DCC192

2025/1

UFMG

Desenvolvimento de Jogos Digitais

A21: IA — Pathfinding II

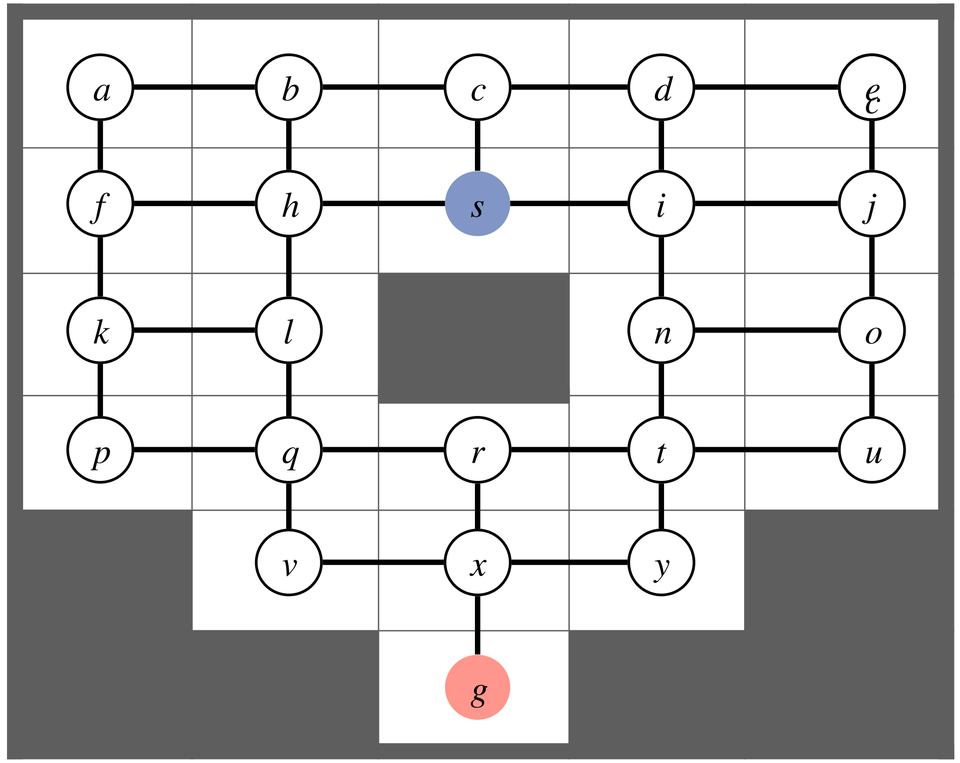
Prof. Lucas N. Ferreira

Plano de aula

- Representação de Mapas em Jogos
- Algoritmos de Busca Não-Informados
  - Busca em profundidade
  - Busca em largura
- **Algoritmos de Busca Informados**
  - **Heurísticas**
  - **Greedy Best-First Search**
  - **A***

Pathfinding: Espaços de Estados

As posições dos objetos em jogos constuman ser contínuas. Precisamos discretizar esse espaço para fazer buscas eficientes. A técnica mais comum é utilizar um grid:



- Conjunto de estados S: cada célula livre é um estado
- **Estado inicial** $s \in S$: a coordenada (i, j) atual da unidade
- **Estado final** $g \in S$: a coordenada destino clicada pelo jogador
- **Função de ações** $A(s)$: direções (up, down, left, ...) das coordenadas vizinhas de s que estão livres
- **Modelo de transição** $T(s, a)$: a célula s' vizinha de s alcançada pela movimentação na direaçao de a
- **Função custo de ação** $C(s, a, s')$: um valor real pré-definido de se caminhar na grama vs. nas pedras (opcional)

Algoritmos de busca (que estudaremos)

- **Busca sem informação**

  Não possuem informação sobre a distância entre um determinado estado s 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**

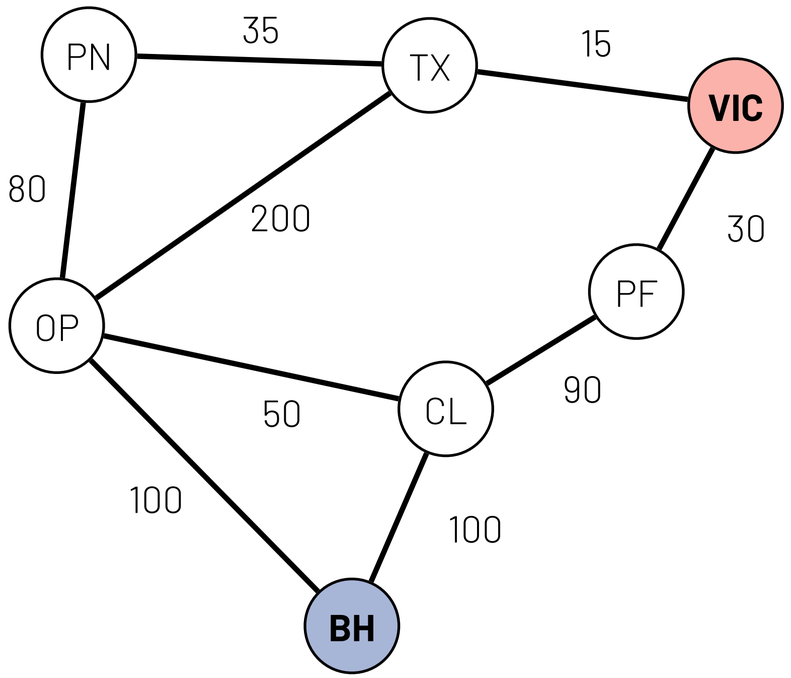
  Possuem informação sobre a distância entre um determinado nó e o estado final

  - Busca gulosa de melhor escolha
  - Algoritmo A*

Busca de custo uniforme (Algoritmo de Dijkstra)

Fronteira é uma fila de prioridade

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

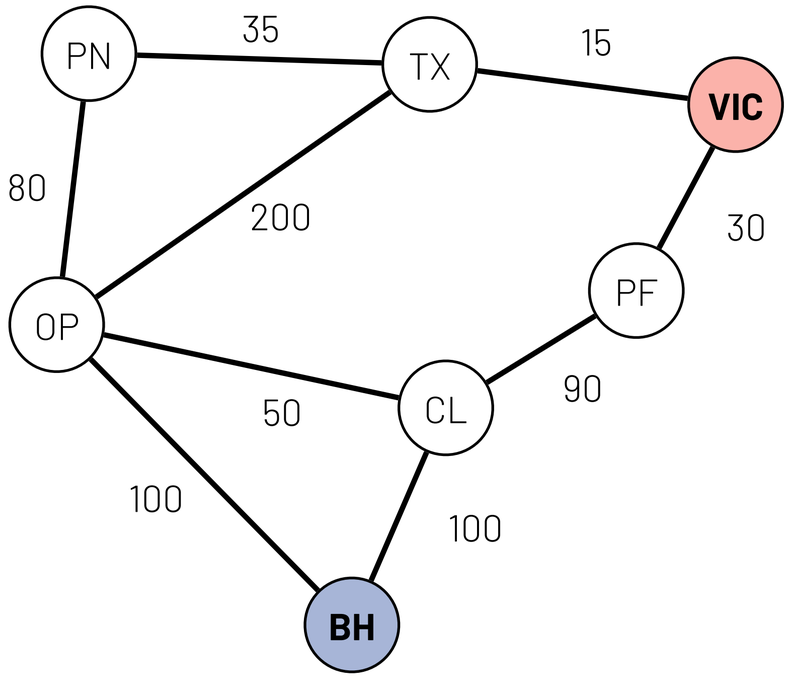


Tempo	Nó	Fronteira (heap)	Alcançado
1			
2			
3			
4			
5			
6			
7			

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

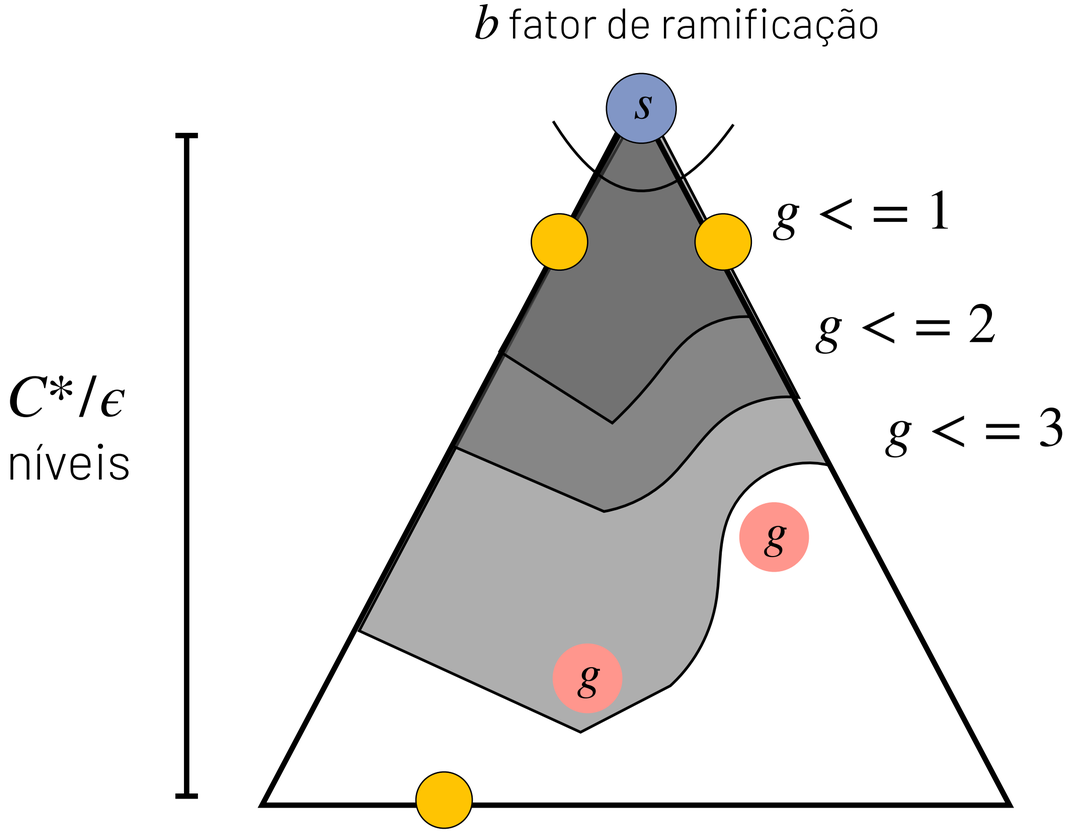
  Armazena aproximadamente todos os nós dos C^*/ϵ níveis até a solução ótima C^*, portanto complexidade de espaço também é $O(b^{C^*/\epsilon})$

- Completo

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

- Ótimo

  Sim! Referência para a prova em anexo.

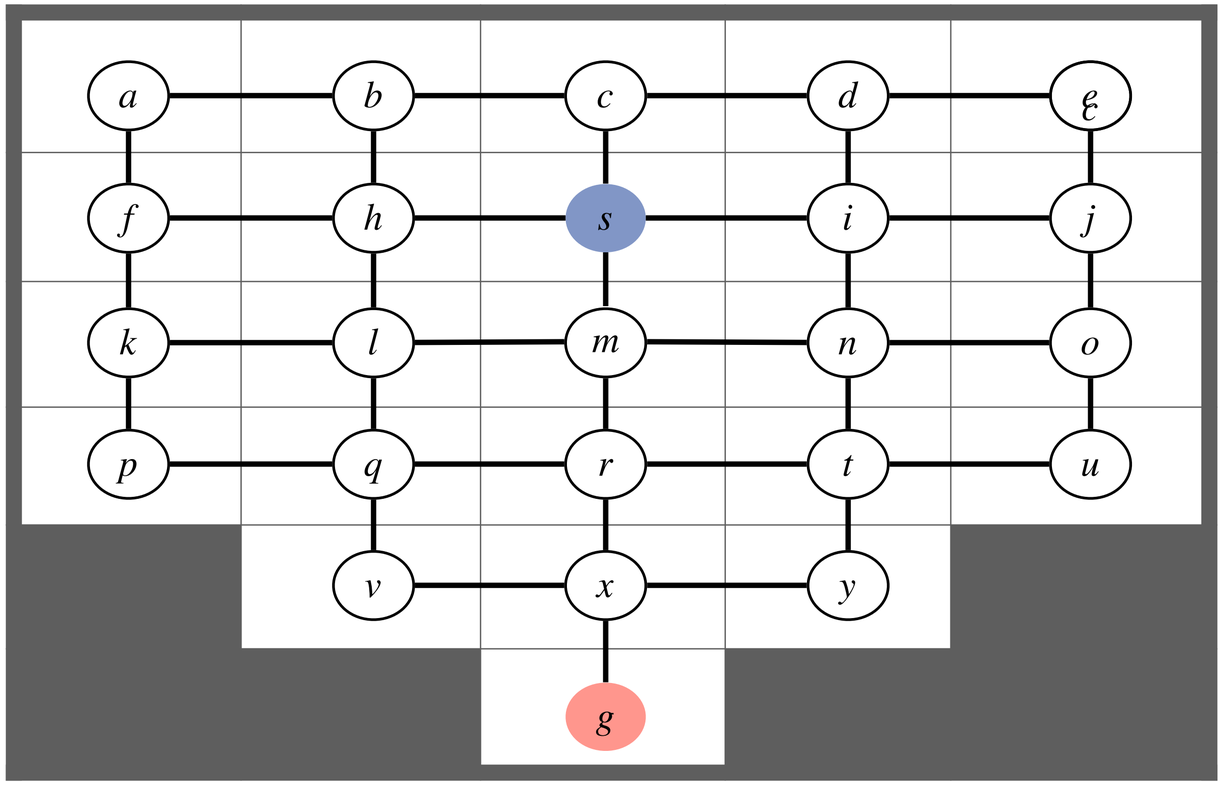


Busca de Custo Uniforme

```
def UCS(s, g, A, T, C):
 1. heap = [(s, 0)]
 2. alcancado = {s}
 3. custo[s] = 0
 4. while heap não estiver vazia:
 5.     n = heap.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.            heap.push((filho, custo_filho))
 12.            alcancado.append(filho)
 13.            custo[filho] = custo_filho
```

Problema da Busca de Custo Uniforme

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



UCS busca igualmente em todas as direções.

Como evitar expandir estados claramente não promissores?

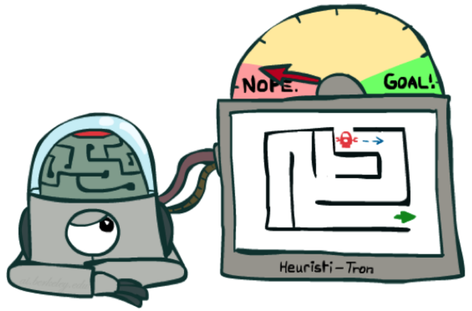
Função Heurística

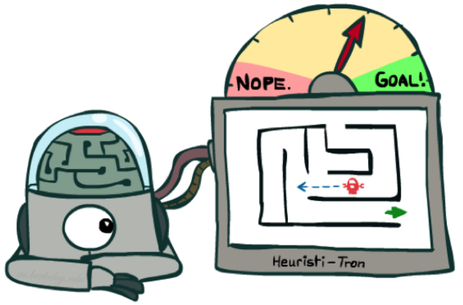
Uma **função heurística** $h(n) : S \rightarrow \mathbb{R}^+$ recebe como entrada um estado n e retorna uma estimativa da distância entre n e g.

- São definidas de maneira particular para cada problema de busca
- Por exemplo, para o problema de encontrar caminhos:

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

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



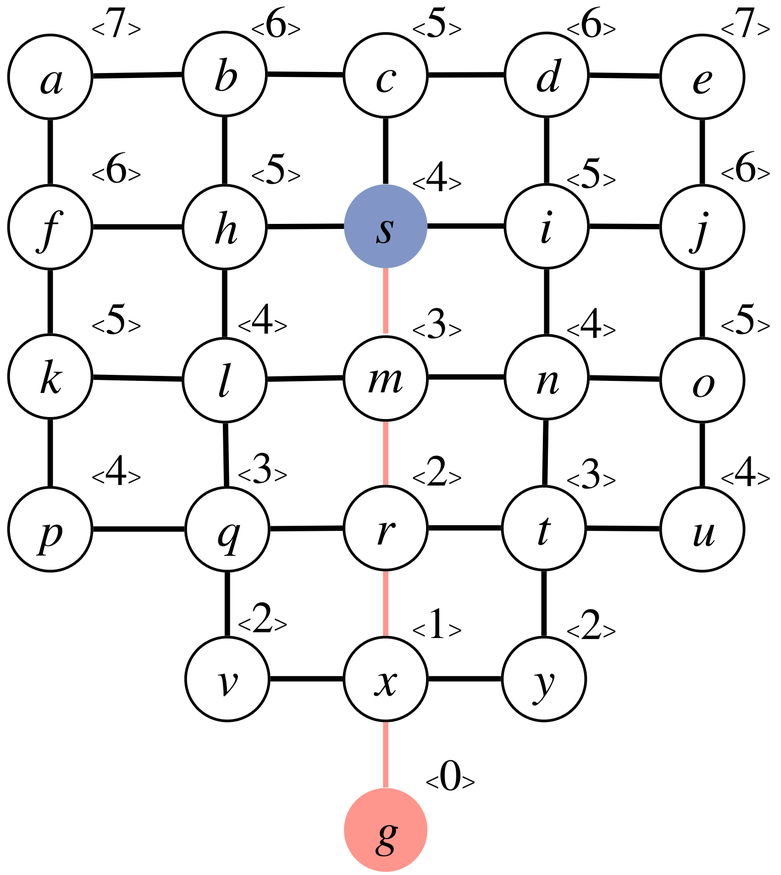


Busca Gulosa de Melhor Escolha

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**



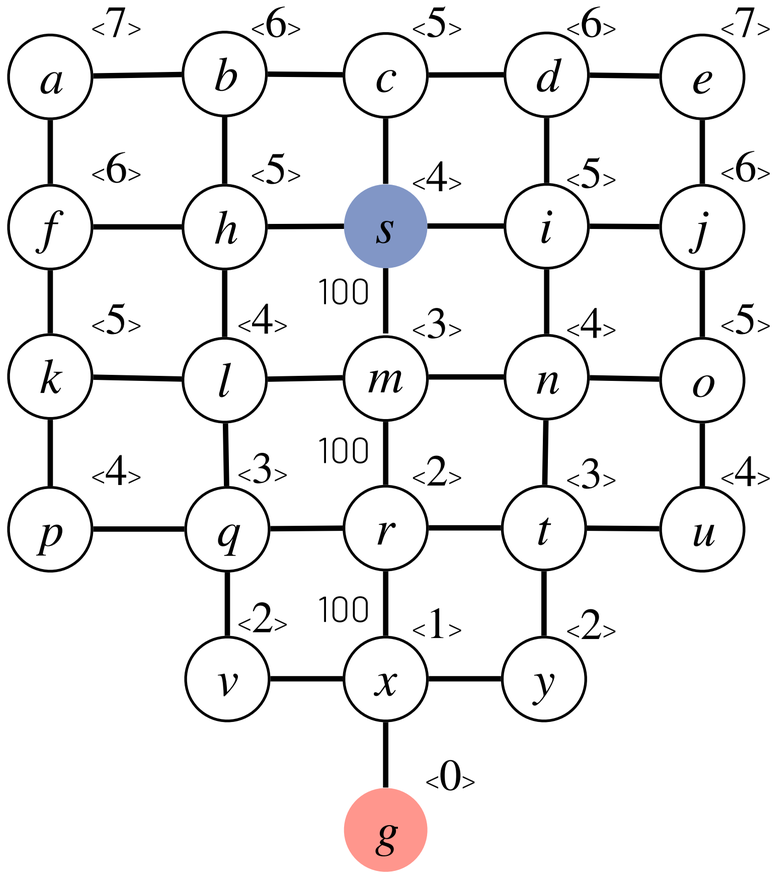
Busca Gulosa de Melhor Escolha

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!*



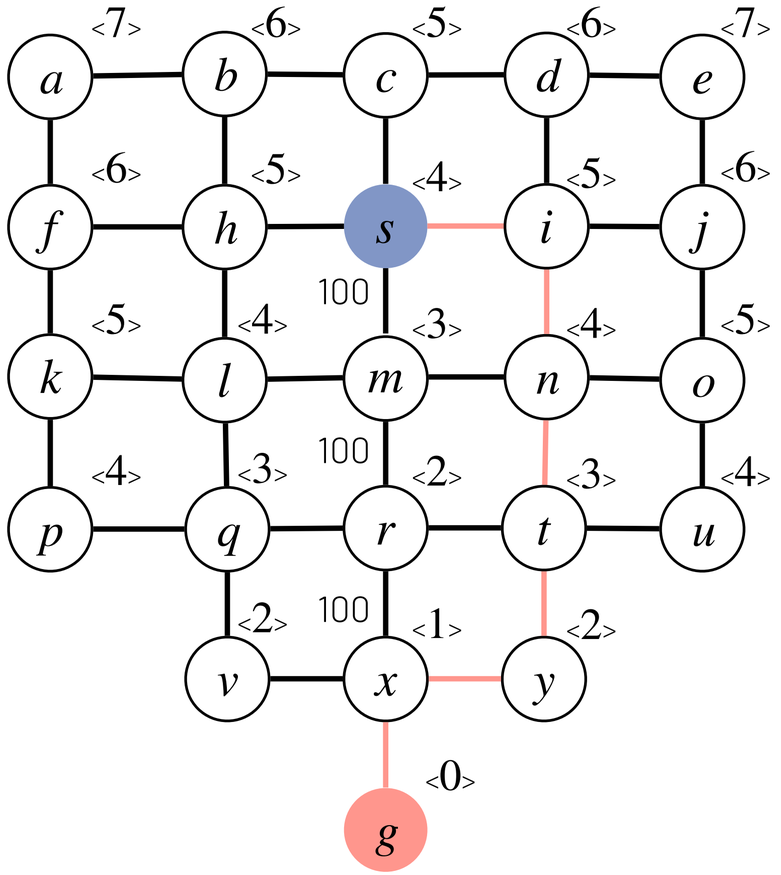
Algoritmo A*

Fronteira é uma fila de prioridade

Expandir o nó n com caminho de menor custo que parece mais próximo pela heurística — 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 $h(n)$

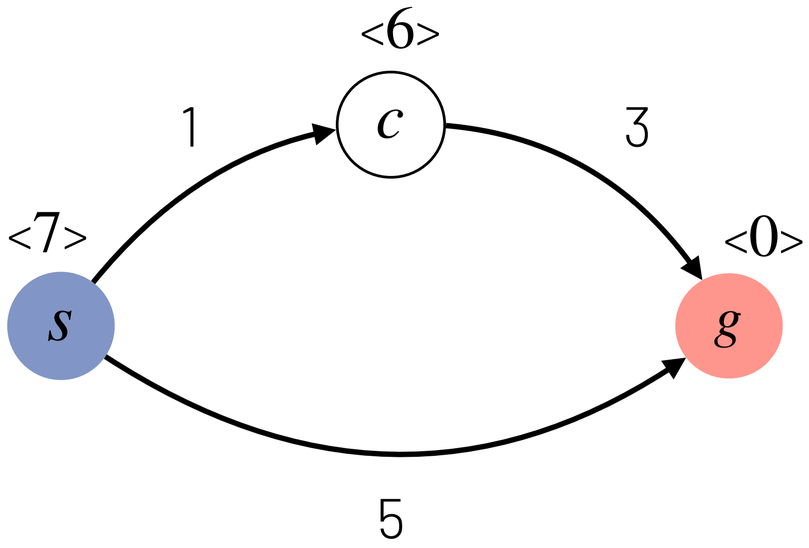


A*

```
def A_star(s, g, A, T, C, h):
 1. heap = [(s, h(s, g))]
 2. alcancado = {s}
 3. custo[s] = 0
 4. while heap não estiver vazia:
 5.     n = heap.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.            heap.push((filho, custo_filho + h(filho, g)))
 12.            alcancado.append(filho)
 13.            custo[filho] = custo_filho
```

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:

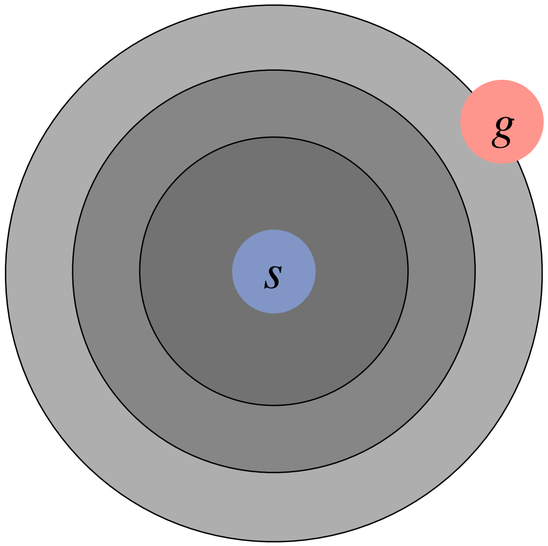
$$0 \leq h(n) \leq h^*(n), \text{ onde } h^*(n) \text{ é o custo ótimo entre } n \text{ e } g$$

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.

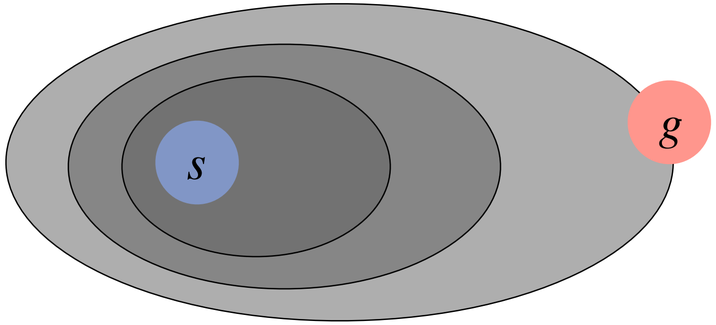
Contornos de Busca

Busca de custo uniforme



Expande igualmente em todas as direções.

Algoritmo A*



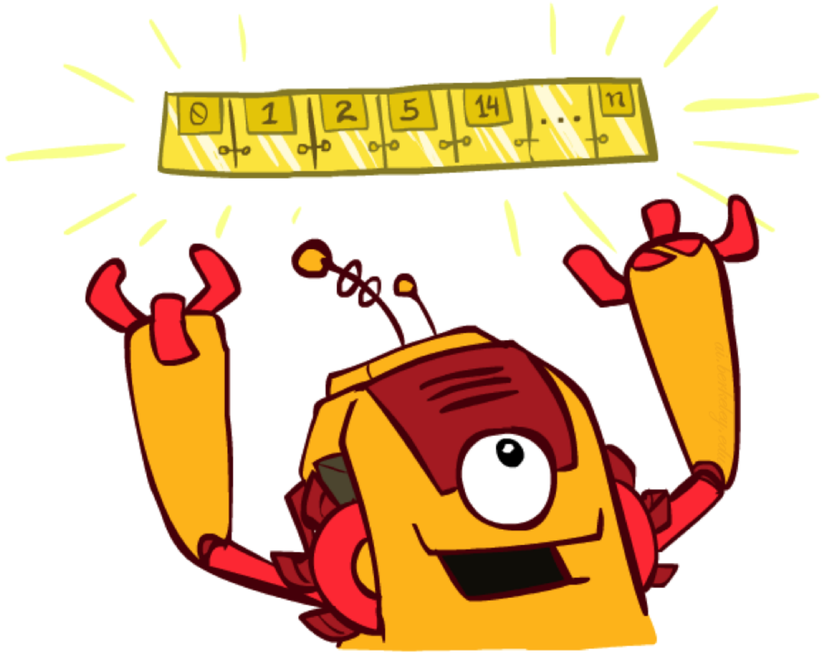
Expande principalmente em direção ao estado final, mas protege suas apostas para garantir a otimização

https://www.redblobgames.com/pathfinding/a-star/introduction.html

Estruturas de Dados de Busca

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

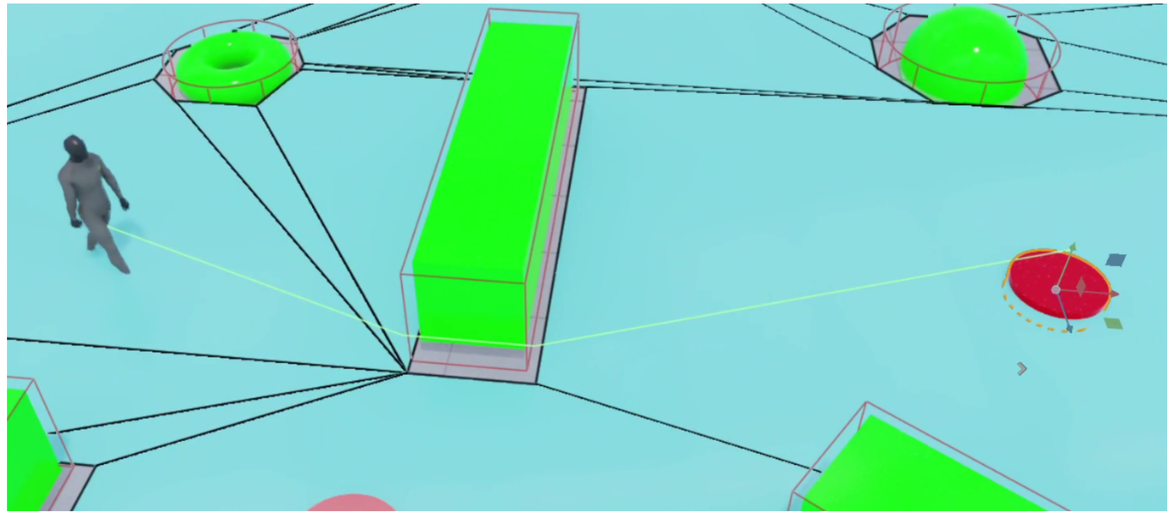
- Conceitualmente, todas as fronteiras são filas de prioridade
- Na prática, na BFS e DFS podemos evitar o custo $log(n)$ da fila de prioridades utilizando uma fila e uma pilha, respectivamente
- É possível implementar todas os algoritmos em uma única função, passando o objeto da fronteira como parâmetro

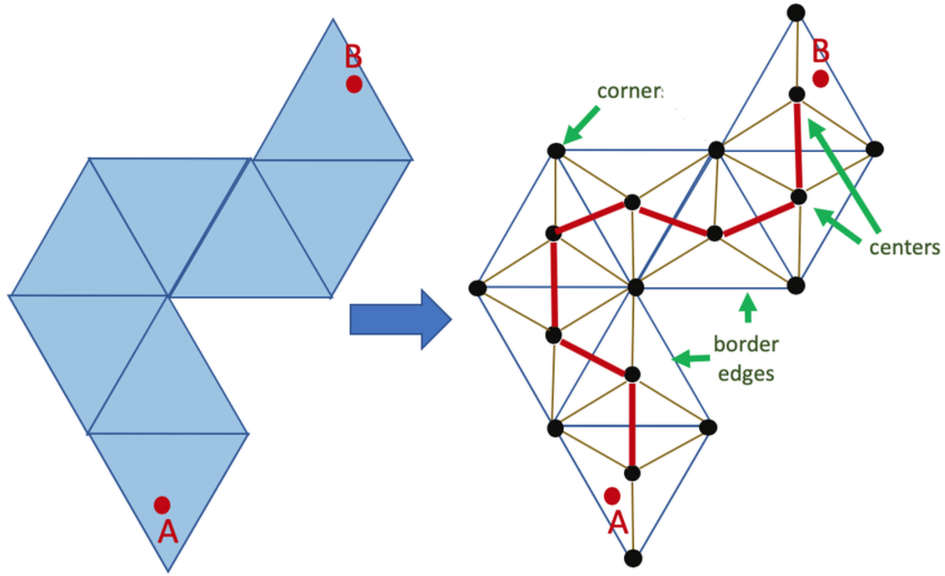


NavMesh: Malha de Navegação

Muitas vezes um grid não é sucifiente para representar bem o espaço de navegação do jogo. Uma alternativa é utilizar uma malha de navegação:

- Cada triângulo é um vértice no grafo e triângulos vizinhos possuem uma aresta entre si
- Dado um ponto qualquer na malha, temos que calcular em que vértice ele percence
- Os algoritmos de busca funcionam da mesma forma que na representação em grade





Próxima aula

A20: IA — Pathfinding II

- Representação de Mapas em Jogos
- Algoritmos de Busca Não-Informados
  - Busca em profundidade
  - Busca em largura
- Algoritmos de Busca Informados
  - Heurísticas
  - Greedy Best-First Search
  - A*