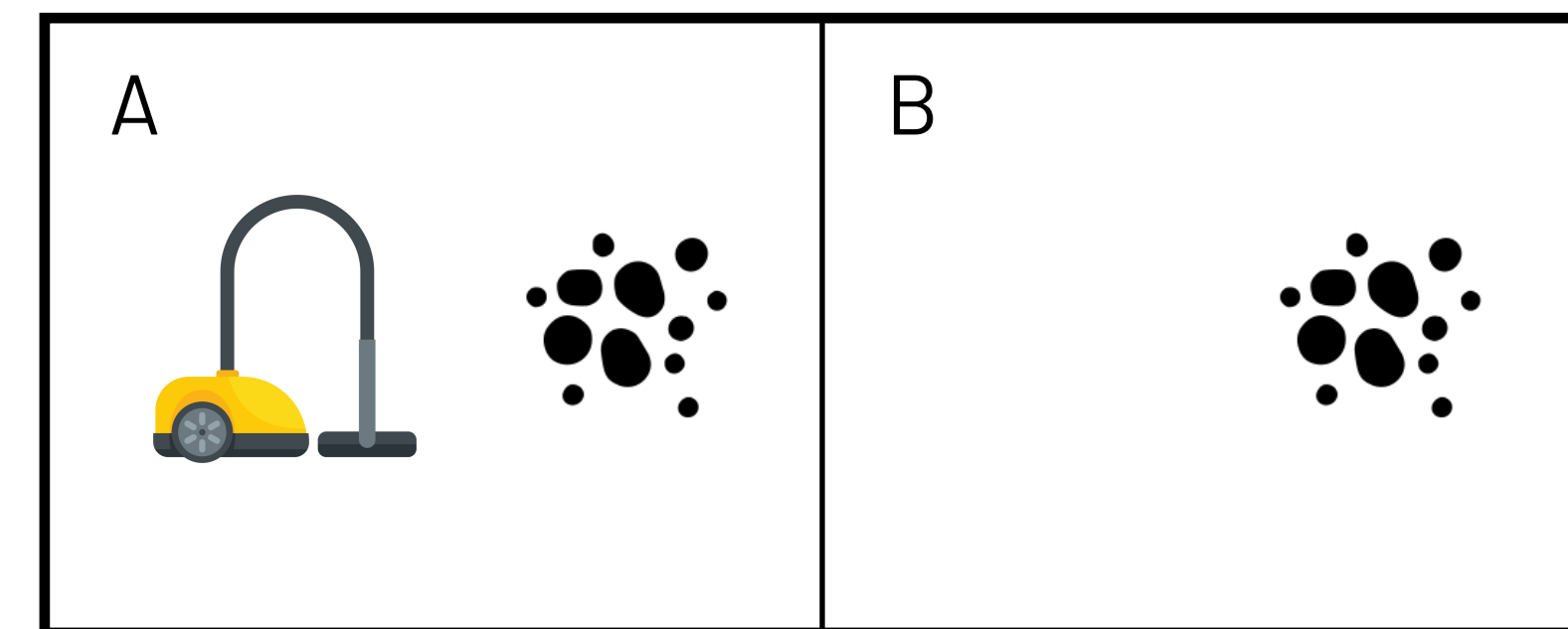
- ▶ **Espaço de estados:** o conjunto de cidades {BH, OP, CL, PN, TX, PF, VIC} – 8 estados
- ▶ **Estado inicial:** BH
- ▶ **Estado final:** Viçosa (VIC)
- ▶ **Ações:** viajar para uma cidade adjacente  
Exemplo –  $A(BH) = \{\text{viajar para OP, viajar para CL}\}$
- ▶ **Modelo de transição:** se viajar para uma cidade adjacente, o agente passa para o estado que representa essa cidade  
Exemplo –  $T(BH, \text{viajar para OP}) = OP$
- ▶ **Função custo de ação:** a distância em km de uma cidade a outra  
Exemplo –  $C(BH, \text{viajar para OP, OP}) = 100$





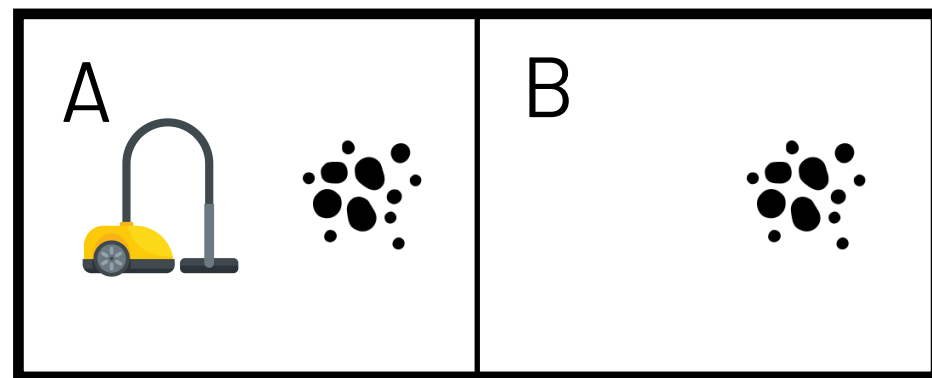
# Exemplo 2: mundo do aspirador de pó

- ▶ **Espaço de estados:** tupla com a posição do agente e se há ou não poeira em A e B
- ▶ **Estado inicial:** qualquer um
- ▶ **Estado final:** A e B limpos
- ▶ **Ações:** aspirador pode mover para esquerda, direita e aspirar  
 $A(s) = \{\text{ESQUERDA}, \text{DIREITA e ASPIRAR}\}$
- ▶ **Modelo de transição:** se mover para esquerda em A, o agente permanece onde está, se mover para a direita em A, ele move para B; se aspirar e houver poeira, a poeira desaparece
- ▶ **Função custo de ação:** cada ação custa 1

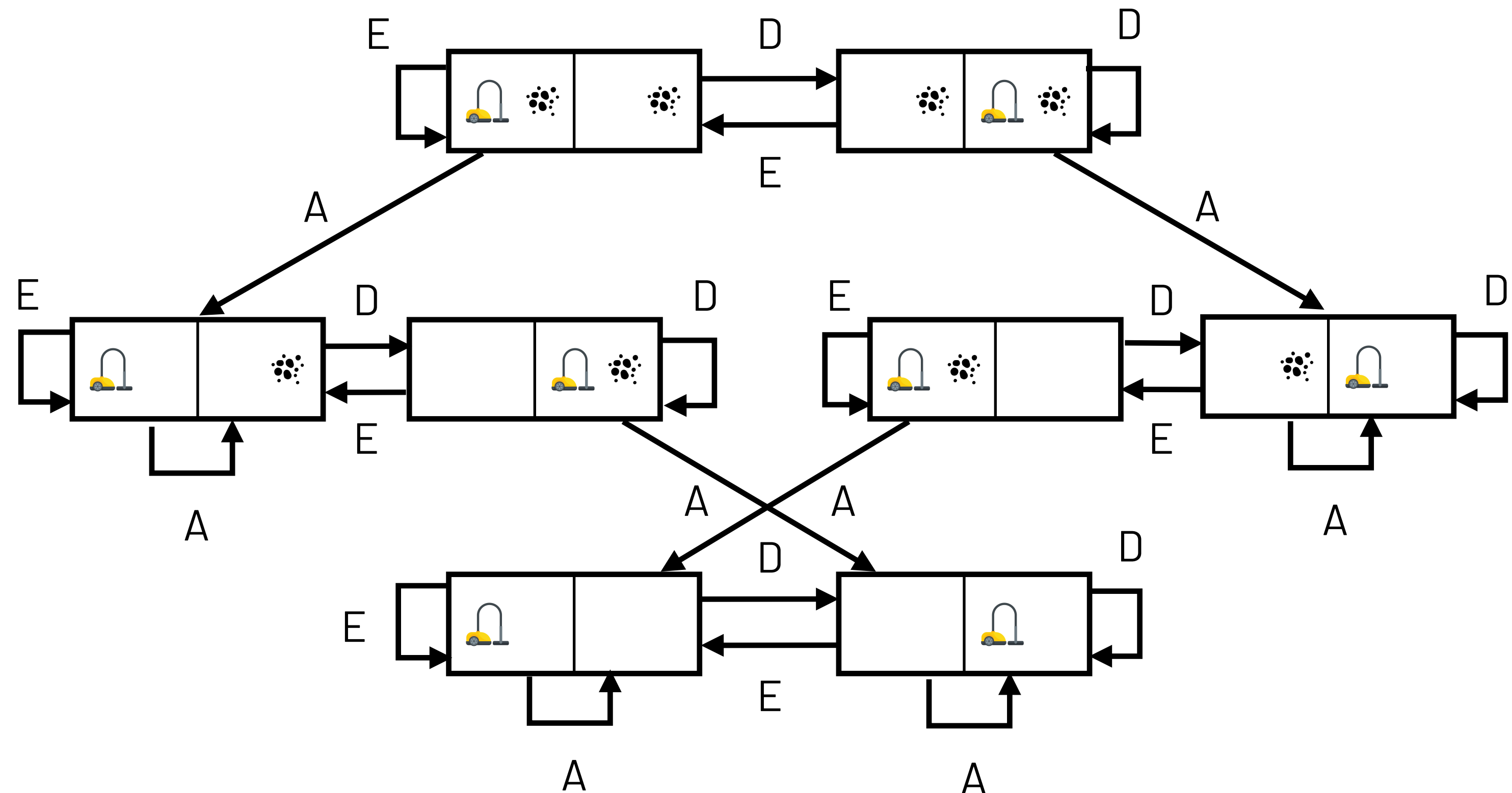


# Exercício 1

Quantos estados existem no problema do aspirador? Quantos estados finais existem? Desenhe o grafo do espaço de estados.



R: O agente pode estar em uma das duas posições, e cada posição pode conter poeira ou não, portanto:  
 $2 \times 2 \times 2 = 8$  estados



# Exemplo 3: quebra-cabeça de oito peças

- ▶ **Espaço de estados:** todas as configurações possíveis do tabuleiro.
- ▶ **Estado inicial:** qualquer um
- ▶ **Estado final:** um estado onde os números aparecem em ordem crescente da esquerda para a direita, de cima para baixo
- ▶ **Ações:** Quadrado vazio (em branco) move para CIMA, BAIXO, ESQUERDA e DIREITA. Se ele estiver nos cantos, alguns movimentos se tornam inválidos
- ▶ **Modelo de transição:** mover o quadrado vazio troca sua posição com um adjacente. Por exemplo, na estado inicial acima, mover o branco para a esquerda, troca o 5 com o vazio
- ▶ **Função custo de ação:** cada ação custa 1

7	2	4
5		6
8	3	1

Estado inicial

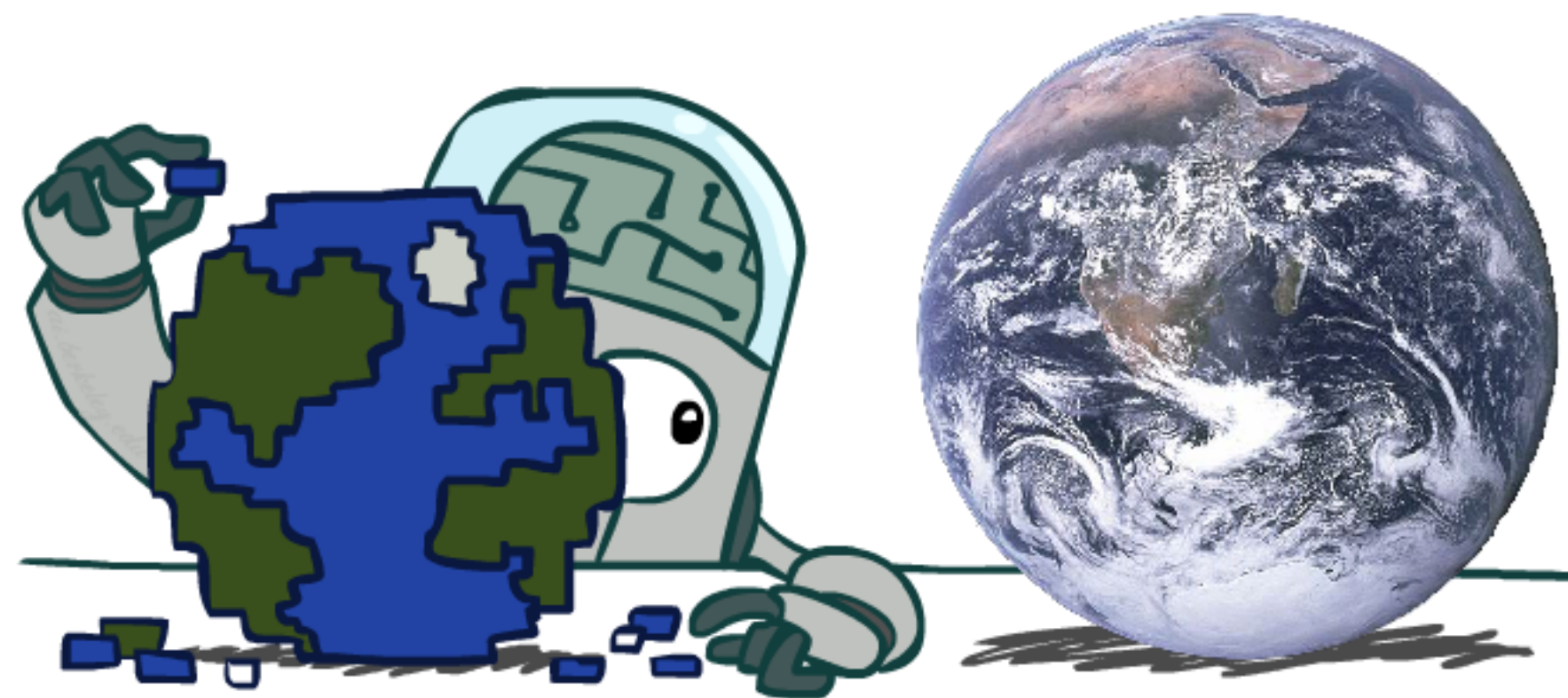
	1	2
3	4	5
6	7	8

Estado final

# Representação de estados

Quando formulamos problemas de busca, **abstraímos os detalhes que não são importantes** para representar os estados, mantendo apenas os detalhes necessários.

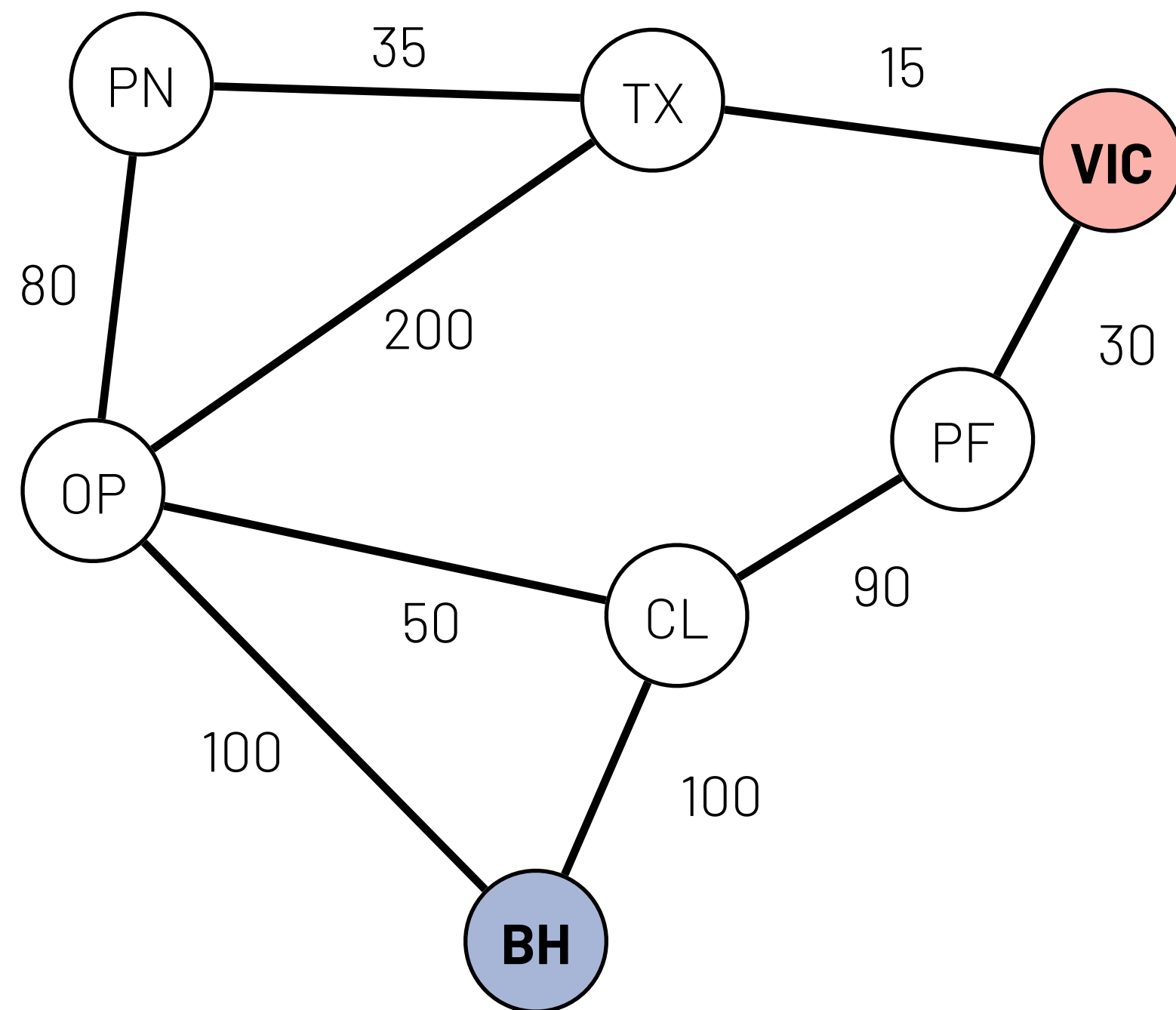
Exemplos:



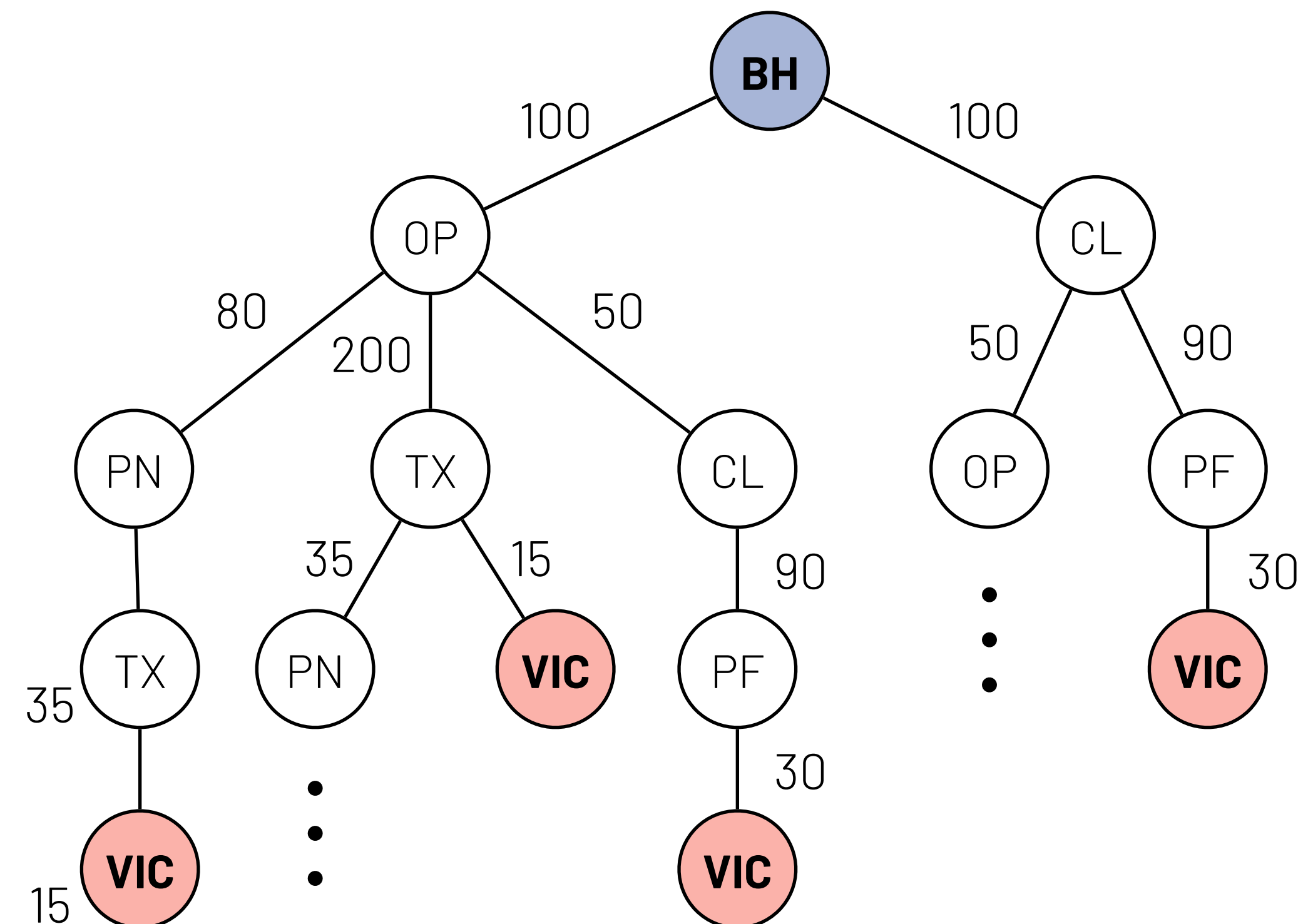
- ▶ **Caminho mais curto:** consideramos apenas a distância entre as cidades. Desconsideramos, entre tantas outras coisas, as condições climáticas das estradas.
- ▶ **Quebra-cabeças:** consideramos apenas as posições finais das peças, desconsideramos as posições intermediárias de deslizamento.

# Árvore de busca

Em IA, os grafos do espaço de estados geralmente não são armazenados em memória, pois, na prática, costumam ser muito grandes.



Ao invés disso, podemos aplicar a função do modelo de transição  $T(s, a)$  para gerar uma **árvore de busca**, começando pelo estado inicial.

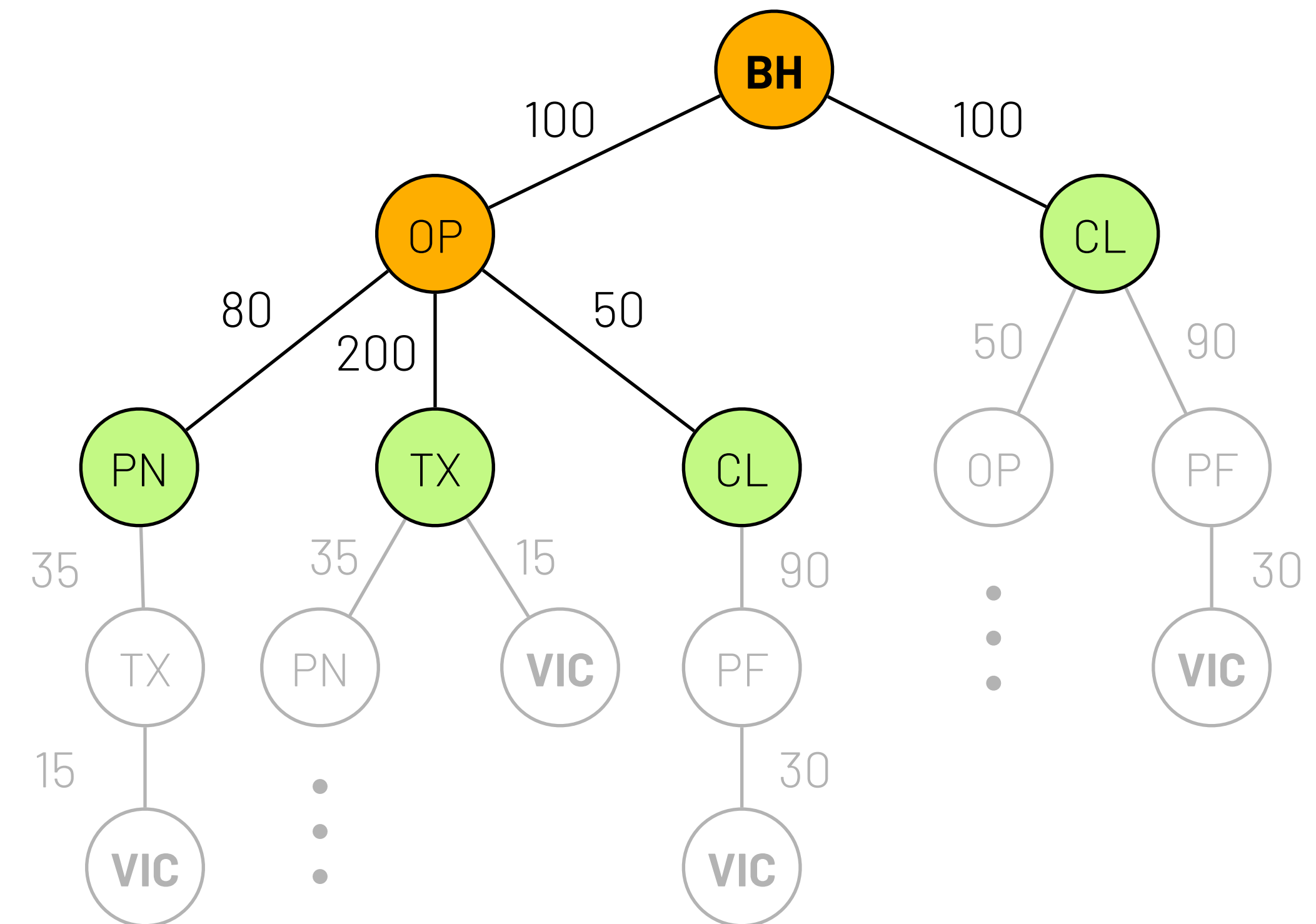




# Árvore de busca

## Definições:

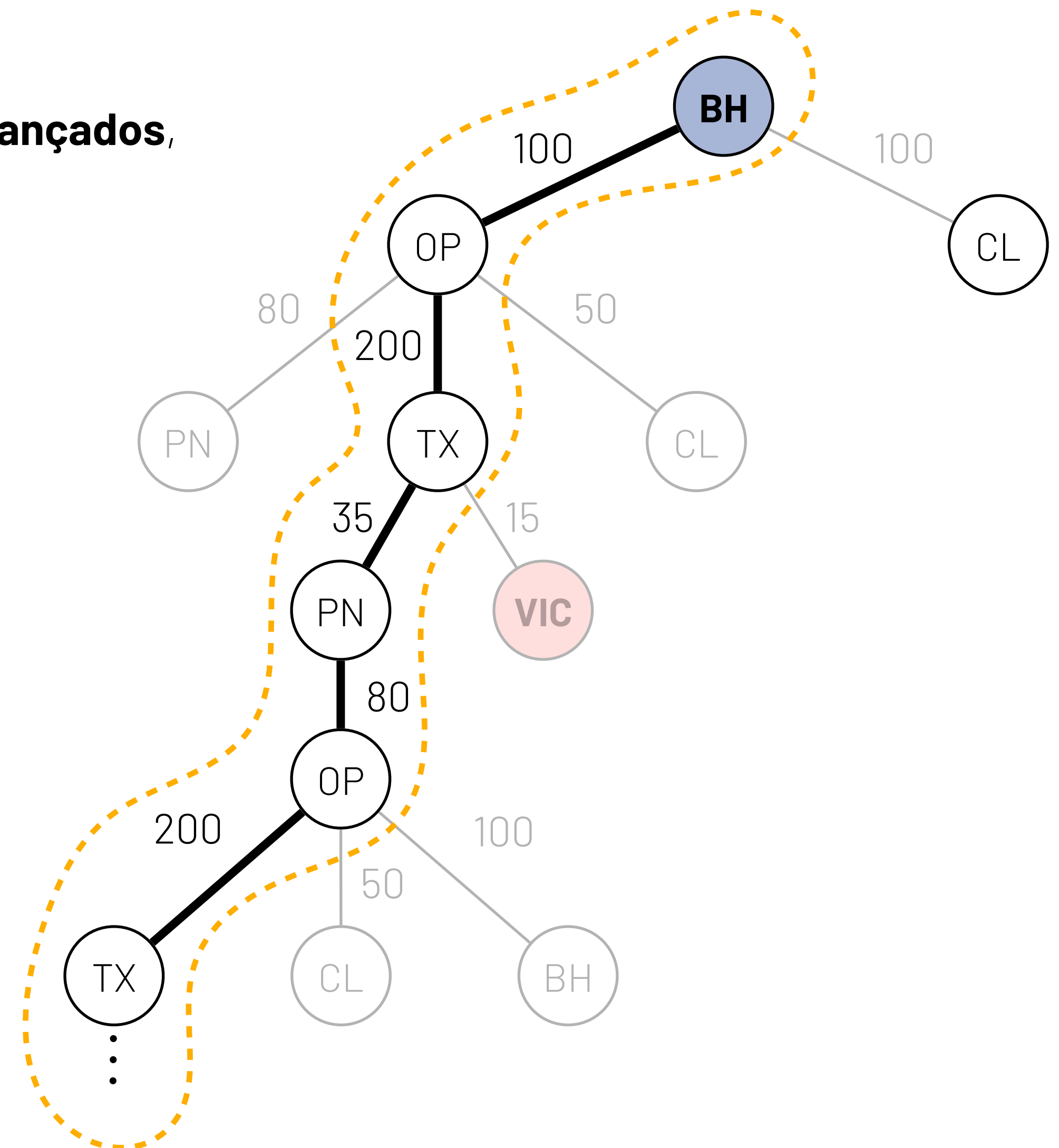
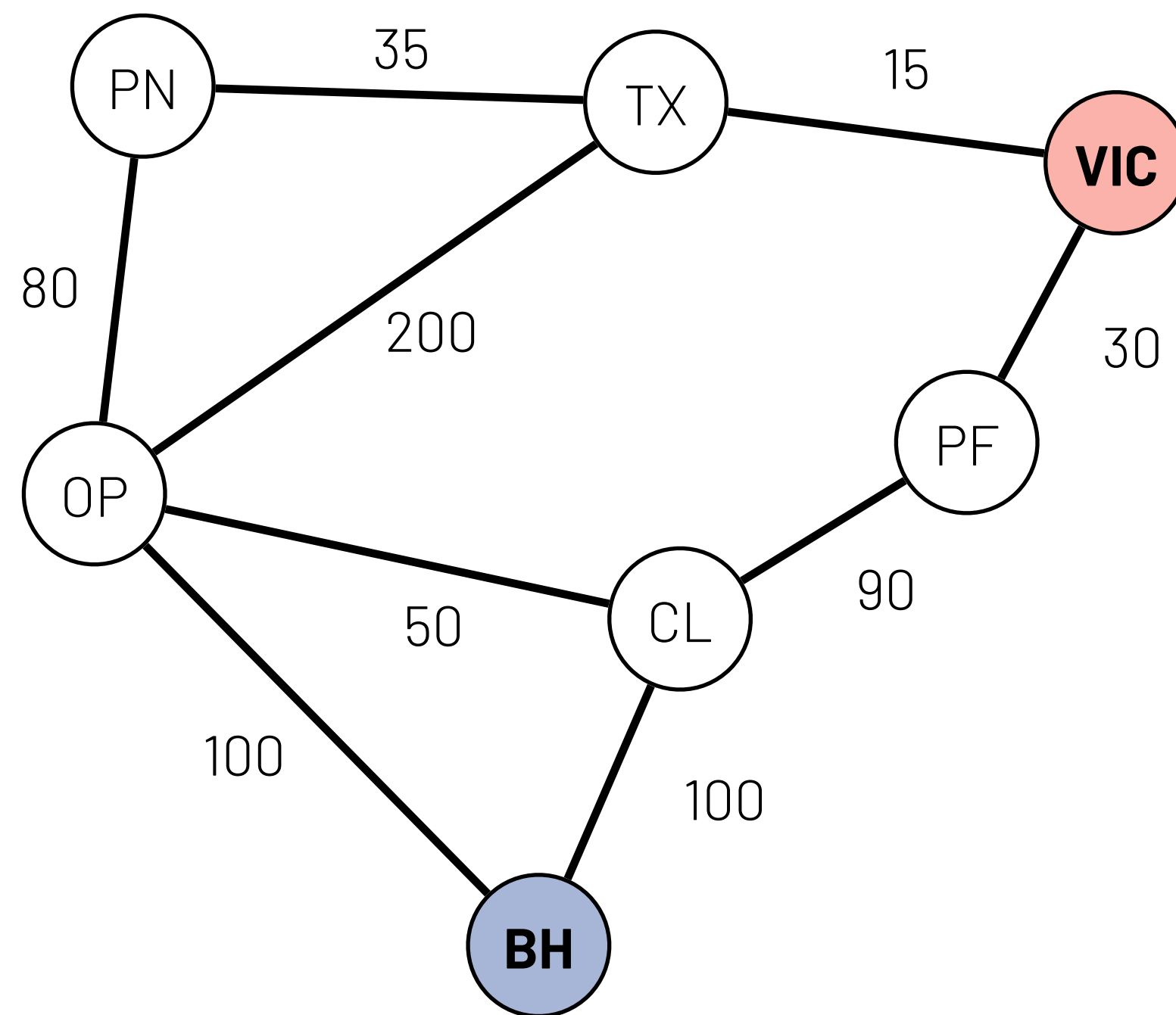
- ▶ O estado inicial é a **raiz** da árvore;
- ▶ Um nó é **expandido** quando um algoritmo considera as ações para aquele estado  $s$  chamando a função  $A(s)$ ;
- ▶ Um nó é **gerado** quando seu pai é expandido;
- ▶ O conjunto de caminhos produzidos até o momento é chamado de **fronteira**;
- ▶ Um **ciclo** ocorre quando um nó aparece múltiplas vezes em um mesmo caminho (e.g., BH, OP, TX, PN, OP, TX, VIC)



# Ciclos e caminhos redundantes

Ciclos são caminhos redundantes geram **árvores de busca infinitas**:

- ▶ Podemos evitar ciclos durante a busca com uma tabela de nós **alcançados**, expandindo apenas aqueles que:
  - ▶ Ainda não foram visitados ou;
  - ▶ Estão sendo visitados por um caminho melhor.





# Algoritmos de busca (que estudaremos)

## ▶ Busca sem informação

Não possuem informação sobre a distância entre um determinado estado  $e$  e o estado final  $g$

- ▶ Busca em largura (Breath-first search - BFS) – assume que ações todas tem o mesmo custo
- ▶ Busca em profundidade (Depth-first search- DFS) – assume que ações todas tem o mesmo custo
- ▶ Busca de custo uniforme (Algoritmo de Dijkstra) – assume ações com custo diferentes

## ▶ Busca informada

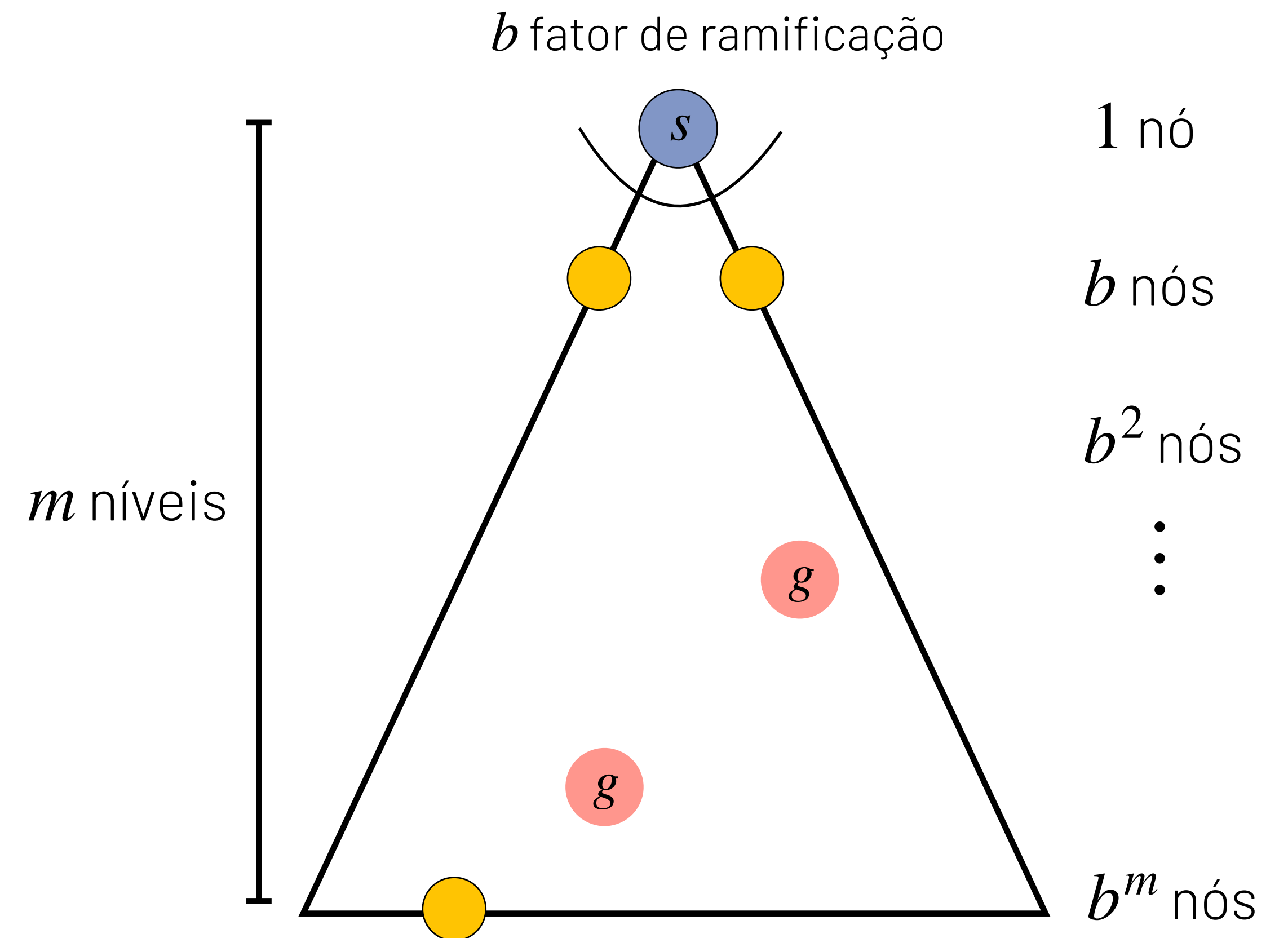
Possuem informação sobre a distância entre um determinado nó e o estado final

- ▶ Busca gulosa de melhor escolha
- ▶ Algoritmo A\*

# Propriedades dos algoritmos de busca

- ▶ Complexidade de tempo  
Quanto tempo demora para o algoritmo retornar uma solução?
- ▶ Complexidade de espaço  
Qual a quantidade de memória necessária para realizar a busca?
- ▶ Completo  
Se existe uma solução, o algoritmo irá encontrá-la;
- ▶ Ótimo  
Se existe uma solução, o algoritmo retorna uma solução ótima;

Em IA, como o grafo é **implícito**, a complexidade é medida em função do fator de ramificação  $b$  e da profundidade máxima  $m$  da árvore de busca.



# Algoritmo genérico de busca em árvore

Os algoritmos de busca em árvore seguem a mesma estrutura geral:

```
def busca-arvore(s, g, A, T, C):  
    1. fronteira = [s] # Inicializar a fronteira com o estado inicial s  
    2. alcancado = {s} # Marcar nó inicial como visitado  
    3. custo[s] = 0     # Inicializar custo do estado inicial  
    4. while fronteira não estiver vazia:  
        5.     n = fronteira.pop() # Escolher um nó da fronteira para expandir  
        6.     if n == g:           # Verificar se o nó n escolhido é o estado final g  
        7.         return caminho entre s e g  
        8.     for filho in T(n, A(n)): # Expandir o nó n escolhido usando função de ações A  
        9.         custo_filho = custo[n] + C(n, filho) # Calcular custo de chegar até o filho por n  
        10.        if filho not in alcancado or custo_filho < custo[filho]:  
        11.            fronteira.append(filho)  
        12.            alcancado.append(filho)  
        13.            custo[filho] = custo_filho
```

A principal diferença entre os algoritmos é a **estratégia de expansão** do nó  $n$ ; Usamos diferentes **estruturas de dados** para implementar essas estratégias.

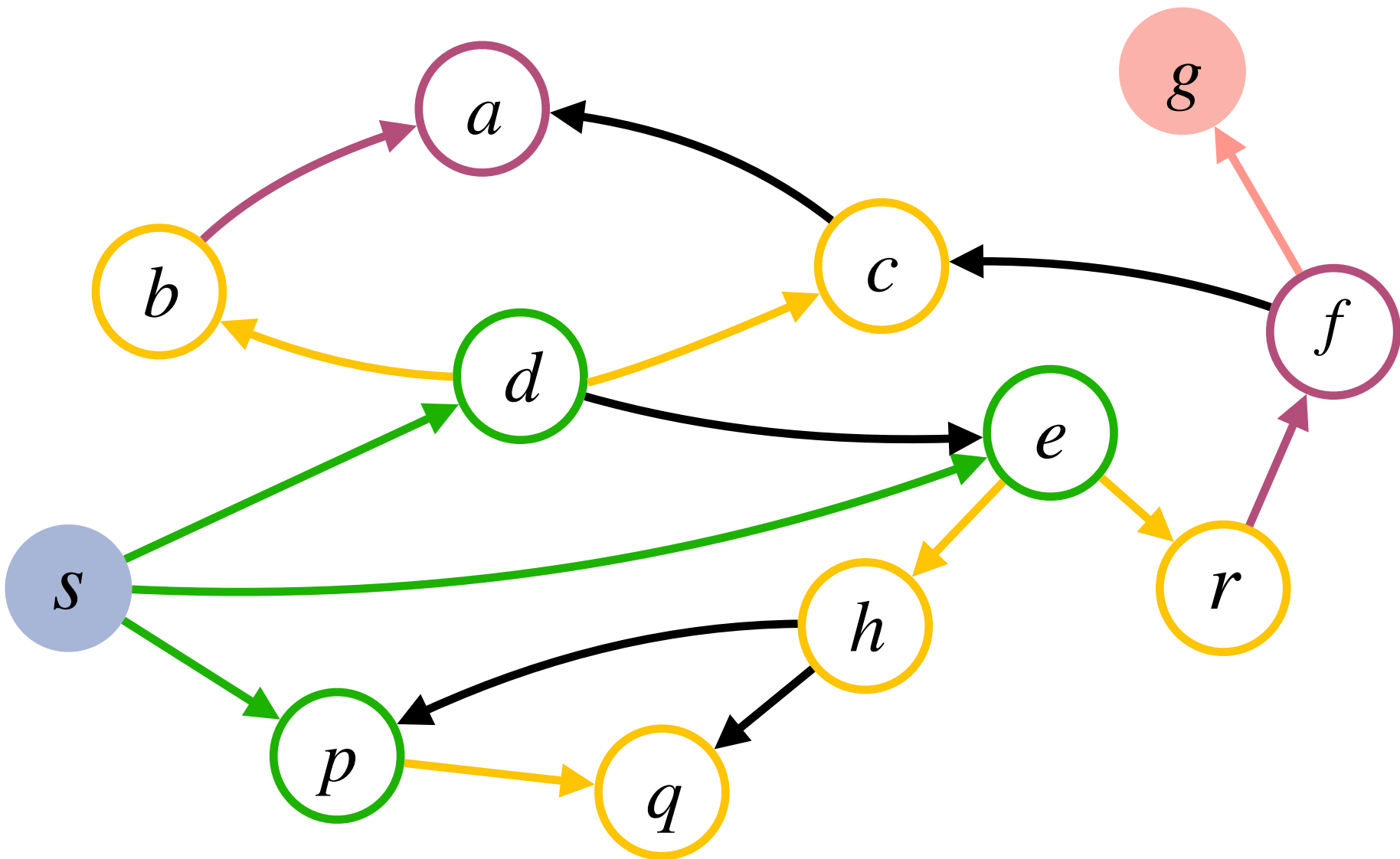
**alcancado** é uma tabela hash (dicionário em python) utilizada para evitar ciclos.

# Busca em largura

## Fronteira é uma fila (FIFO)

Expandir o nó mais raso primeiro

- Nós a uma aresta de distância (d, e, p)
- Nós a duas arestas de distância (b, c, h, r, q)
- Nós a três arestas de distância (a, f)
- ...



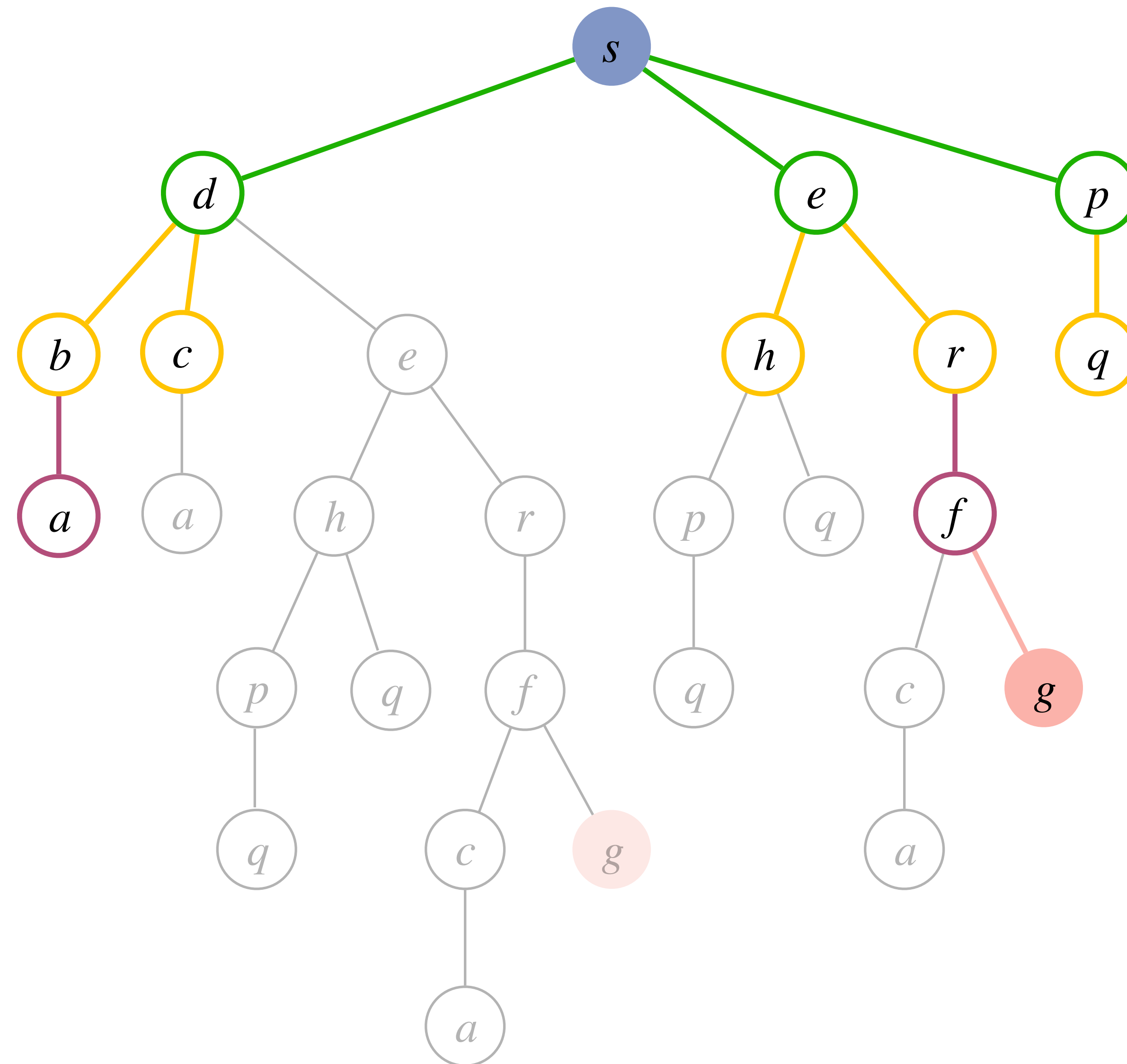
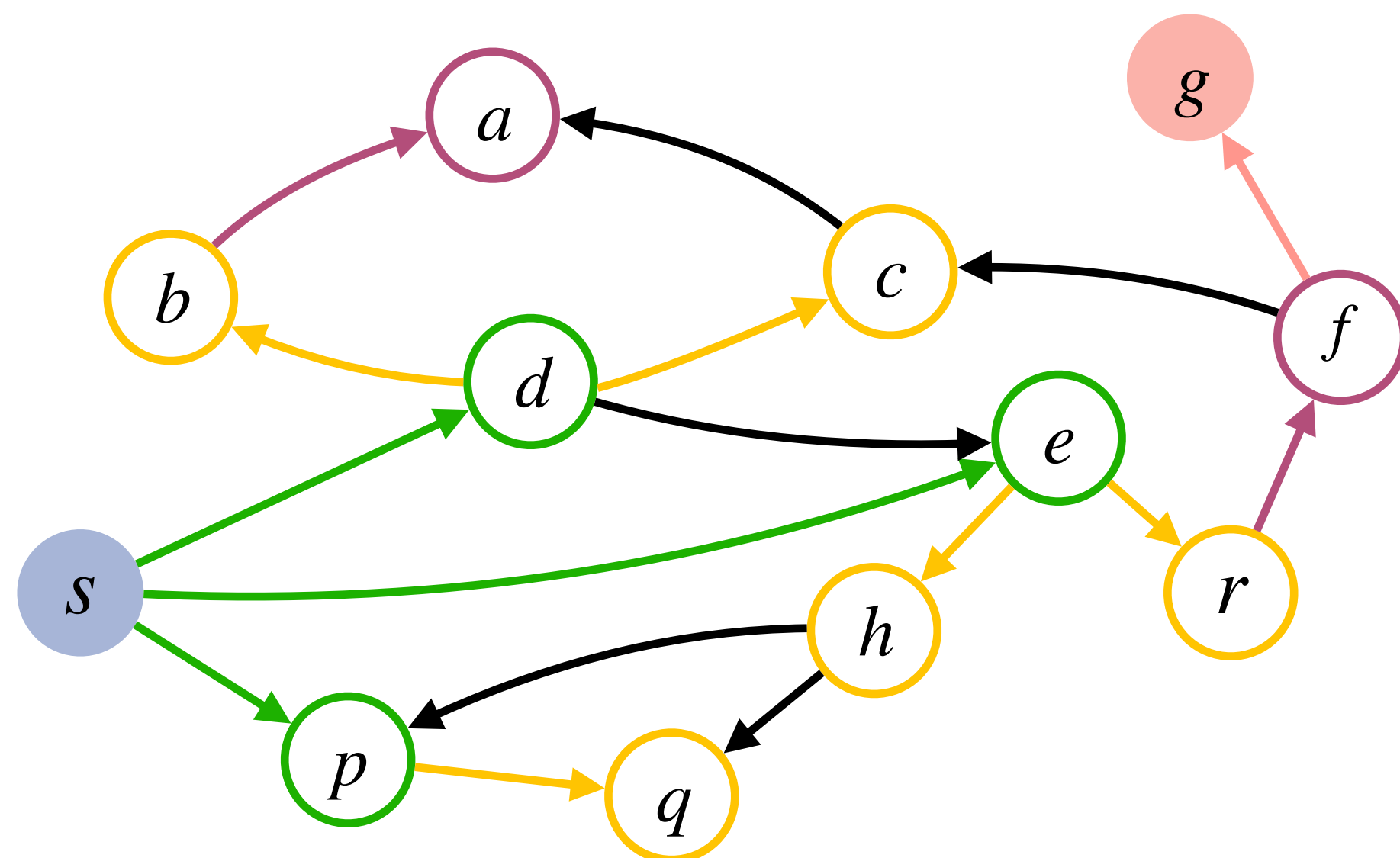
Tempo	Nó	Fronteira (fila)	Alcançado
1	s	[d, e, p]	{s, d, e, p}
2	d	[e, p, b, c]	{d, e, p, b, c}
3	e	[p, b, c, h, r]	{d, e, p, b, c, h, r}
4	p	[b, c, h, r, q]	{d, e, p, b, c, h, r, q}
5	b	[e, h, r, q, a]	{d, e, p, b, c, h, r, q, a}
6	c	[h, r, q, a]	{d, e, p, b, c, h, r, q, a}
7	h	[r, q, a]	{d, e, p, b, c, h, r, q, a}
8	r	[q, a, f]	{d, e, p, b, c, h, r, q, a}
9	q	[a, f]	{d, e, p, b, c, h, r, q, a}
10	a	[f, g]	{d, e, p, b, c, h, r, q, a}
11	f	[g]	{d, e, p, b, c, h, r, q, a}
12	g	[]	{d, e, p, b, c, h, r, q, a}

# Busca em largura

## Fronteira é uma fila (FIFO)

Expandir o nó mais raso primeiro

- ▶ Nós a uma aresta de distância (d, e, p)
- ▶ Nós a duas arestas de distância (b, c, h, r, q)
- ▶ Nós a três arestas de distância (a, f)
- ▶ ...





# Implementação da busca em largura

```
def BFS(s, g, A, T, C):
```

```
1. fila = [s]
```

```
2. alcancado = {s}
```

```
3. custo[s] = 0
```

```
4. while fila não estiver vazia:
```

```
5.     n = fila.pop(0)           # Escolher o primeiro nó da fila para expandir
```

```
6.     if n == g:                # Verificar se o nó n escolhido é o estado final g
```

```
7.         return caminho entre s e g
```

```
8.     for filho in T(n, A(n)):  # Expandir o nó n escolhido usando função de ações A
```

```
9.     custo_filho = custo[n] + C(n, filho) # Calcular custo de chegar até o filho por n
```

```
10.     if filho not in alcancado or custo_filho < custo[filho]:
```

```
11.         fila.append(filho)
```

```
12.         alcancado.append(filho)
```

```
13.     custo[filho] = custo_filho
```

Na BFS, a **fronteira** é uma **fila (FIFO)**

Na BFS, não é necessário manter os custos

# Propriedade da Busca em Largura

► Complexidade de tempo

Explora todos os nós acima da solução mais rasa — seja  $d$  a profundidade da solução mais rasa, a complexidade de tempo é  $O(b^d)$

► Complexidade de espaço

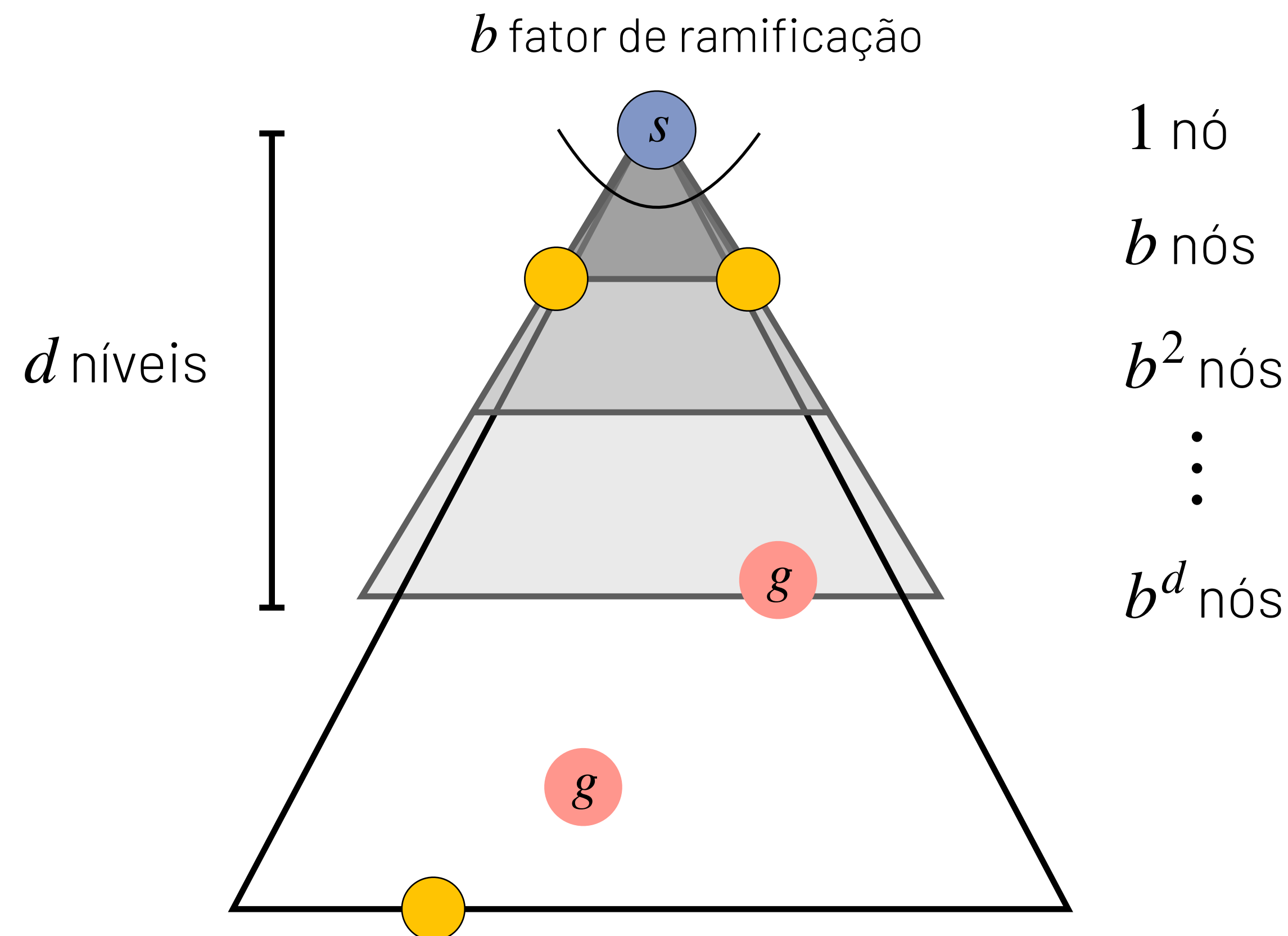
Armazena todos os nós até o nível  $d$  antes da solução  $g$ , portanto complexidade de espaço é  $O(b^d)$

► Completo

Sim. Se existe uma solução, então  $d$  é finito e a BFS vai encontrá-la

► Ótimo

Sim, mas apenas se os custos forem uniformes (iguais a 1)





# Próxima aula

**A3:** Busca no espaço de estados II

Algoritmos de busca em profundidade, busca de custo uniforme, função heurísticas estratégias informadas: busca gulosa de melhor escolha e  $A^*$