

INF623

2024/1



Inteligência Artificial

A3: Busca no espaço de estados II

Plano de aula

- ▶ Algoritmos de busca sem informação
 - ▶ Busca em profundidade
 - ▶ Iterativa vs. recursiva
 - ▶ Busca de custo uniforme
- ▶ Função Heurística
- ▶ Algoritmos de busca informada
 - ▶ Busca guloso pela melhor escolha
 - ▶ A^*

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.

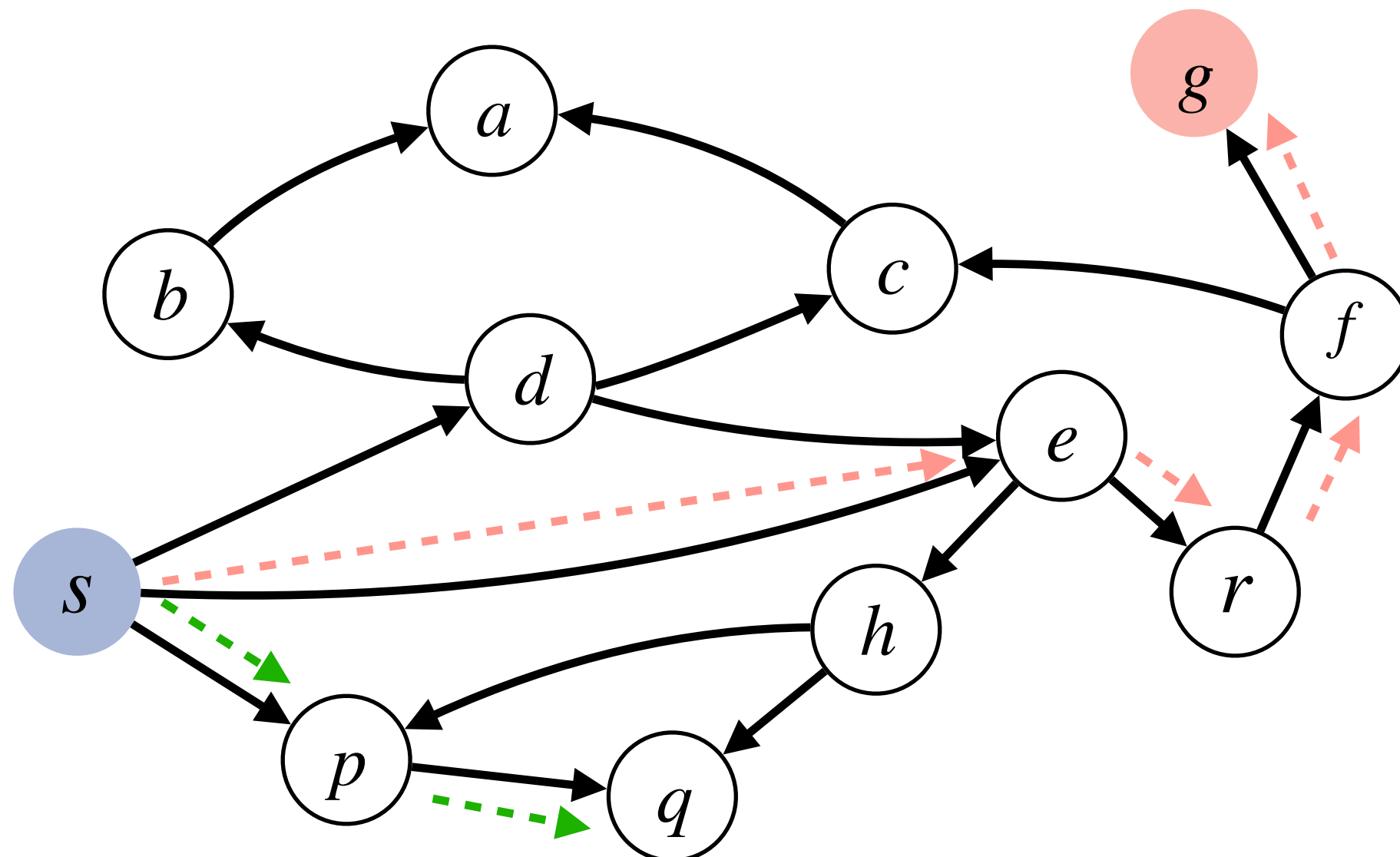
alcancado é uma tabela hash (dicionário em python) utilizada para evitar ciclos.

Busca em profundidade (iterativa)

Fronteira é uma pilha (LIFO)

Expandir o nó mais profundo primeiro

- ▶ Nós do primeiro caminho
- ▶ Nós do segundo caminho
- ▶ Nós do terceiro caminho
- ▶ ...



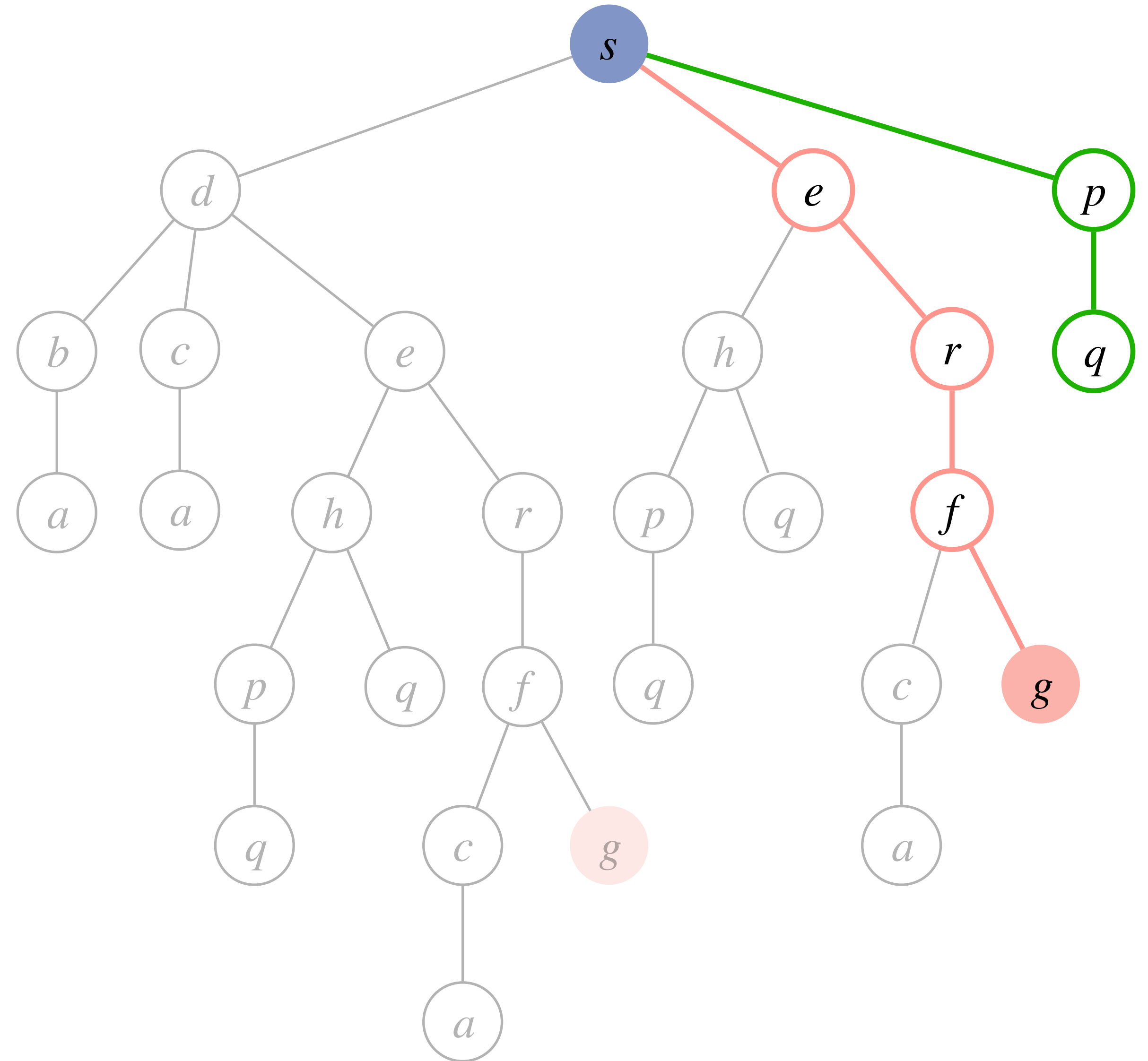
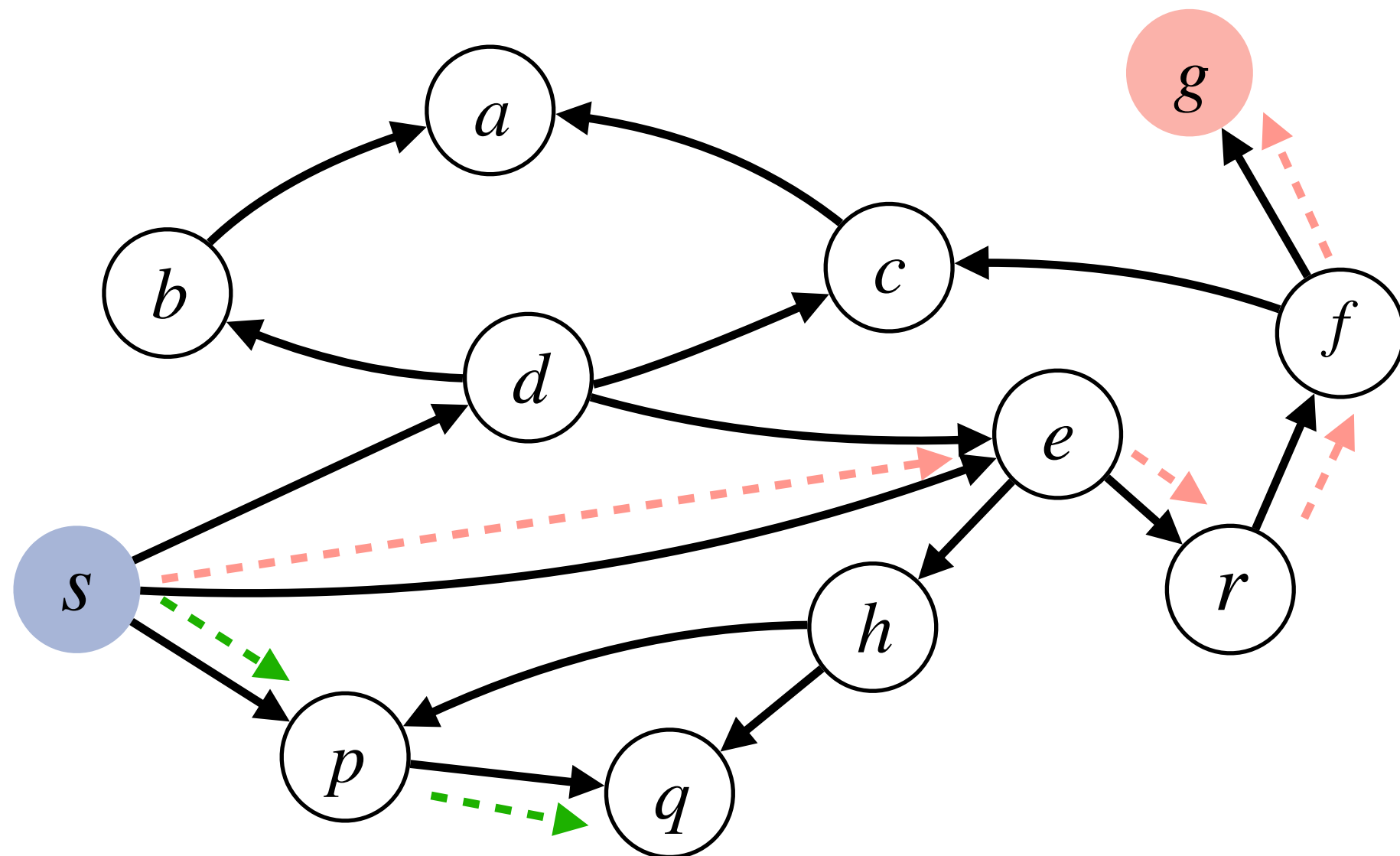
Tempo	Nó	Fronteira (pilha)	Alcançado
1	s	[d, e, p]	{s, d, e, p}
2	p	[d, e, q]	{s, d, e, p, q}
3	q	[d, e]	{s, d, e, p, q}
4	e	[d, h, r]	{s, d, e, p, q, h, r}
5	r	[d, h, f]	{s, d, e, p, q, h, r, f}
6	f	[d, h, g]	{s, d, e, p, q, h, r, f}
7	g	[d, h]	{s, d, e, p, q, h, r, f}
8			
9			
10			
11			
12			

Busca em profundidade (iterativa)

Fronteira é uma pilha (LIFO)

Expandir o nó mais profundo primeiro

- ▶ Nós do primeiro caminho
- ▶ Nós do segundo caminho
- ▶ Nós do terceiro caminho
- ▶ ...



Implementação iterativa da busca em profundidade

```
def BFS(s, g, A, T, €):
```

```
1. pilha = [s]
```

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

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

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

```
5.     n = pilha.pop()           # Escolher o último 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.             pilha.append(filho)
```

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

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

Na DFS, a **fronteira** é uma **pilha (LIFO)**

Na DFS, também não é necessário manter os custos

Implementação recursiva da busca em profundidade

```
def DFS_util(n, g, A, T, alcancado):  
    1. if n == g:  
    2.     return caminho entre s e g  
    3. for filho in T(n, A(n)):  
    4.     if filho not in alcancado:  
    5.         alcancado.append(n)  
    6.         DFS_util(filho, g, A, T, alcancado)
```

Na implementação recursiva, a fronteira é a pilha de chamadas recursivas da função DFS_util

```
def DFS(s, g, A, T):  
    1. alcancado = {s}  
    2. DFS_util(s, g, A, T, alcancado)
```

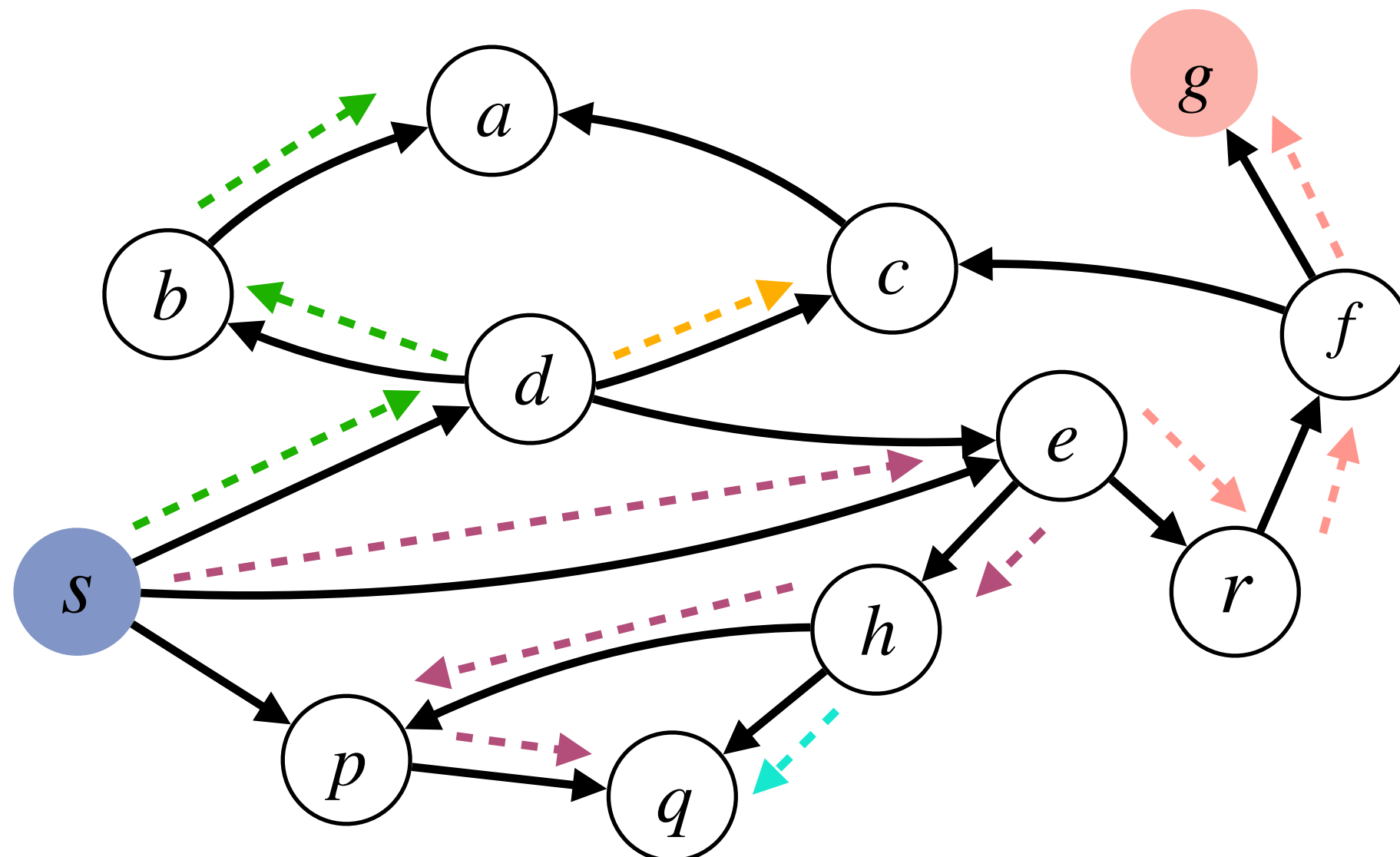
Para espaços de busca muito grandes com estrutura de árvore finita, é comum implementar a DFS sem detecção de ciclos para otimização de memória!

Busca em profundidade (recursiva)

Fronteira é a pilha da recursão

Expandir o nó mais profundo primeiro

- ▶ Nós do primeiro caminho
- ▶ Nós do segundo caminho
- ▶ Nós do terceiro caminho
- ▶ ...



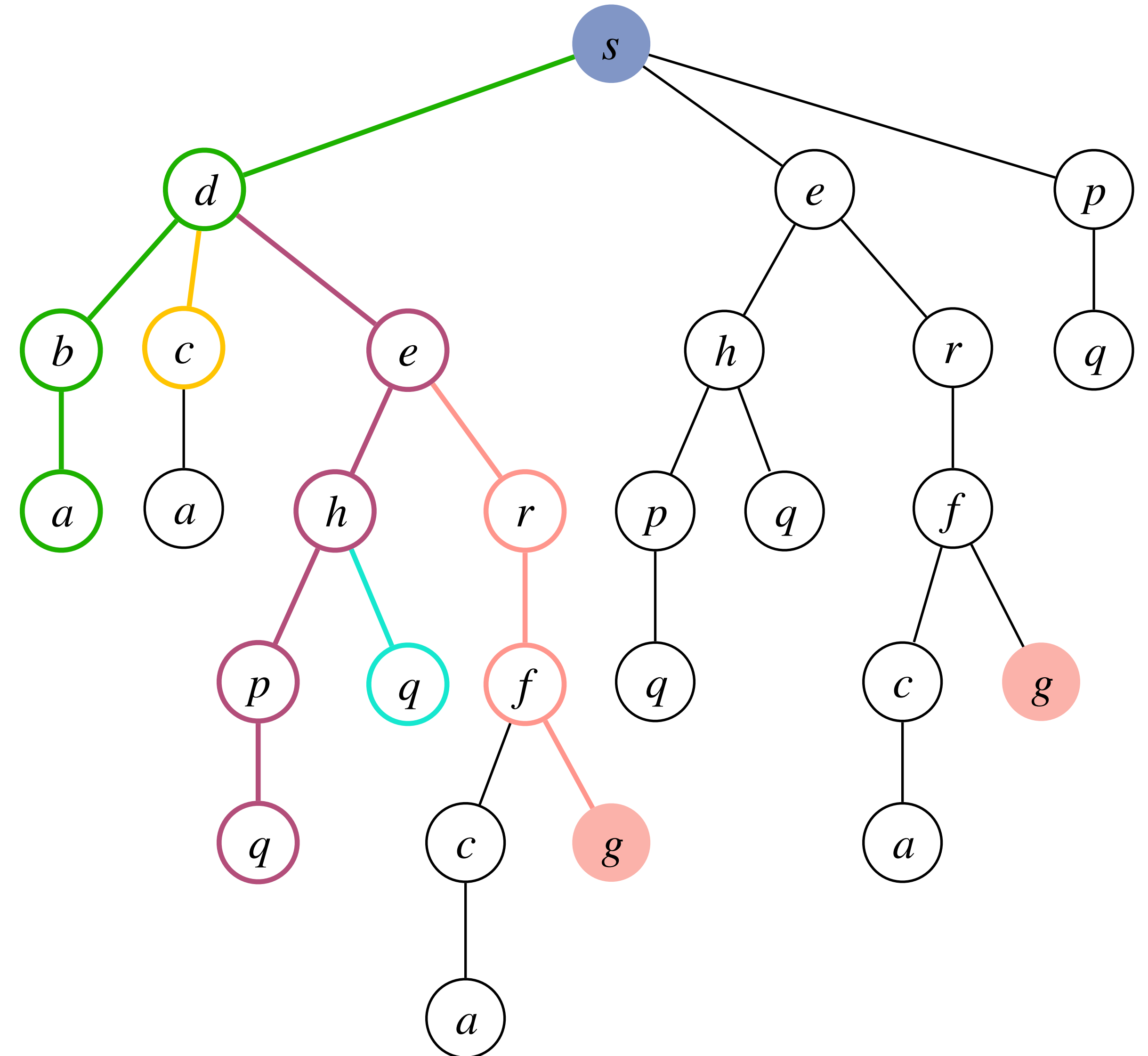
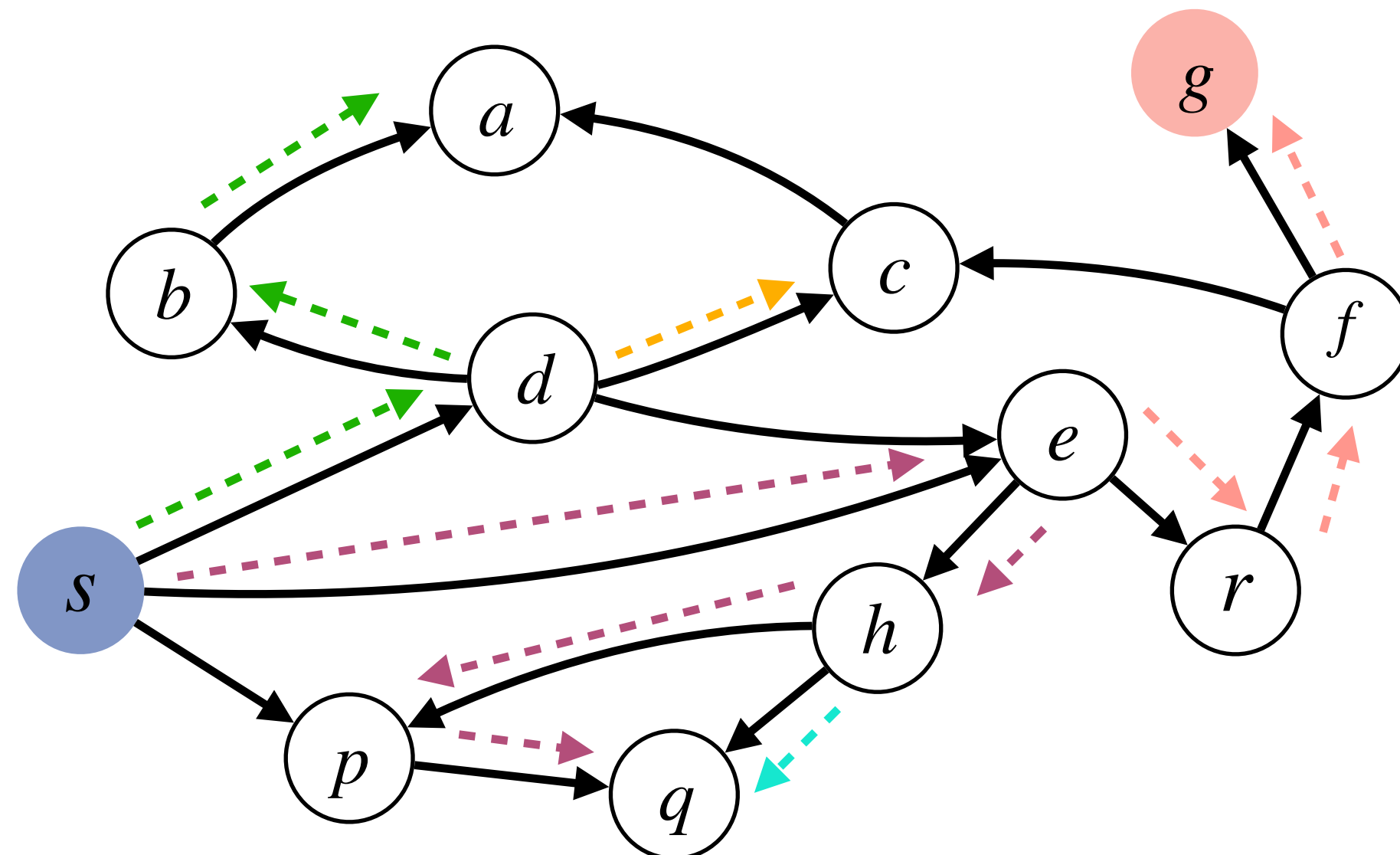
Tempo	Nó	Alcançado
1	s[]	{s}
2	d[s]	{s, d}
3	b[d]	{s, d, b}
4	a[b]	{s, d, b, a}
5	c[d]	{s, d, b, a, c}
6	e[d]	{s, d, b, a, c, e}
7	h[e]	{s, d, b, a, c, e, h}
8	p[h]	{s, d, b, a, c, e, h, p}
9	q[h]	{s, d, b, a, c, e, h, p, q}
10	r[e]	{s, d, b, a, c, e, h, p, q, r}
11	f[r]	
12	g[f]	

Busca em profundidade (recursiva)

Fronteira é a pilha da recursão

Expandir o nó mais profundo primeiro

- ▶ Nós do primeiro caminho
- ▶ Nós do segundo caminho
- ▶ Nós do terceiro caminho
- ▶ ...



Propriedade da busca em profundidade

- ▶ Complexidade de tempo

No pior caso, terá que processar a árvore toda. Seja m a profundidade total (finita) da árvore, a complexidade de tempo é $O(b^m)$

- ▶ Complexidade de espaço

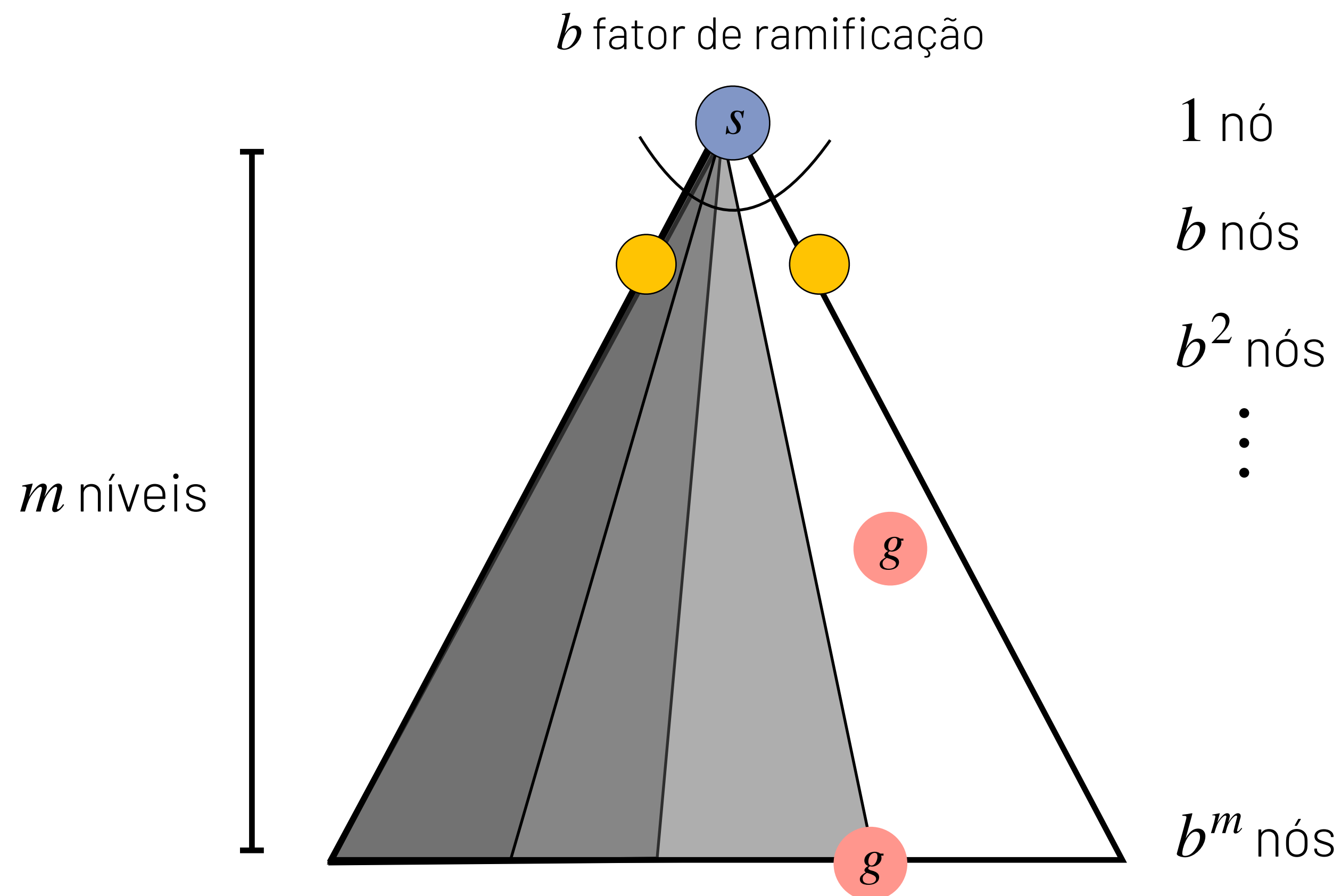
Armazena os irmãos de cada nó no caminho até a raiz, portanto complexidade de espaço é $O(bm)$

- ▶ Completo

Apenas se a árvore de busca for finita (ou usando verificação de ciclos).

- ▶ Ótimo

Não, ele a solução mais à esquerda (recursivo) ou à direita (iterativo), independente de custo



Exercício 1: Quando utilizar busca em profundidade?

Se a DFS não é ótima e completa apenas em árvores finitas, porque utilizar esse algoritmo?

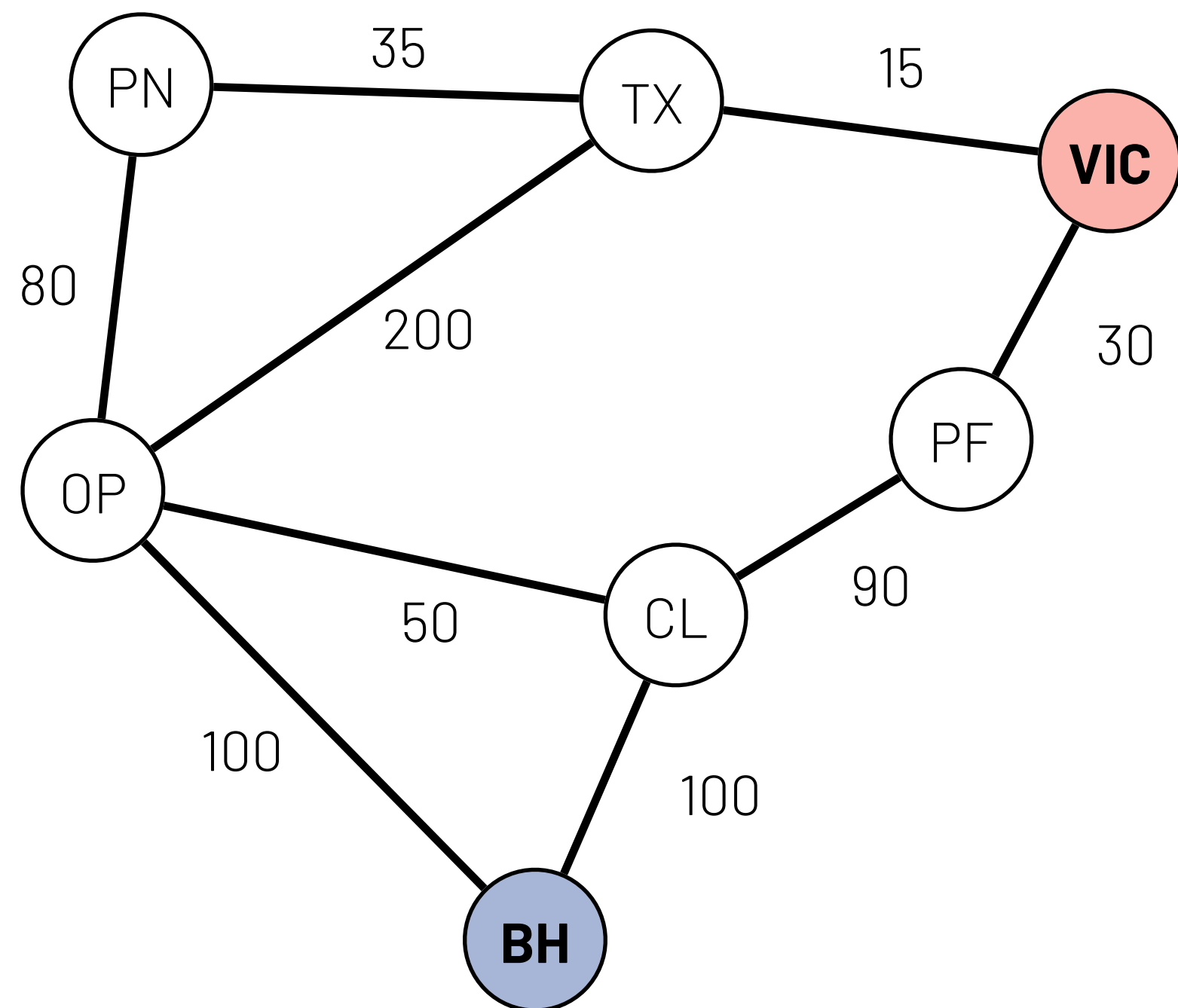
R: Quando o grafo do problema de busca é estruturalmente uma árvore finita, a busca em profundidade é uma excelente solução pois utiliza consideravelmente menos memória que a busca em largura. Exemplos em IA:

- ▶ *Satisfação de restrição*
- ▶ *Satisfatibilidade proposicional*
- ▶ *Programação lógica*

Busca de custo uniforme (Algoritmo de Dijkstra)

Fronteira é uma fila de prioridade

Expandir o nó n com caminho de menor custo $g(n)$



Tempo	Nó	Fronteira (heap)	Alcançado
1	BH(0)	OP(100) , CL(100)	OP(100), CL(100)
2	OP(100)	CL(100) , PN(180), TX(300)	OP(100), CL(100), PN(180), TX(300)
3	CL(100)	PN(180) , TX(300), PF(190)	OP(100), CL(100), PN(180), TX(300), PF(190)
4	PN(180)	TX(300) , PF(190) , TX(215)	OP(100), CL(100), PN(180), TX(215), PF(190)
5	PF(190)	TX(300) , TX(215) , VIC(220)	OP(100), CL(100), PN(180), TX(215), PF(190), VIC(220)
6	TX(215)	TX(300) , VIC(220)	OP(100), CL(100), PN(180), TX(215), PF(190), VIC(220)
7	VIC(220)	TX(300)	OP(100), CL(100), PN(180), TX(215), PF(190), VIC(220)

Propriedade da busca de custo uniforme

- ▶ Complexidade de tempo

Explora todos os nós com custo menor que o da melhor solução C^* . Se as ações custam no mínimo ϵ , então complexidade de tempo $O(b^{C^*/\epsilon})$

- ▶ Complexidade de espaço

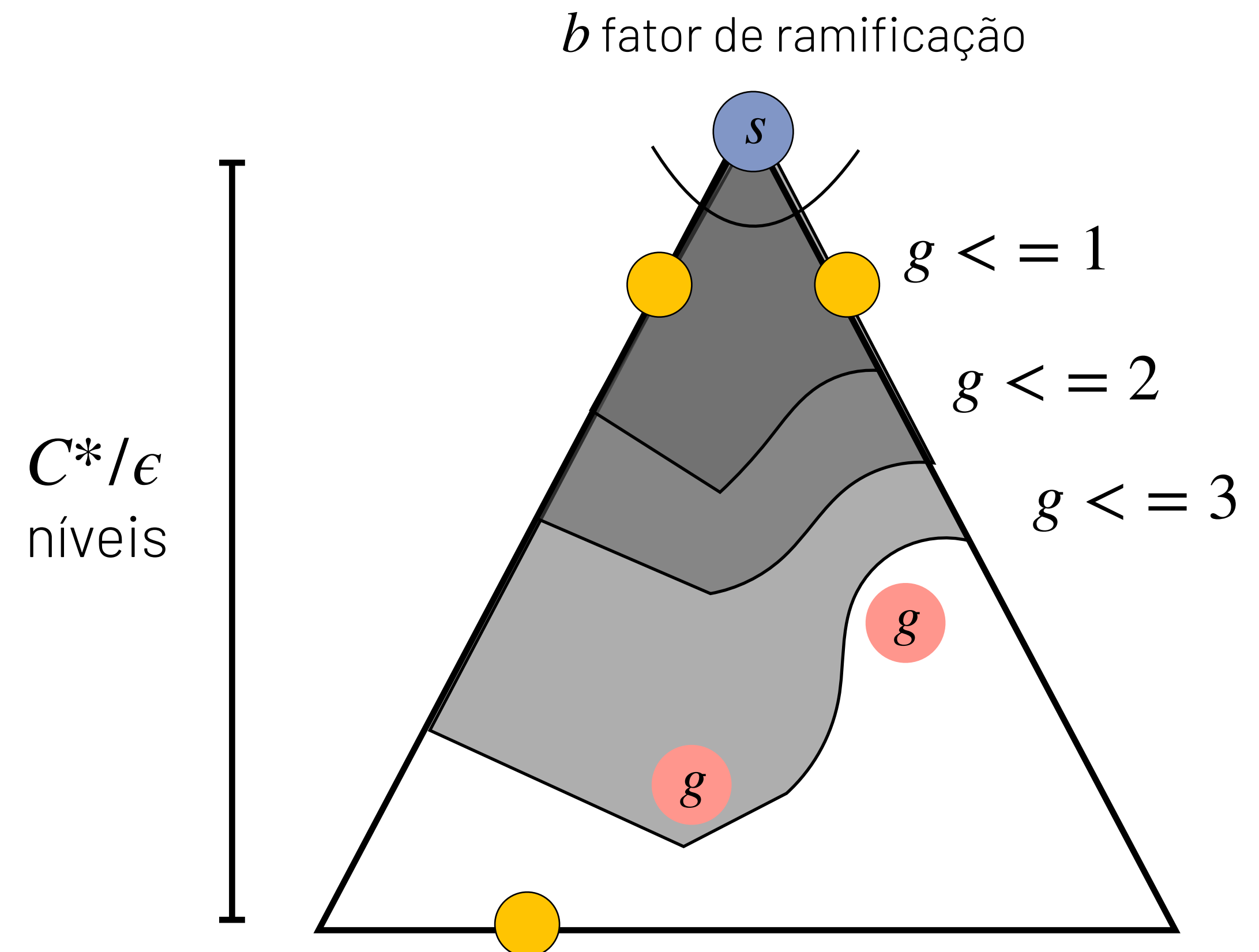
Armazena aproximadamente todos os nós dos C^*/ϵ níveis até a solução ótima C^* , portanto complexidade de espaço também é $O(b^{C^*/\epsilon})$

- ▶ Completo

Assumindo que C^* é finito e $\epsilon > 0$, sim!

- ▶ Ótimo

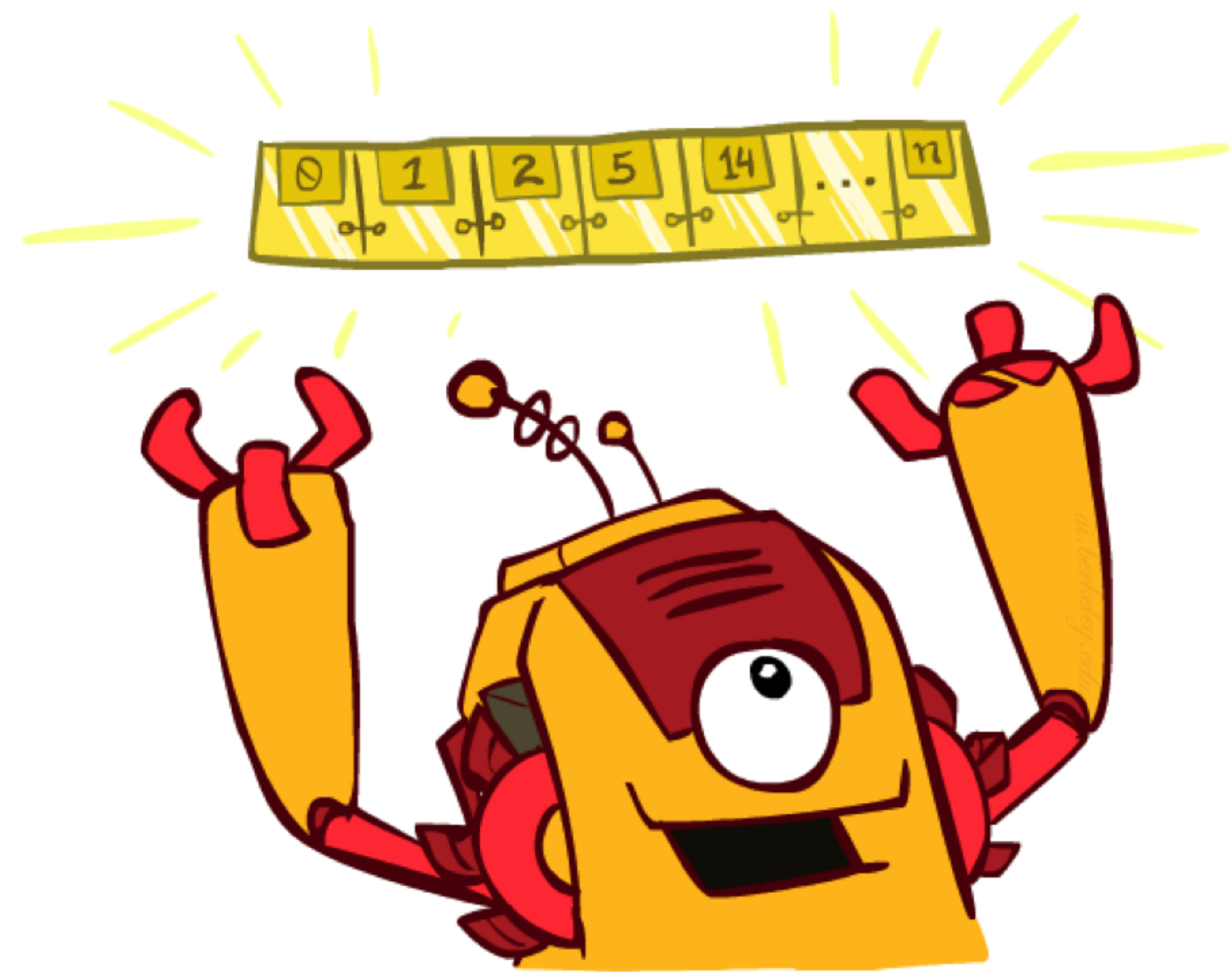
Sim! Referência para a prova em anexo.



Estruturas de dados de busca

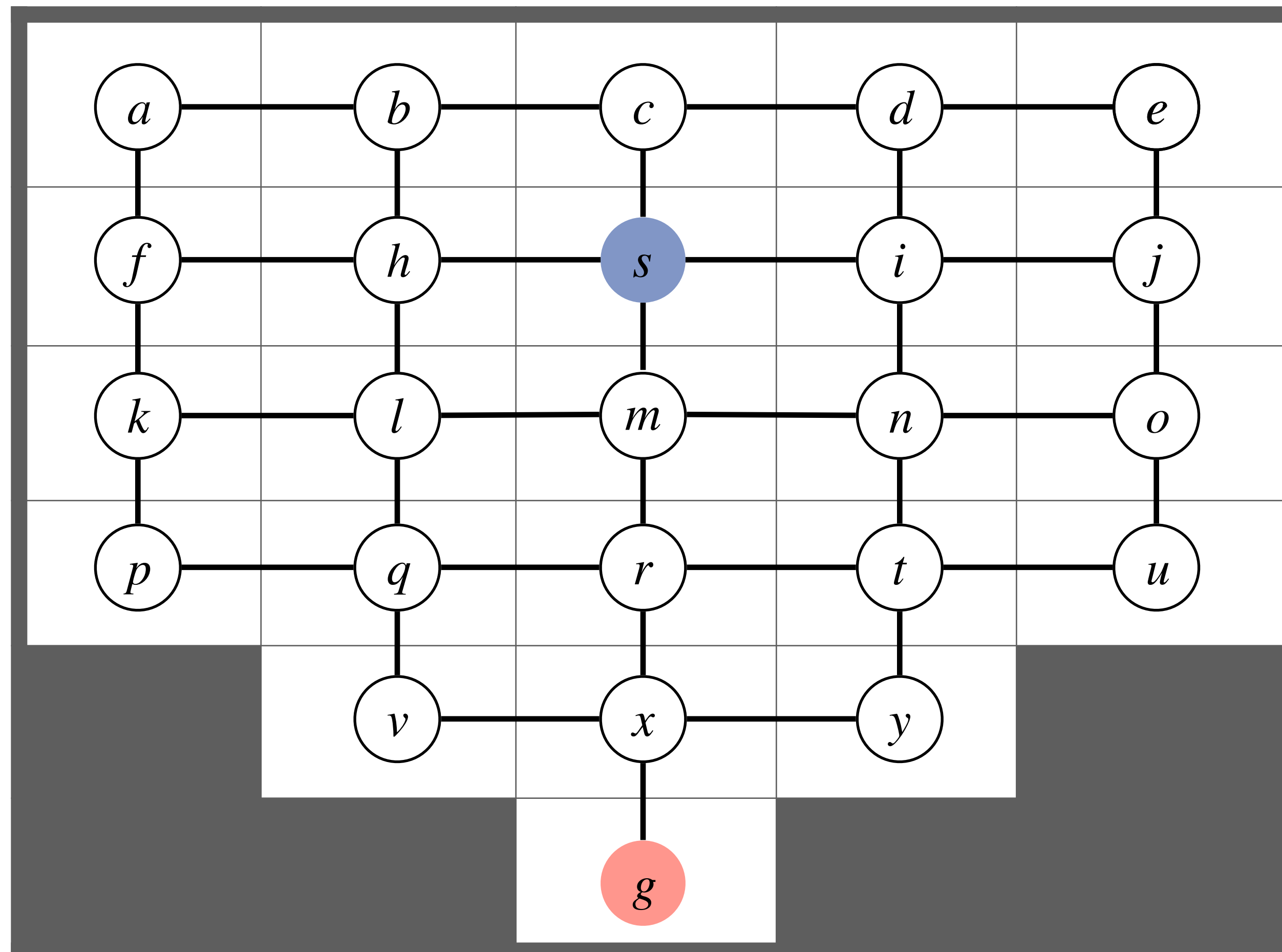
Todos os algoritmos de busca são os mesmos. O que muda são as estratégias de fronteira.

- ▶ Conceitualmente, todas as fronteiras são filas de prioridade
- ▶ Na prática, na BFS e DFS podemos evitar o custo $\log(n)$ da fila de prioridades utilizando uma fila e uma pilha, respectivamente
- ▶ É possível implementar todos os algoritmos em uma única função, passando o objeto da fronteira como parâmetro



Problema da busca de custo uniforme

Considere a busca de custo uniforme no seguinte problema de busca de caminho mais curto em um grid (todas as ações tem custo 1):



O algoritmo busca igualmente em todas as direções.

Como evitar expandir estados claramente não promissores?

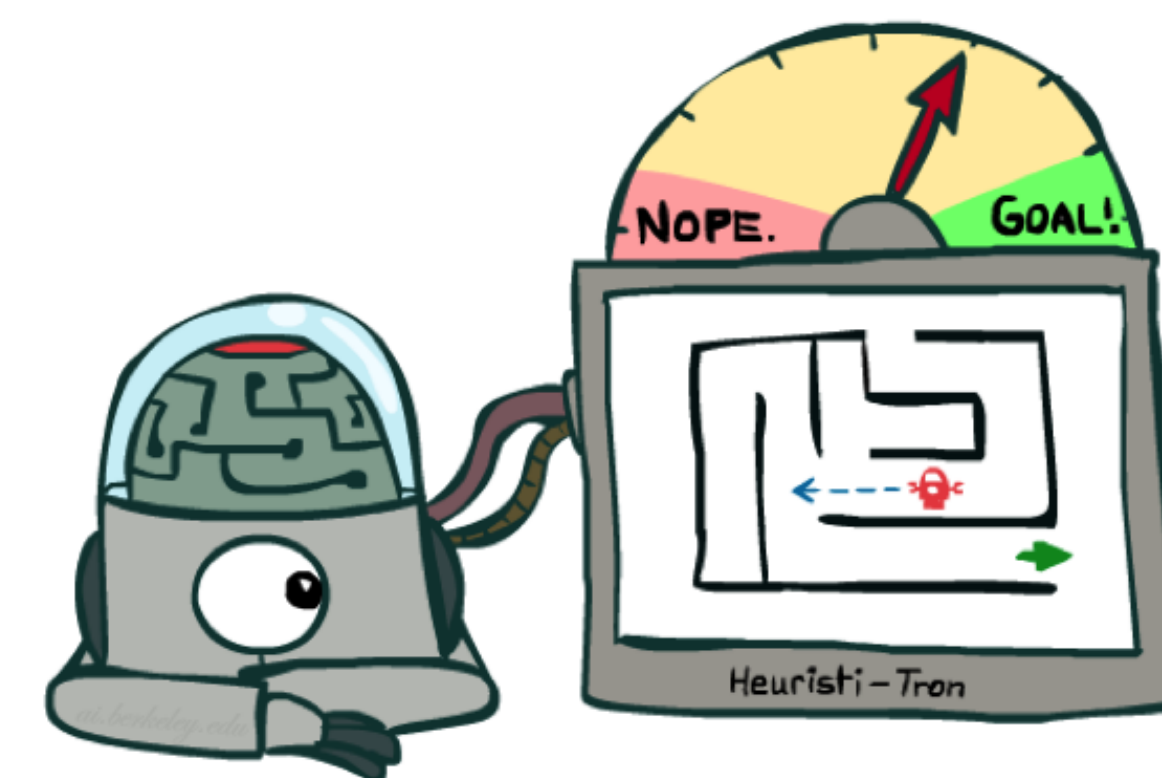
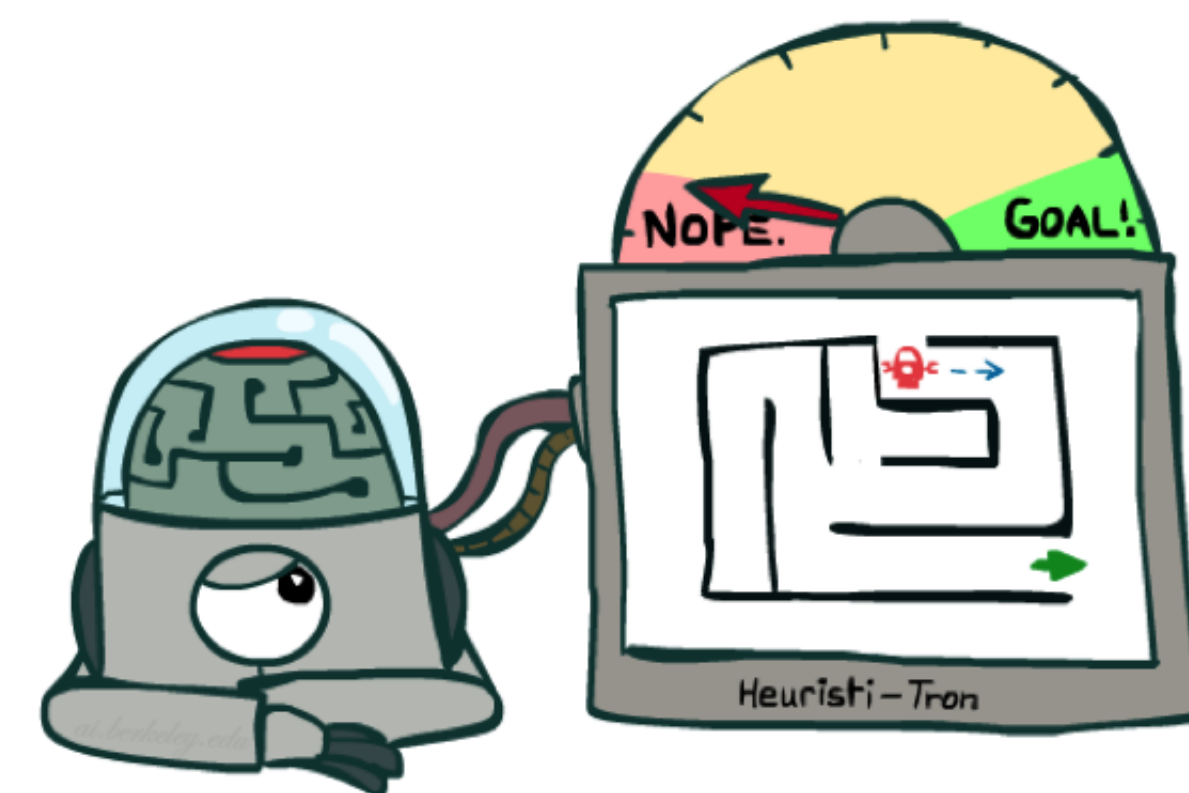
Função Heurística

Uma **função heurística** $h(n) : S \rightarrow \mathbb{R}^+$ recebe como entrada um estado n e retorna uma estimativa da distância entre n e g .

- ▶ São definidas de maneira particular para cada problema de busca
- ▶ Por exemplo, para o problema de caminho mais curto:

- ▶ **Distância Manhattan:** $h(n) = |x_n - x_g| + |y_n - y_g|$

- ▶ **Distância Euclidiana:** $h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}$

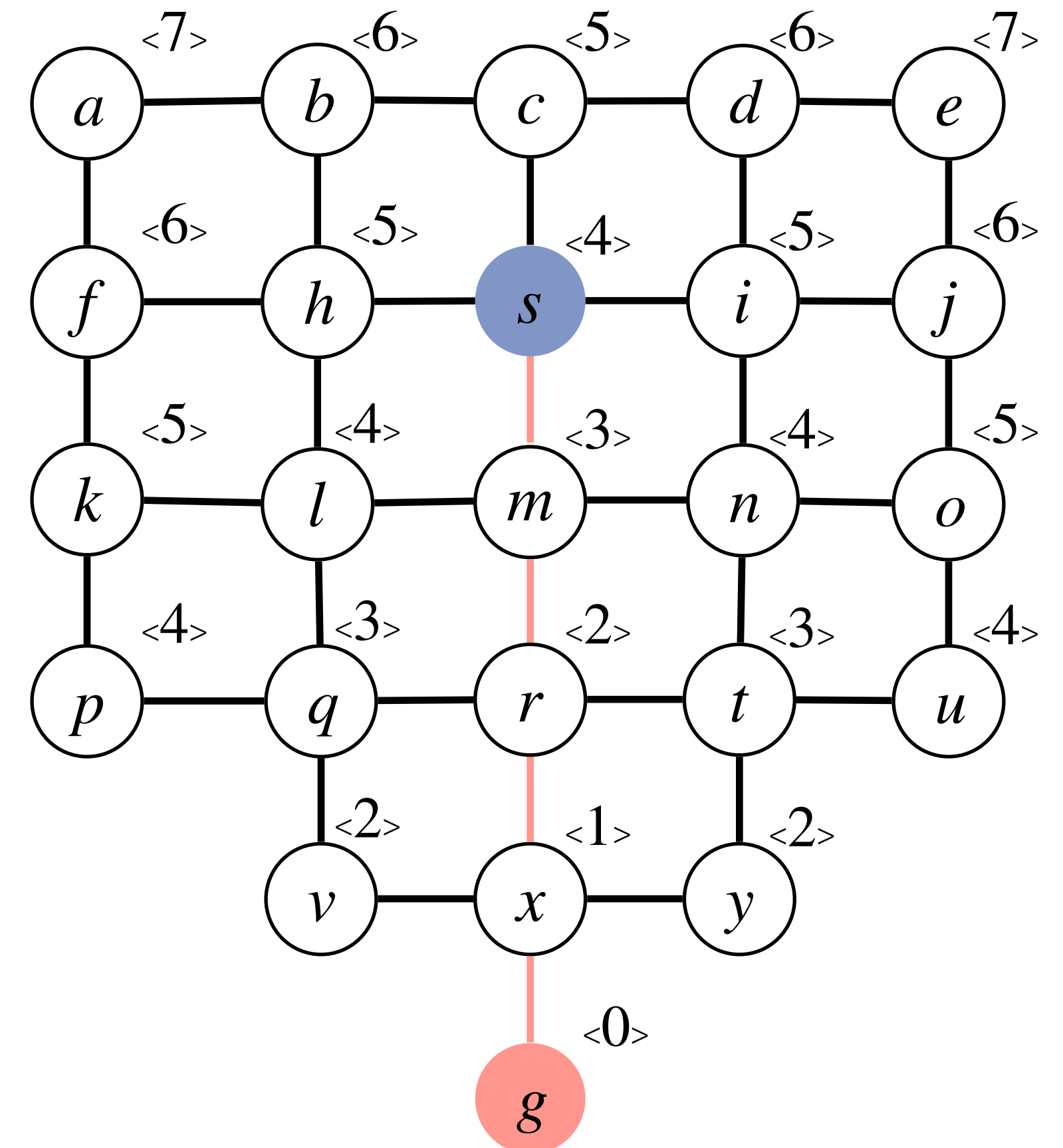


Algoritmo de busca gulosa pela melhor escolha

Fronteira é uma fila de prioridade

Expandir o nó n que parece estar mais próximo do estado final – aquele com menor $h(n)$

- ▶ Nesse problema, o algoritmo de busca gulosa pela melhor escolha retorna a solução ótima **S-M-R-X-G** da forma mais rápida possível – explorando apenas os nós da **solução ótima**



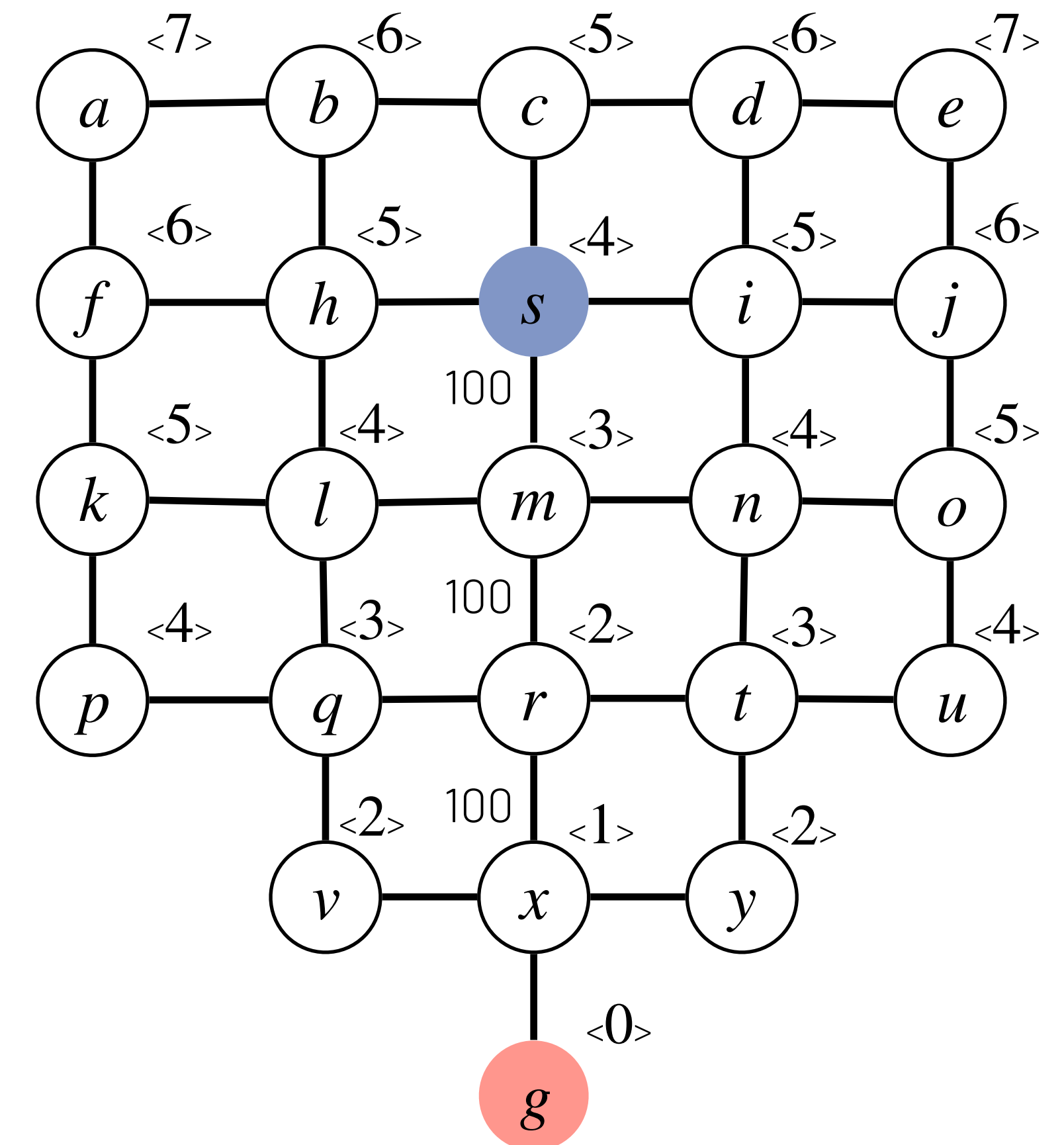
Algoritmo de busca gulosa pela melhor escolha

Fronteira é uma fila de prioridade

Expandir o nó n que parece estar mais próximo do estado final – aquele com menor $h(n)$

Exercício: Você consegue imaginar um problema onde esse algoritmo não seria ótimo?

R: O algoritmo de busca gulosa pela melhor escolha não considera o custo $g(n)$ dos estados. Nesse problema, ele retorna a mesma solução **S-M-R-X-G** mesmo ela sendo a pior!



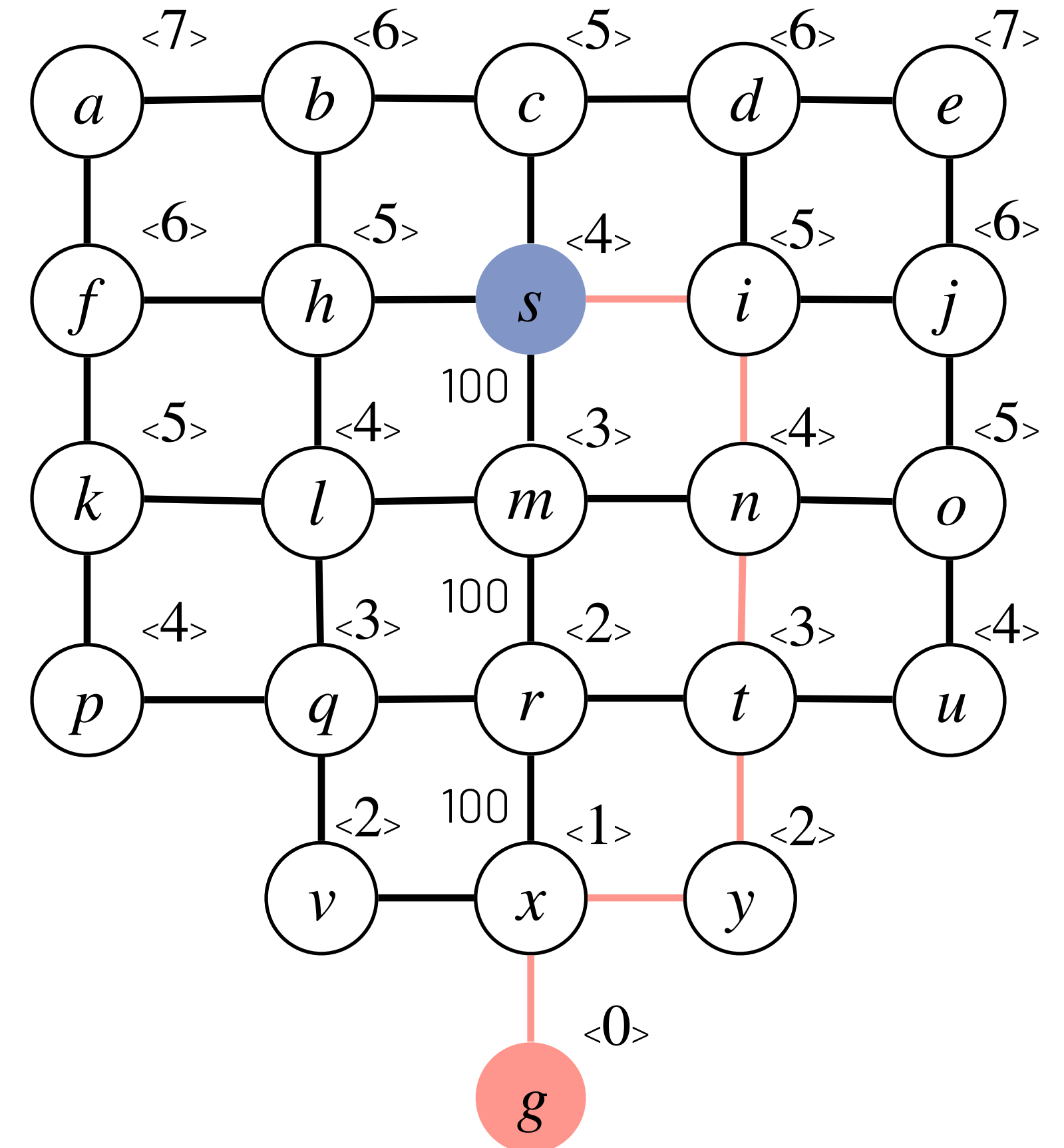
Algoritmo A*

Fronteira é uma fila de prioridade

Expandir o nó n que parece mais próximo com caminho de menor custo – aquele com menor $f(n) = g(n) + h(n)$

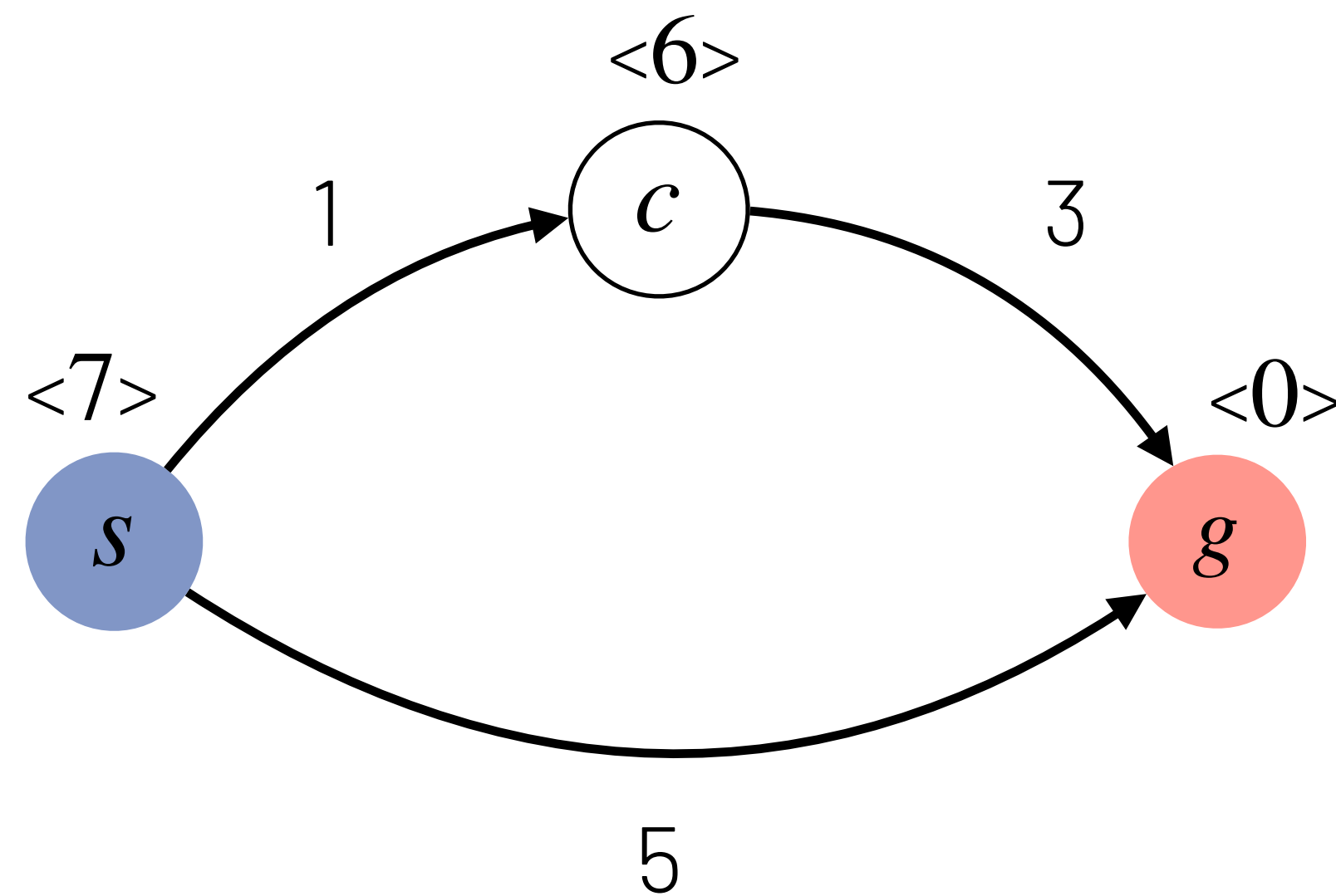
A* pode ser visto como a combinação da **busca de custo uniforme** com a **busca gulosa por melhor escolha**:

- ▶ **Custo uniforme**: ordena por custo do caminho $g(n)$
- ▶ **Melhor escolha**: ordena pela função heurística $h(n)$



Exercício: o algoritmo A^* é ótimo?

A solução encontrada pelo A^* no grafo abaixo é ótima? Execute o algoritmo e mostre a árvore de nós expandidos.



Qual o problema com essa função heurística?

$h(n) > h^*(n)$, onde $h^*(n)$ é o custo ótimo entre n e g

Essa função heurística não é **admissível!**

Função heurística admissível

Uma função heurística h é **admissível** se:

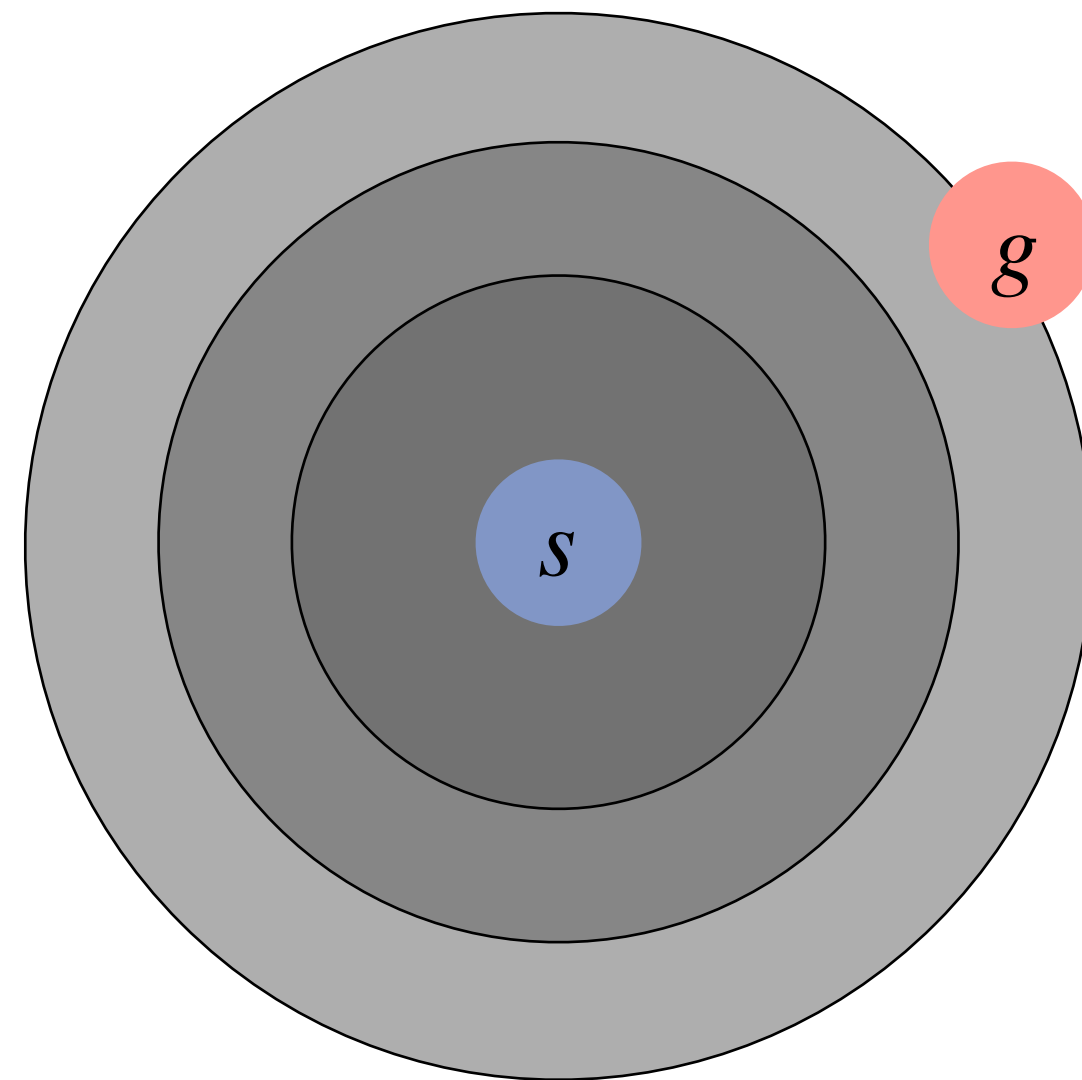
$$0 \leq h(n) \leq h^*(n), \text{ onde } h^*(n) \text{ é o custo ótimo entre } n \text{ e } g$$

O algoritmo A* é ótimo apenas se h for admissível!

A maior parte do trabalho na resolução de problemas difíceis de busca consiste em encontrar heurísticas admissíveis.

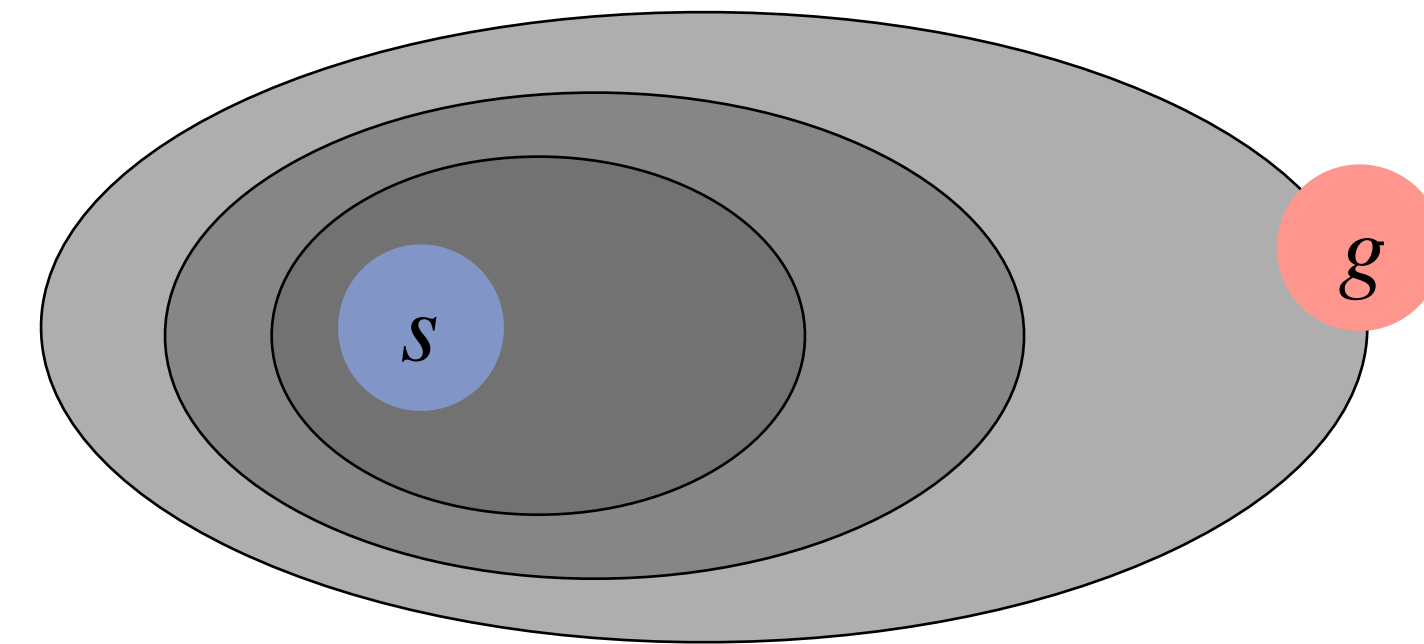
Contornos de busca

Busca de custo uniforme



Expande igualmente em todas as direções.

Algoritmo A*



Expande principalmente em direção ao estado final, mas protege suas apostas para garantir a otimização

Próxima aula

A4: Busca local e otimização I

Formalização e exemplos de problemas de busca local. Algoritmos subida de encosta, têmpera simulada e busca em feixe.