

AULA 03

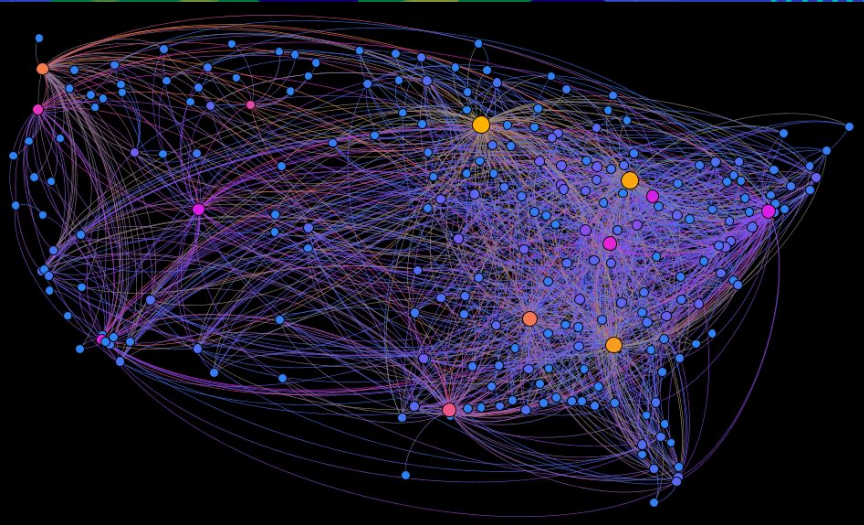
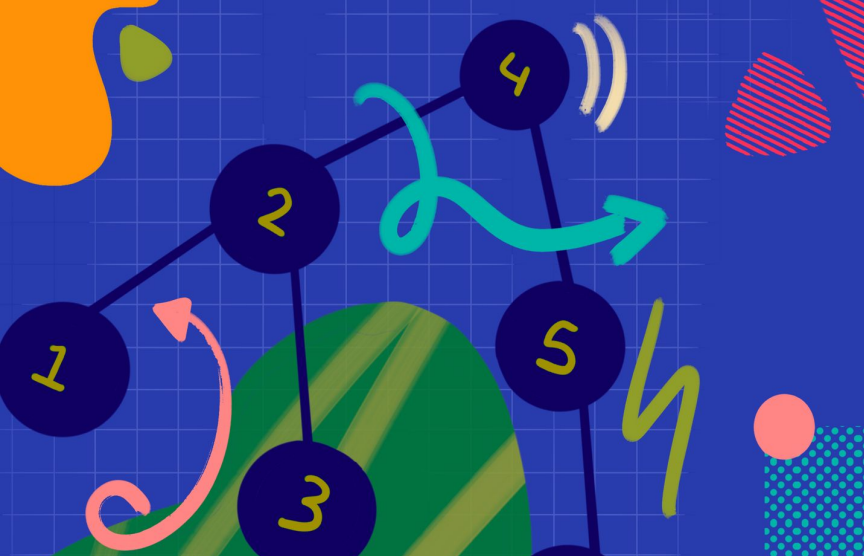
Métodos de Busca Com Informação

CAMILA LARANJEIRA

mila.laranjeira@gmail.com



PUC Minas



Agenda

- Heurística
- Métodos de Busca
 - Gulosa (greedy)
 - A^*
 - IDA^*
 - RBFS

Sobre os slides

Esses slides usam material de:

- José Augusto Baranauskas do Departamento de Computação e Matemática – FFCLRP-USP
- Lecture 3: Informed Search: A* and Heuristics | Berkeley CS188: AI (Spring 2022)
 - <https://inst.eecs.berkeley.edu/~cs188/sp22/assets/slides/Lecture3.pdf>

Função de Custo

Até então, trabalhos com o custo como algo pré-definido.

- Função de custo $f(n)$

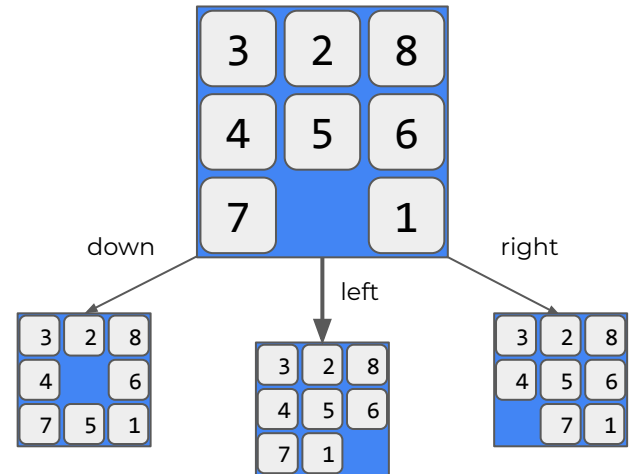
$$f(n) = g(n)$$

- Com $g(n)$ sendo o custo fixo de uma ação

```
3 class MagicTrain():
4     def __init__(self, goal, cost_walk=0, cost_train=0):
5         self.initial_state = 1
6         self.goal = goal
7         self.actions = {('andar', cost_walk), ('trem mágico', cost_train)}
8
9     def isEndState(self, state):
10        if state == self.goal: return True
11        return False
12
13    def makeMove(self, state, action):
14        name, cost = action
15        if name == 'andar': return (name, cost, state+1)
16        elif name == 'trem mágico': return (name, cost, state * 2)
17
18    def getValidMoves(self, state):
19        valid_moves = []
20        for action in self.actions:
21            name, cost, new_state = self.makeMove(state, action)
22
23            # Restrição do problema
24            if new_state <= self.goal:
25                valid_moves.append( (name, cost, new_state) )
26
27        return valid_moves
```

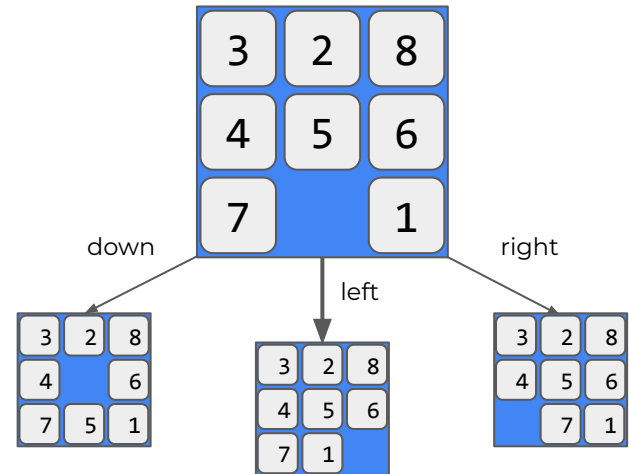
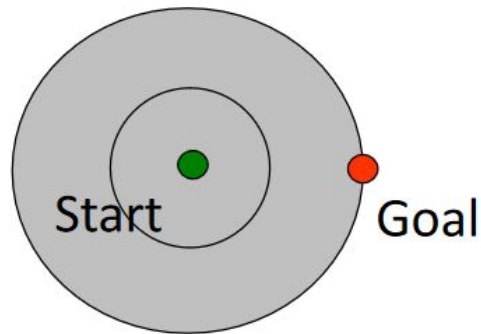
Função de Custo

- Por exemplo, no problema do quebra-cabeça-8 definimos o custo constante $c = 1$ para cada movimento. A busca cega encontra a solução com menor número de movimentos.
 - $f(n) = g(n) = 1$
- Mas como será feita a exploração pela busca cega?



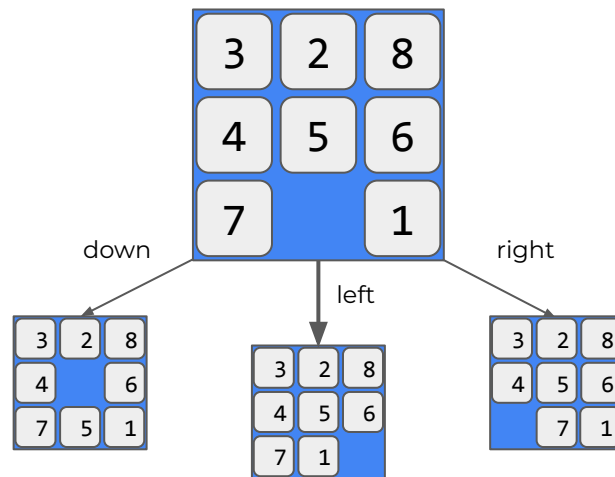
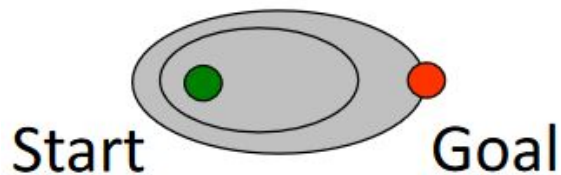
Função de Custo

- Por exemplo, no problema do quebra-cabeça-8 definimos o custo constante $c = 1$ para cada movimento. A busca cega encontra a solução com menor número de movimentos.
 - $f(n) = g(n) = 1$
- Todas as ações são igualmente promissoras
- A busca explora uniformemente as possibilidades e desperdiça muito esforço



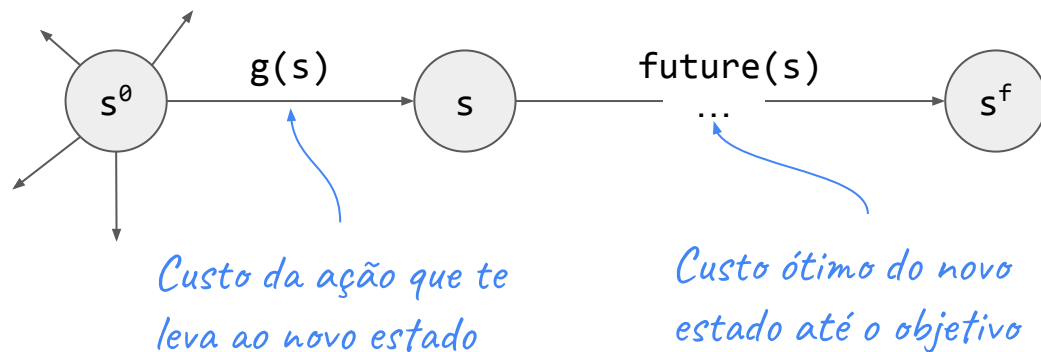
Função de Custo

- O que queremos é encontrar uma função que nos informe as ações mais promissoras
 - $f(n) = ?$
- O objetivo é direcionar a busca em direção ao alvo



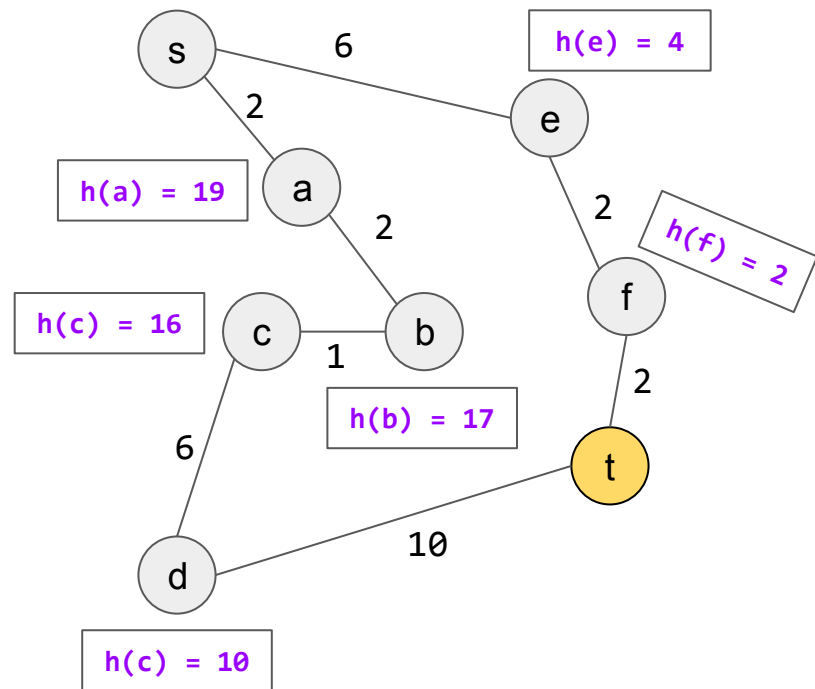
Função de Custo

- Função ideal: o quão próximo da meta um determinado nó está?
 - $f(s) = g(s) + \text{future}(s)$
 - Se ela existisse o problema estaria resolvido.



Função de Custo

- Função ideal: o quão próximo da meta um determinado nó está?
 - $f(s) = g(s) + \text{future}(s)$
 - Se ela existisse o problema estaria resolvido.
- Considere o problema ao lado
 - Como o UCS funcionaria nos custos $g(s)$?
 - E se considerarmos a previsão de futuro perfeita $h(s)$?



Função de Custo

- Função ideal: o quão próximo da meta um determinado nó está?
 - $f(s) = g(s) + \text{future}(s)$
 - Se ela existisse o problema estaria resolvido.
- Não temos bola de cristal, então o que fazer?

Função de Custo

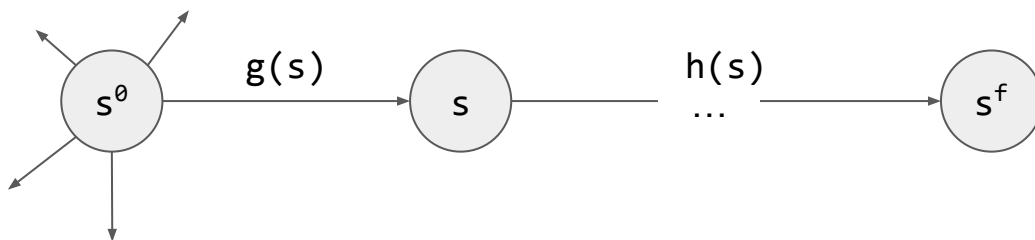
- Função ideal: o quão próximo da meta um determinado nó está?
 - $f(s) = g(s) + \text{future}(s)$
 - Se ela existisse o problema estaria resolvido.
- Não temos bola de cristal, então o que fazer?



Heurística

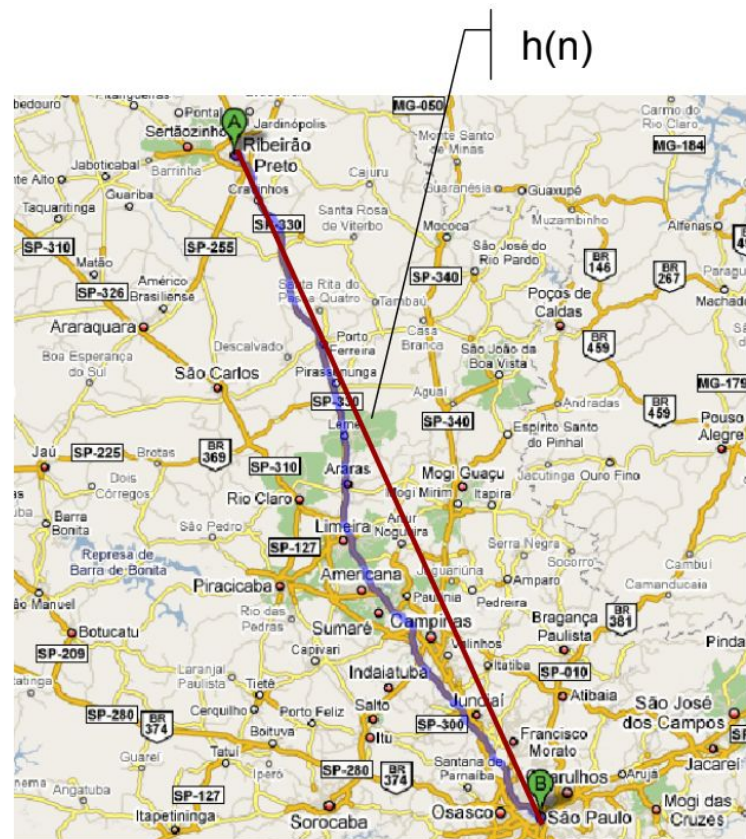
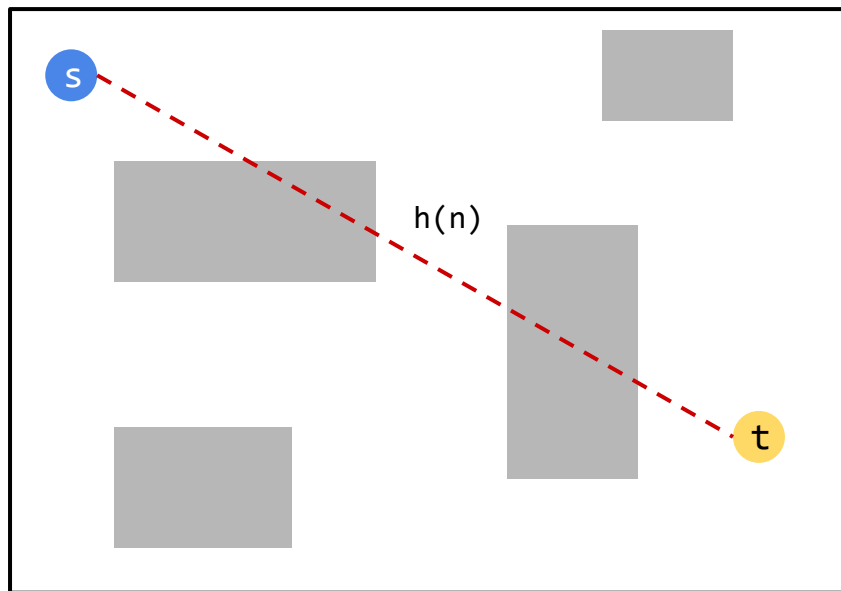
- Função ideal: o quão próximo da meta um determinado nó está?
 - $f(s) = g(s) + \text{future}(s)$
 - Se ela existisse o problema estaria resolvido.
- Solução: Relaxe as restrições do seu problema. O futuro não precisa ser estimado com perfeição!
 - $f(s) = g(s) + h(s)$

Heurística: aproximação



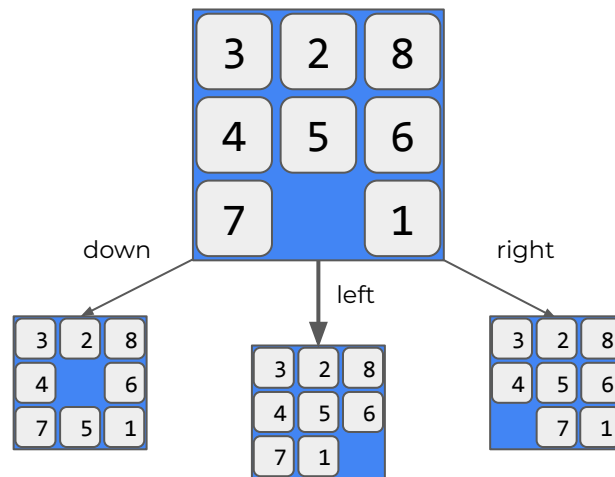
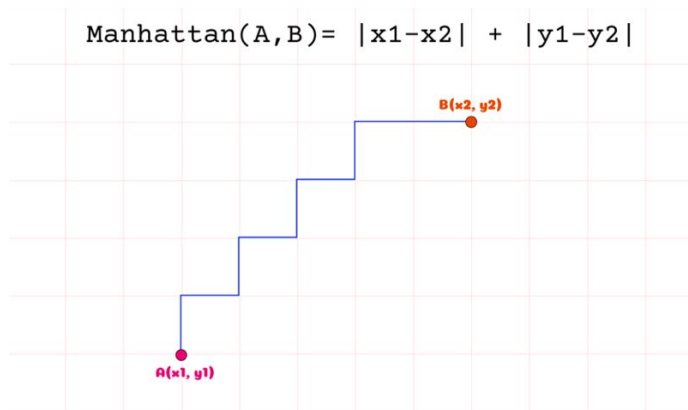
Heurística

- Ex: distância euclidiana



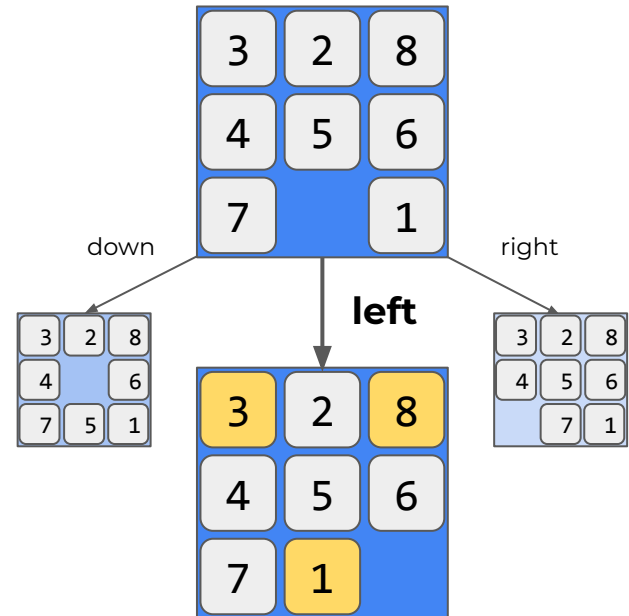
Heurística

- No problema quebra-cabeça-8, podemos desconsiderar as limitações de movimento e usar a **distância de manhattan** das peças até suas posições finais como função heurística
 - $f(n) = g(n) + h(n)$
 - $f(n) = 1 + \text{manhattan}(n)$



Heurística

- No problema quebra-cabeça-8, podemos desconsiderar as limitações de movimento e usar a **distância de manhattan** das peças até suas posições finais como função heurística
 - $f(n) = g(n) + h(n)$
 - $f(n) = 1 + \text{manhattan}(n)$
- Qual o valor da função heurística para **left**?
 - Apenas 1, 3 e 8 estarão fora de lugar

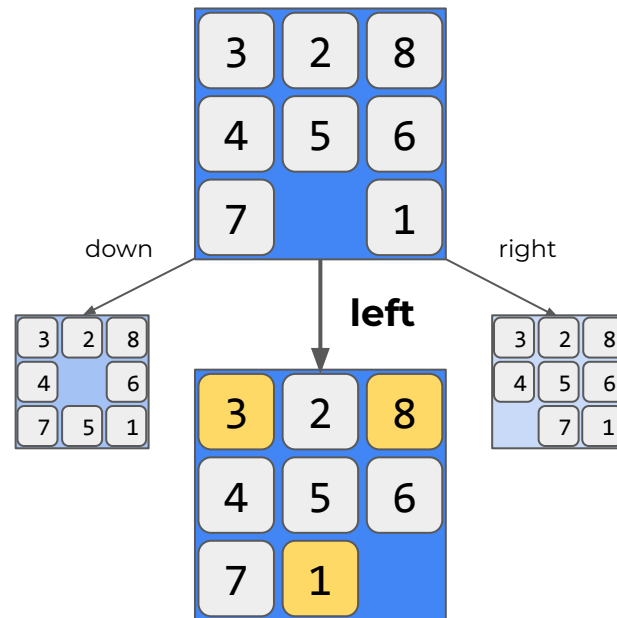
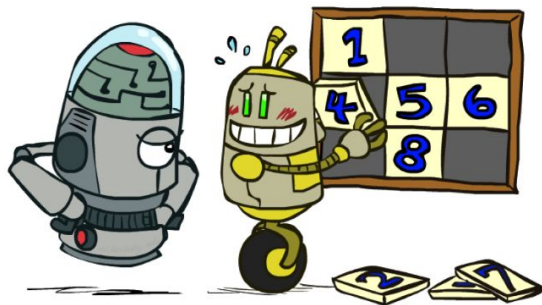
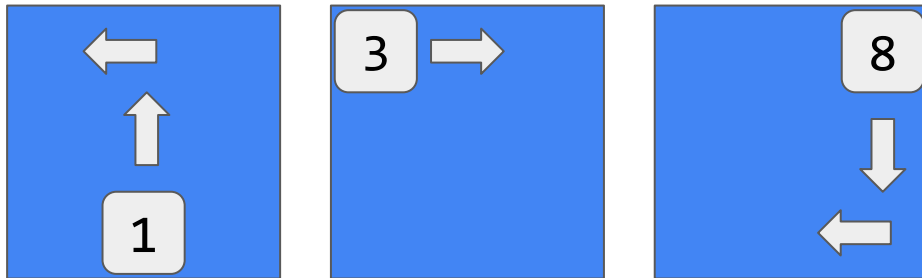


Heurística

- O valor da função heurística para **left** é:

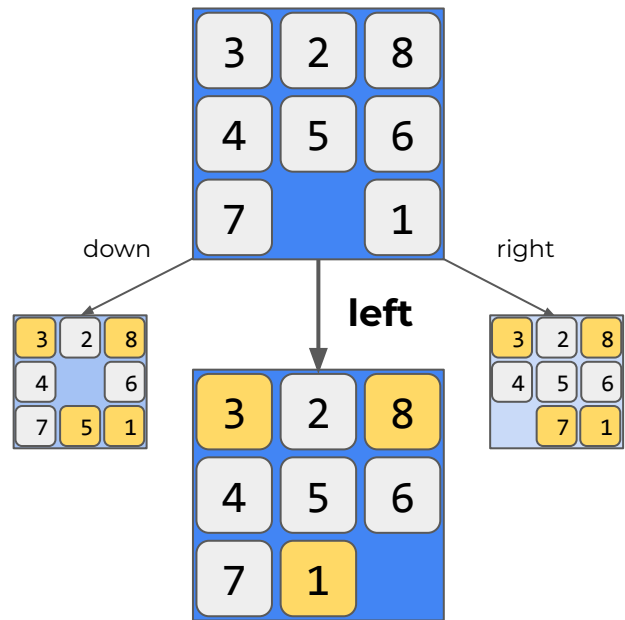
- $h(\text{left}) = 3 + 2 + 3 = 8$

A heurística **estima** que estarei a 8 movimentos da solução



Heurística

- Avaliando todas as alternativas de estado, temos
 - $h(\text{left}) = 3 + 2 + 3 = 8$
 - $h(\text{down}) = 4 + 2 + 1 + 3 = 10$
 - $h(\text{right}) = 4 + 2 + 1 + 3 = 10$
- Nesse contexto, **left** é a melhor escolha.

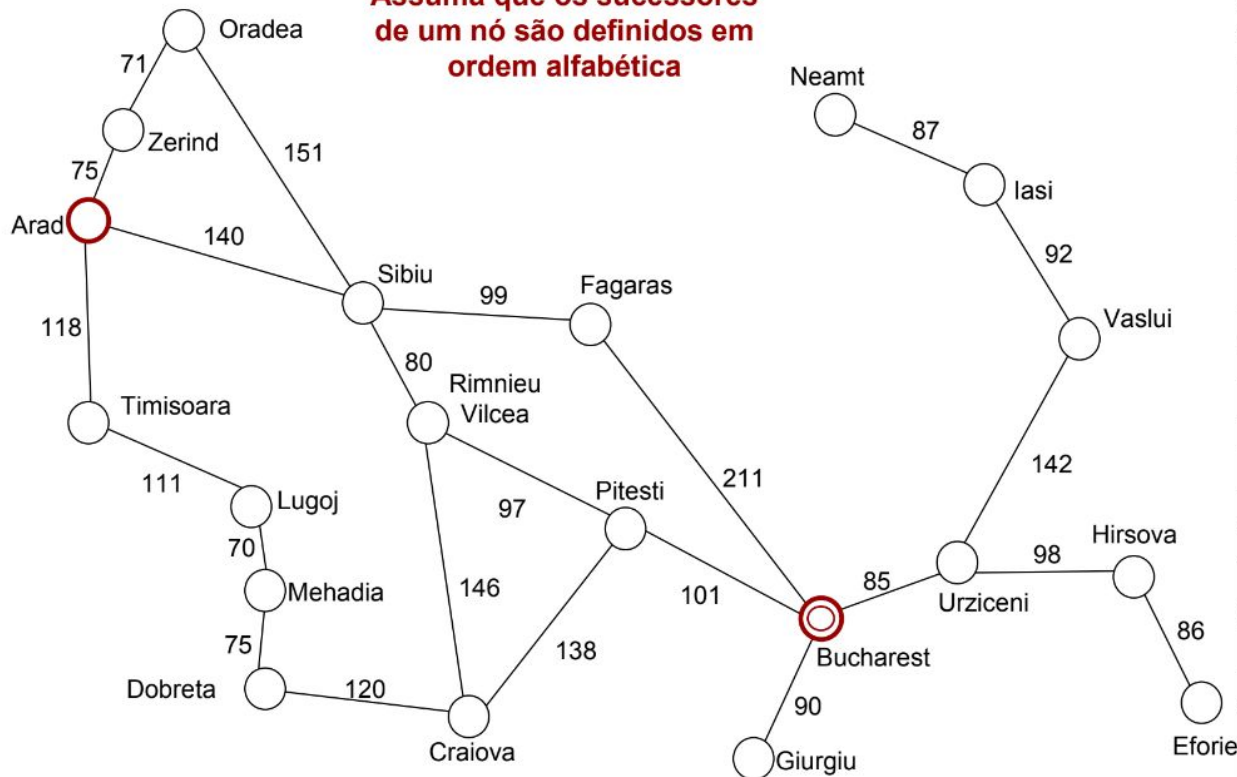


Busca Gulosa

- Busca gulosa pela melhor escolha ([best-first greedy search](#)), expande os nós de menor custo
- Avalia os nós apenas de acordo com a função heurística
 - $f(n) = h(n)$
- Se assemelha à busca em profundidade (DFS), pois segue um único caminho até que:
 - Encontre o objetivo
 - Caia em um nó sem saída (adotando o backtrack)
- Mesmos defeitos do DFS: não é ótima nem completa se houver caminhos infinitos

Busca Gulosa

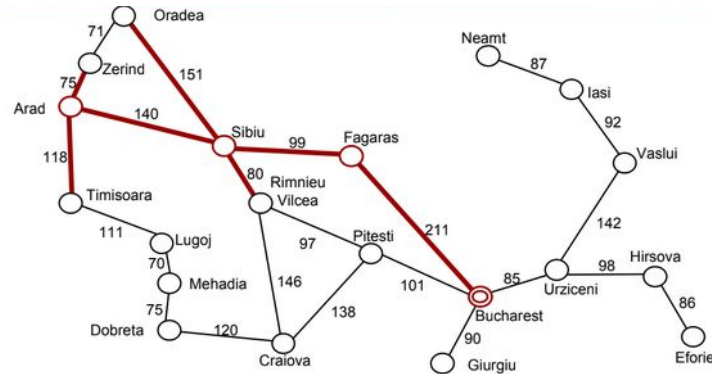
Assuma que os sucessores de um nó são definidos em ordem alfabética



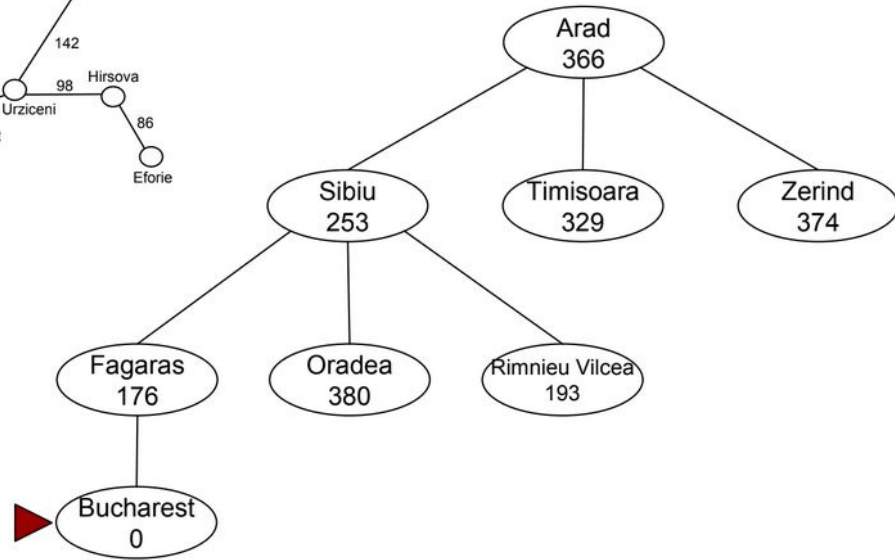
Distância em linha reta até Bucharest

h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieniu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Busca Gulosa



h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieniu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Custo da solução encontrada A-S-F-B = 450

Custo da solução ótima A-S-RV-P-B = 418

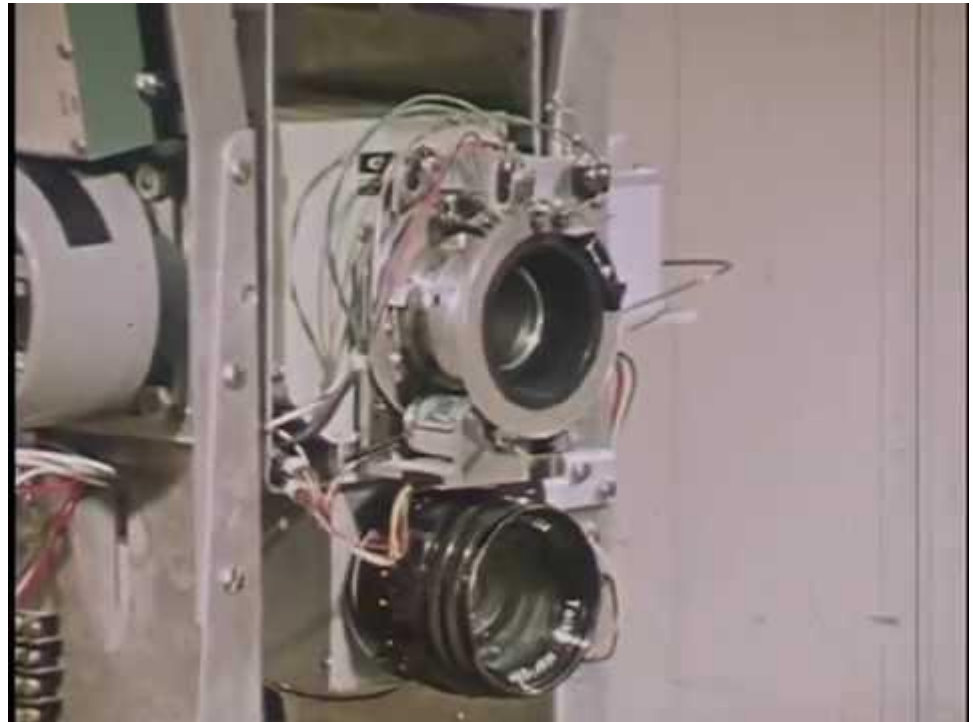
Busca Gulosa

- Não é ótima, nem completa;
- Mas é muito rápida se a sua heurística for de boa qualidade!

Busca A*

- Shakey The Robot (1972)
- Primeiro robô de propósito geral
- Autores propuseram um algoritmo baseado em heurísticas para o planejamento de caminho do Shakey
 - Ótimo e completo
 - O mais rápido!*

*Para uma dada heurística, nenhum outro algoritmo irá expandir menos nós.

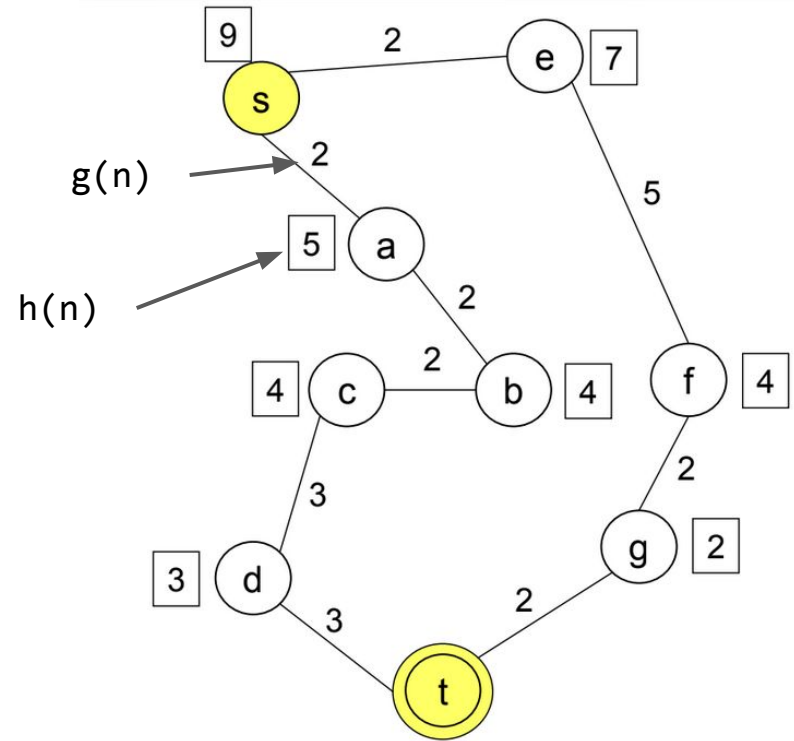


Busca A*

- Lembra quando dissemos sobre o Uniform Cost Search que:
 - Se a heurística for uma função constante, é um caso particular do A*
- Essencialmente, A* é o UCS onde a fila de prioridade é ordenada de acordo com:
 - $f(n) = g(n) + h(n)$
- Ele é considerado completo e ótimo quando:
 - Heurística do nó objetivo é $h(t) = 0$
 - $h(n)$ é admissível (otimista): seu custo é menor ou igual que o custo real até o objetivo
 - $0 \leq h(n) \leq \text{future}(n)$

Busca A*

- Assumindo a heurística dada ao lado
 - Ela é admissível?



Busca A*

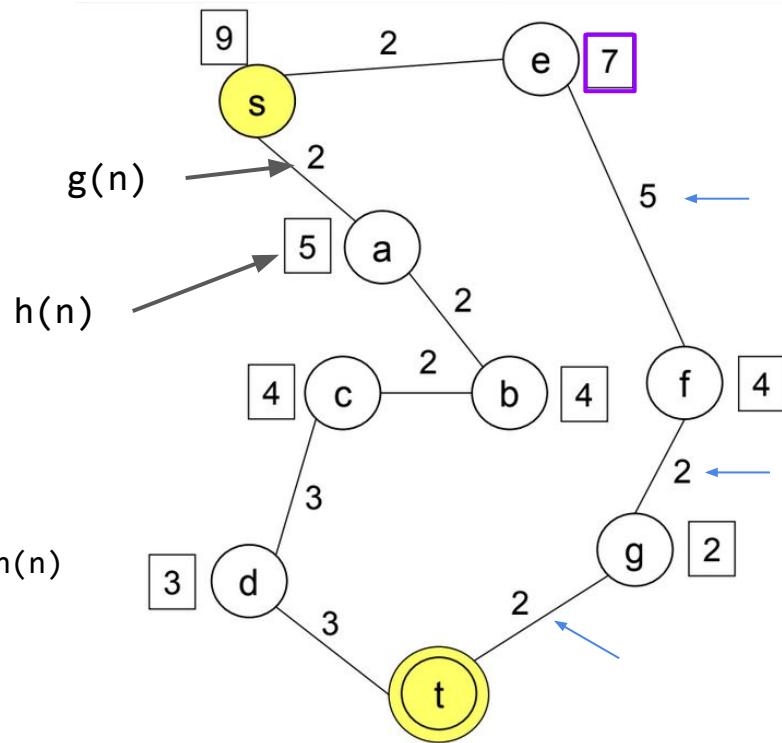
- Assumindo a heurística dada ao lado
 - Ela é admissível?

$$h(n) \leq \text{future}(n)$$

$$h(e) = 7$$

$$\text{future}(e) = 5 + 2 + 2$$

- Essa relação precisa ser verdadeira para qualquer $h(n)$



Busca A*

$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca

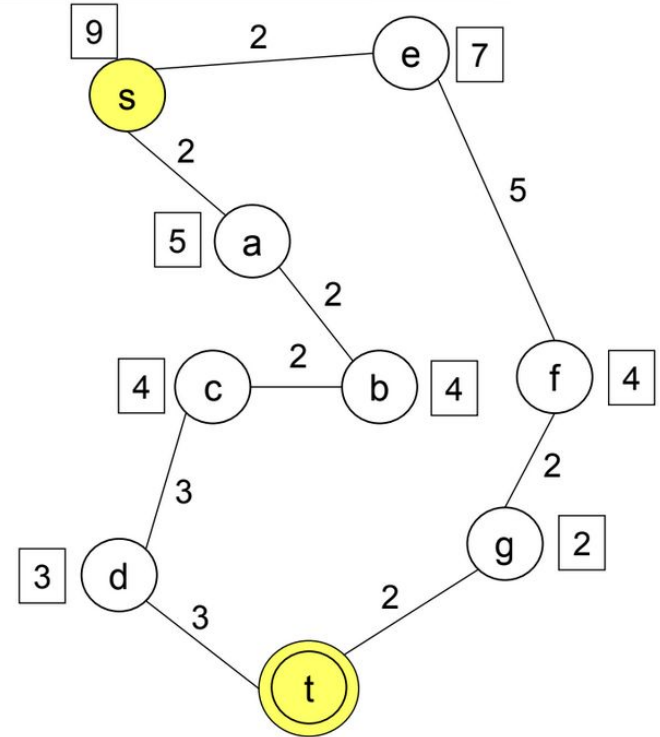


função Busca-de-Custo-Uniforme (*problema*)

retorna uma solução ou falha

Busca-Genérica (*problema*, Insere-Ordem-Crescente)

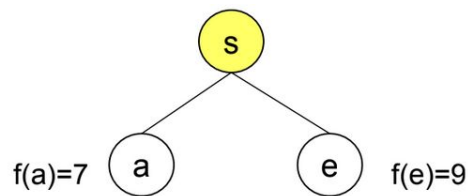
Fila de prioridade



Busca A*

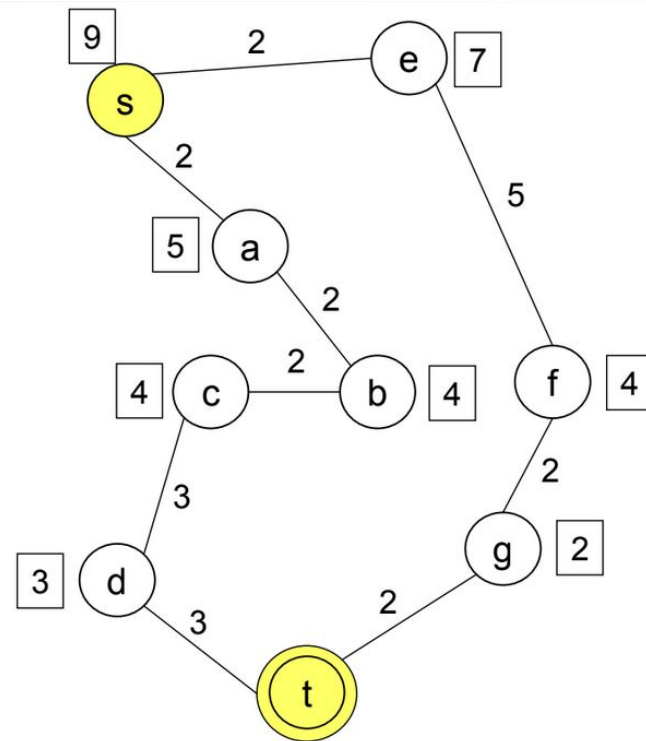
$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



$$f(s, a) = 2 + 5$$

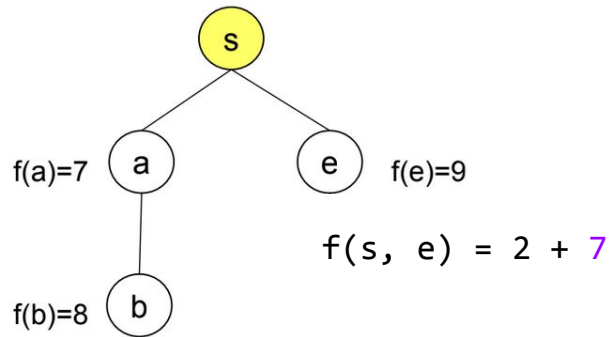
$$f(s, e) = 2 + 7$$



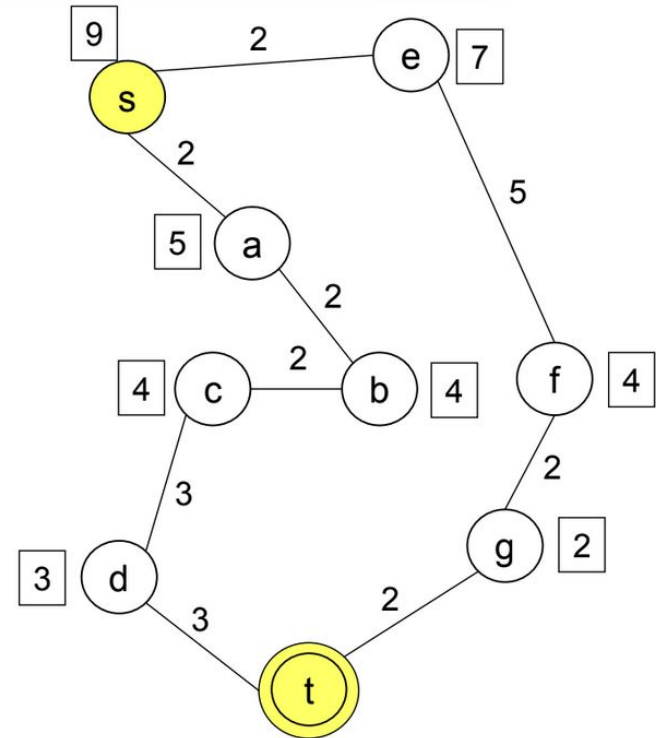
Busca A*

$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



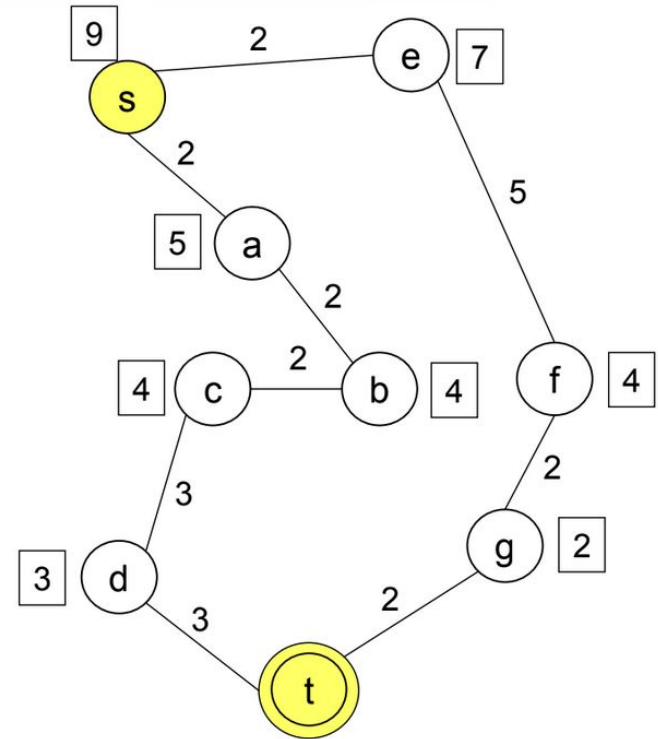
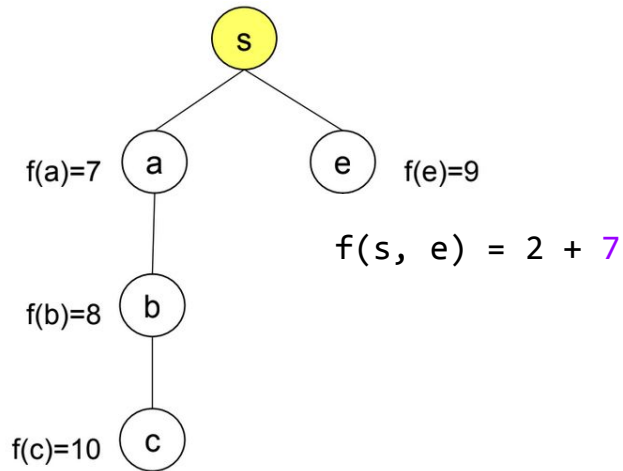
$$f(sab) = 2+2 + 4$$



Busca A*

$$f(n) = g(n) + h(n)$$

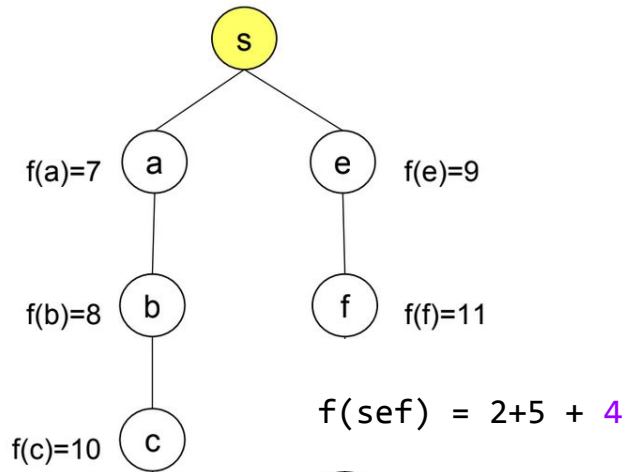
- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



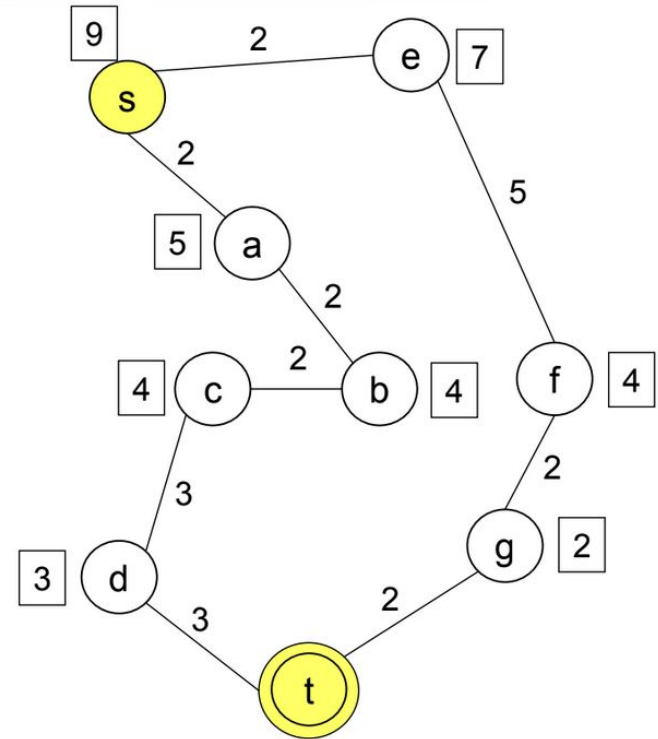
Busca A*

$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



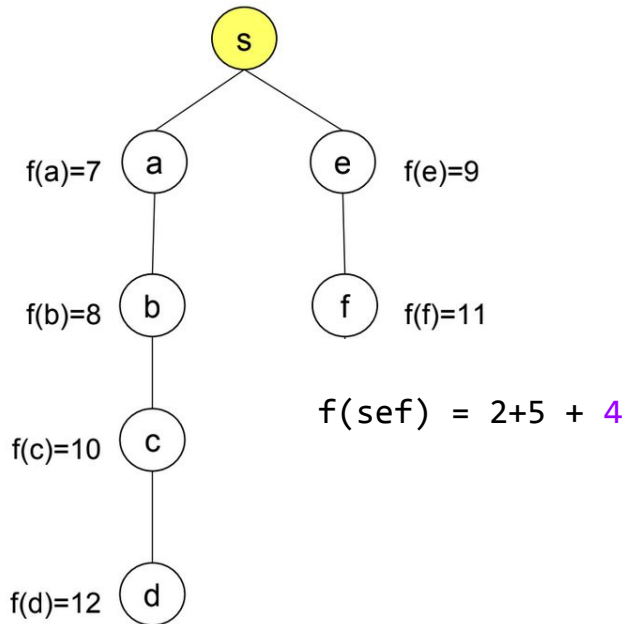
$$f(sabc) = 2+2+2 + 4$$



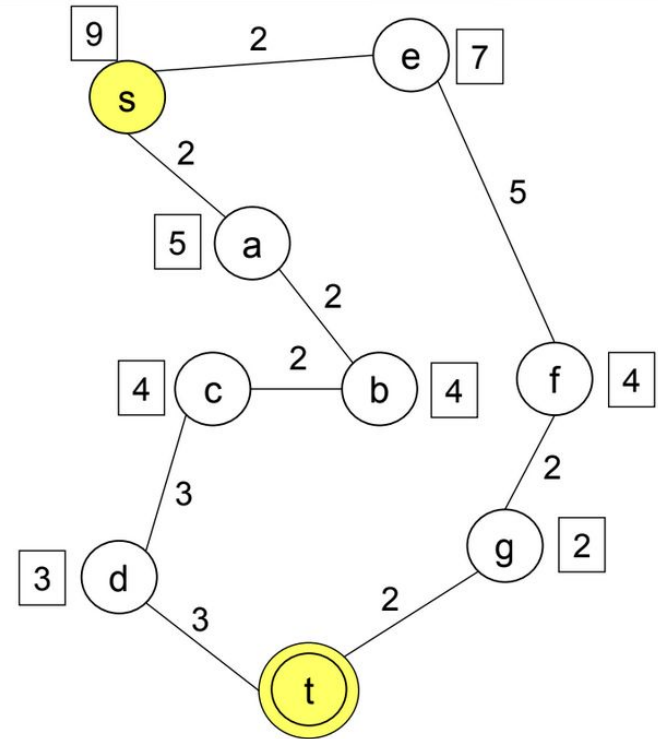
Busca A*

$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



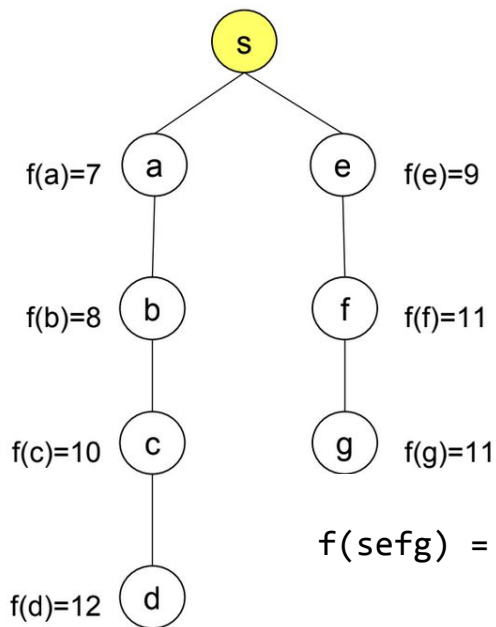
$$f(sabcd) = 2+2+2+3 + 3$$



Busca A*

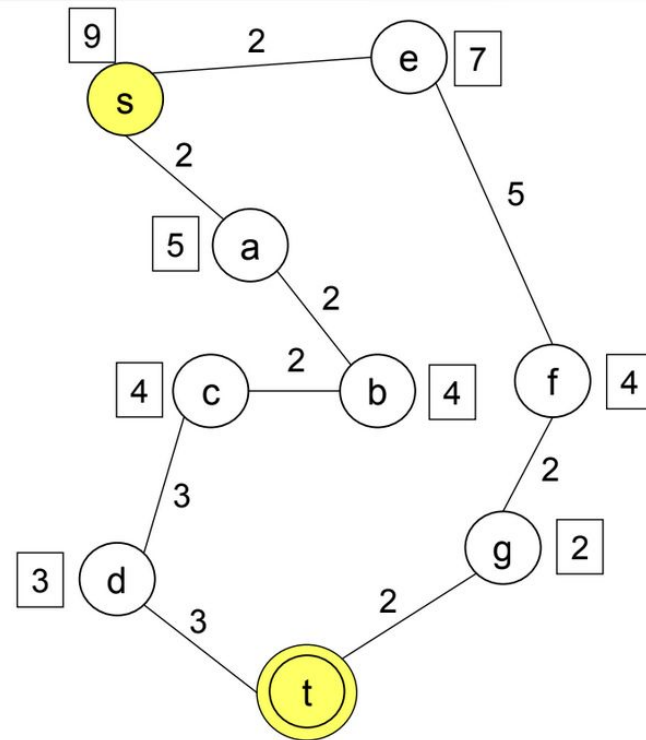
$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca



$$f(\text{sefg}) = 2+5+2 + 2$$

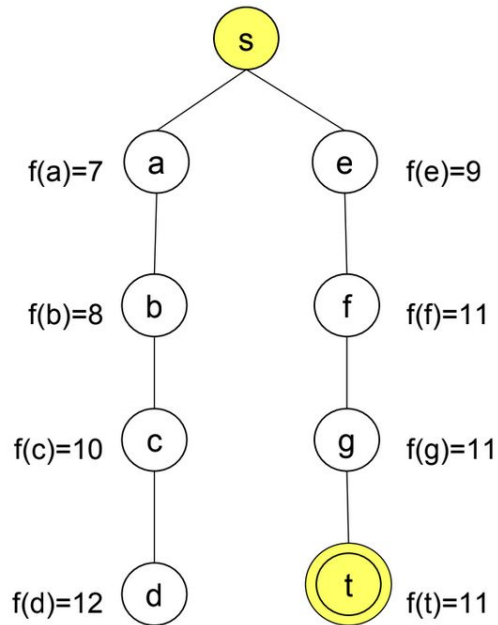
$$f(\text{sab cd}) = 2+2+2+3 + 3$$



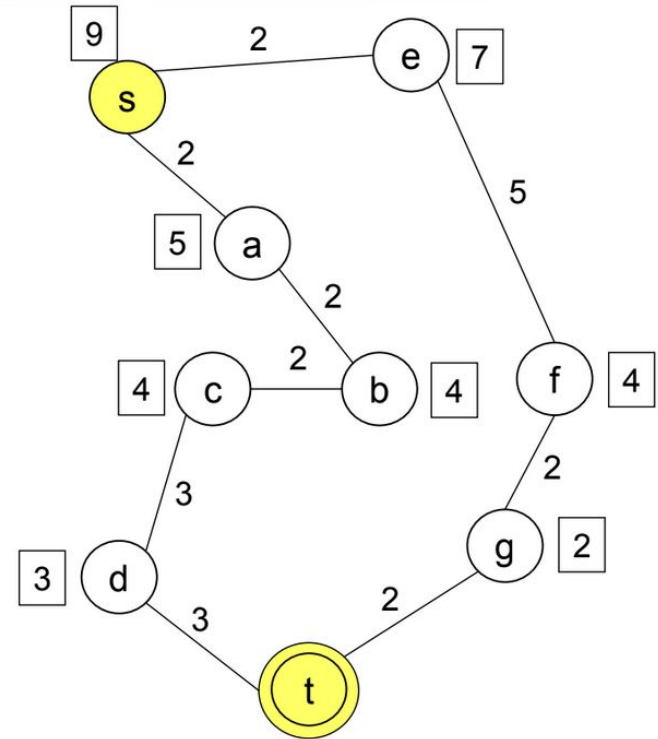
Busca A*

$$f(n) = g(n) + h(n)$$

- Diferente do custo $g(n)$, o valor da heurística $h(n)$ não é incrementado ao longo da busca

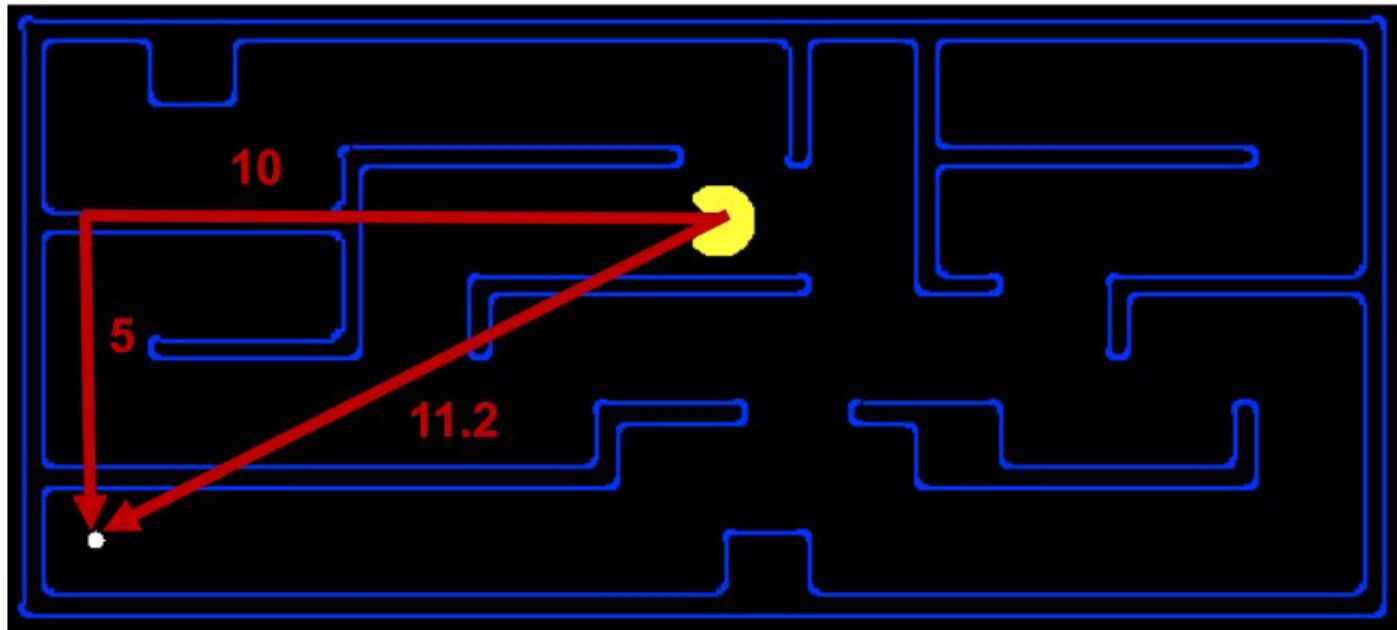


$$f(sabcd) = 2+2+2+3 + 3 \quad f(sefgt) = 2+5+2+2 + 2$$



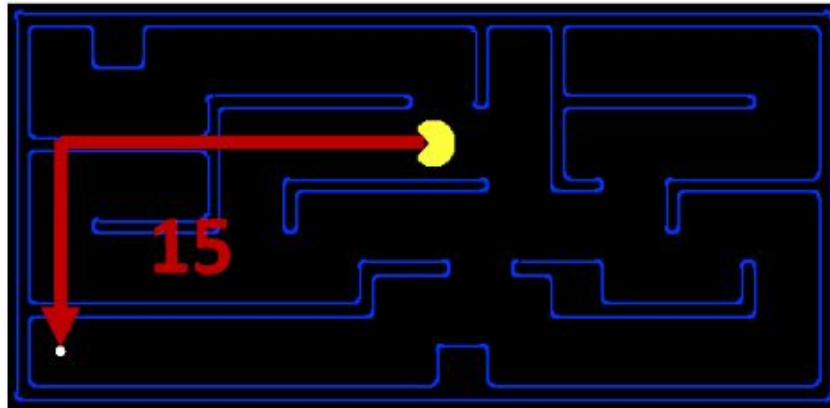
Escolhendo Heurísticas

- No exemplo do Pacman, qual a melhor heurística? Euclideana ou Manhattan?



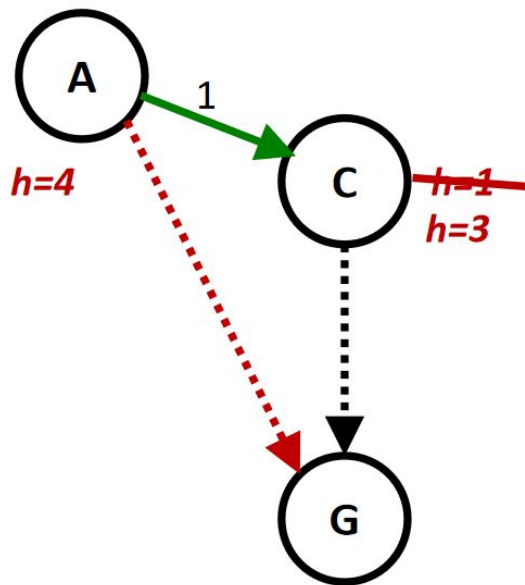
Escolhendo Heurísticas

- No exemplo do Pacman, qual a melhor heurística? Euclideana ou Manhattan?
- Heurísticas admissíveis que aproximam melhor o problema real são melhores!
- Encontrar heurísticas boas, baratas de calcular e admissíveis é a chave do sucesso!



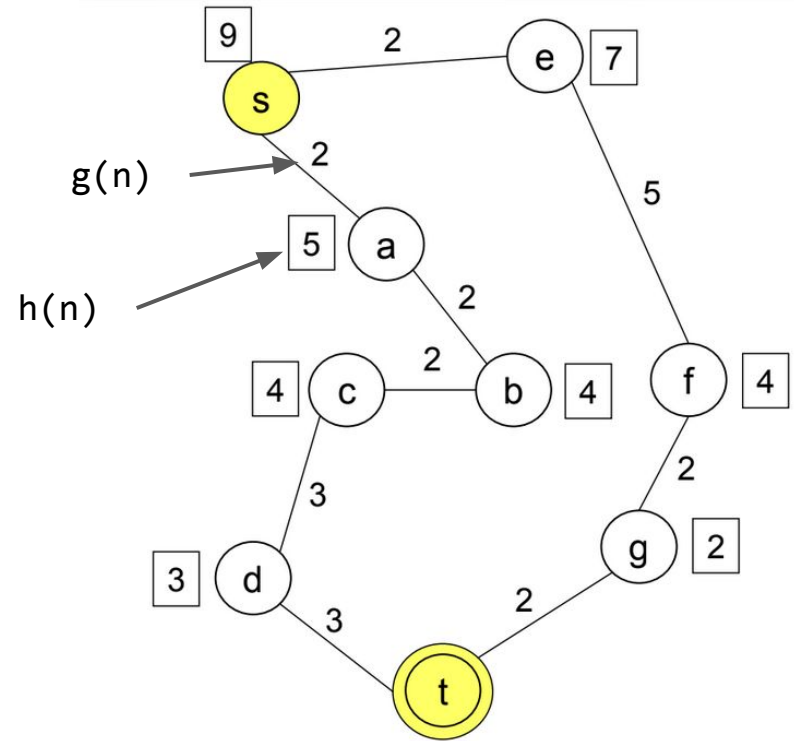
Admissibilidade vs Consistência

- Admissibilidade: custo da heurística \leq custo real até a meta
 - $0 \leq h(n) \leq \text{future}(n)$
- Consistência: custo da heurística em “arcos” \leq custo do arco
 - $h(A) - h(C) \leq g(A,C)$
 - ou $h(A) \leq g(A,C) + h(C)$ (triangle inequality)
- Lembrando: $h(\text{objetivo}) = 0$
- Consistência implica admissibilidade



Admissibilidade e Consistência

- Assumindo a heurística dada ao lado
 - Ela é consistente?



Admissibilidade e Consistência

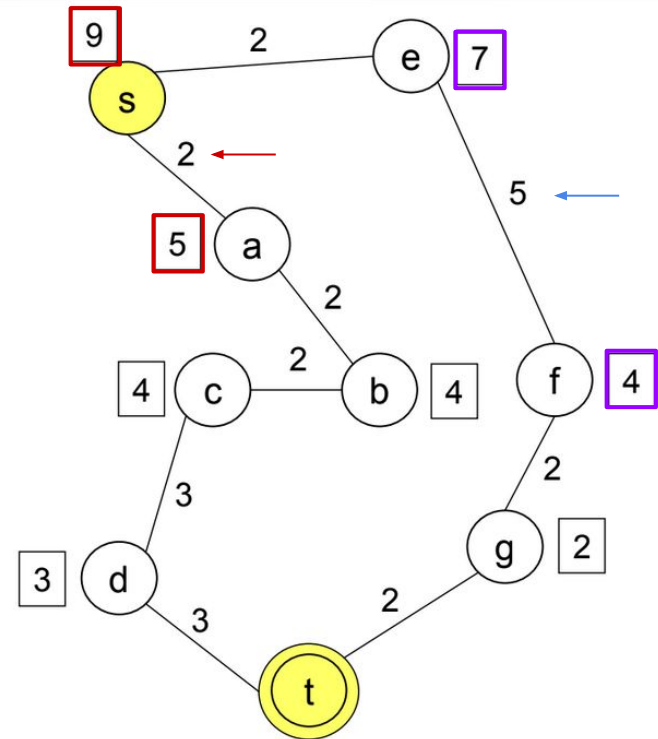
- Assumindo a heurística dada ao lado
 - Ela é consistente?

$$h(e) - h(f) \leq g(e, f)$$

$$7 - 4 \leq 5$$

$$h(s) - h(a) \leq g(s, a)$$

$$9 - 5 \leq 2$$



Admissibilidade e Consistência

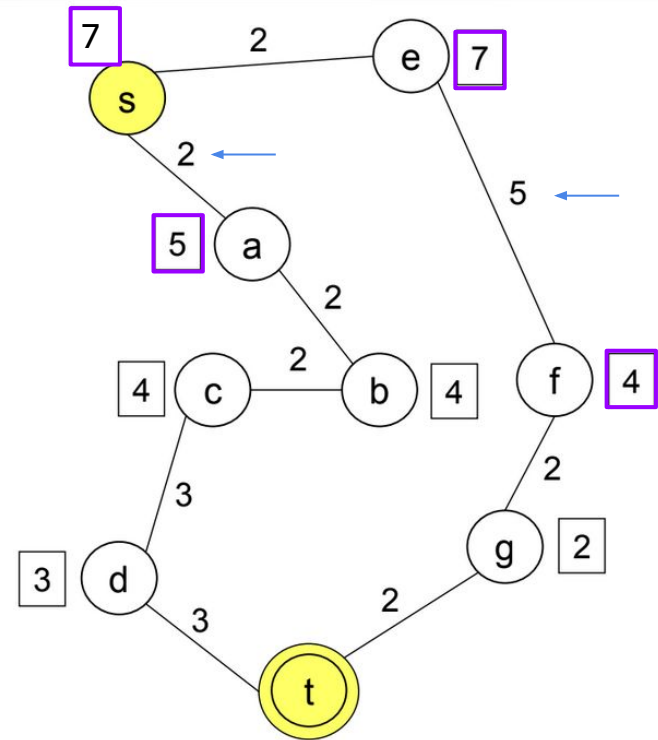
- Assumindo a heurística dada ao lado
 - Ela é consistente?

$$h(e) - h(f) \leq g(e, f)$$

$$7 - 4 \leq 5$$

$$h(s) - h(a) \leq g(s, a)$$

$$7 - 5 \leq 2$$



Admissibilidade e Consistência

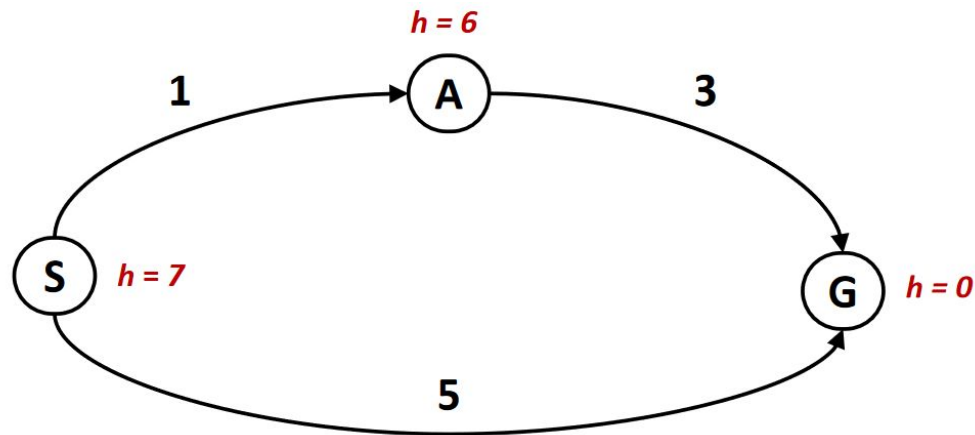
- Heurísticas não admissíveis, ou inconsistentes, podem desviar seu algoritmo da solução ótima
- Admissível

✗ $h(A) \leq \text{future}(A)$

- Consistente

✗ $h(A) - h(G) \leq g(A, G)$

- Solução encontrada: S-G
- Solução ótima: S-A-G

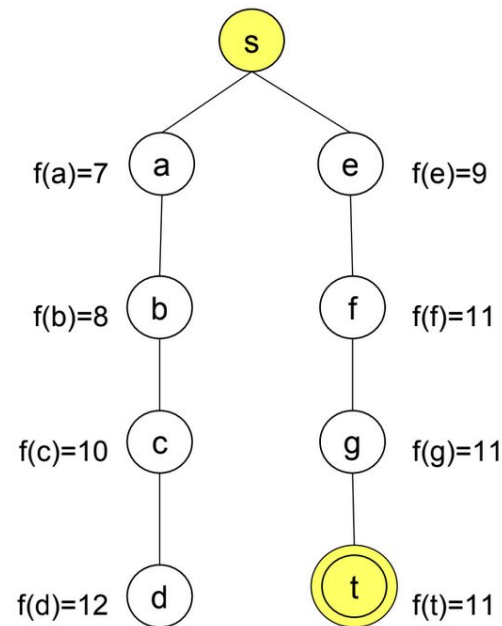


Admissibilidade e Consistência

- Se a heurística é consistente (e portanto admissível), o **valor-f** cresce durante a busca

$$f(n) = g(n) + h(n)$$

- Ou seja, o custo de um mesmo caminho nunca diminui
 - O exemplo ao lado explora dois caminhos distintos



Admissibilidade e Consistência

- Se a heurística é consistente (e portanto admissível), o **valor-f** cresce durante a busca

$$f(n) = g(n) + h(n)$$

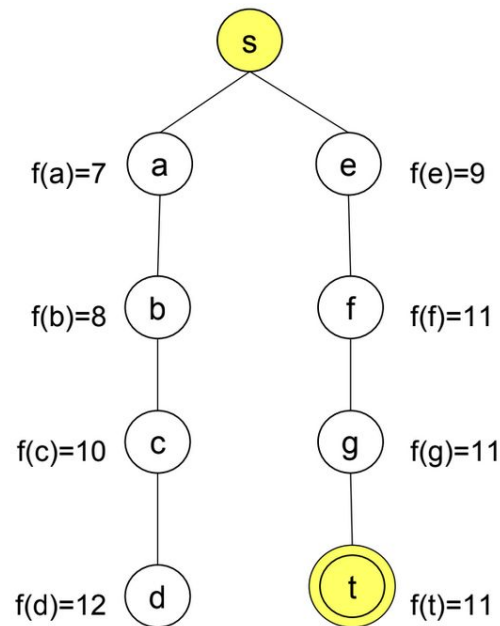
- Ou seja, o custo de um mesmo caminho nunca diminui
 - O exemplo ao lado explora dois caminhos distintos

$$f(b) = g(s, \dots, b) + h(b)$$

$$f(c) = g(s, \dots, b) + g(b, c) + h(c)$$

$$h(b) - h(c) \leq g(b, c) \text{ \#consistência}$$

$$h(b) \leq g(b, c) + h(c)$$



Admissibilidade e Consistência

- Se a heurística é consistente (e portanto admissível), o **valor-f** cresce durante a busca

$$f(n) = g(n) + h(n)$$

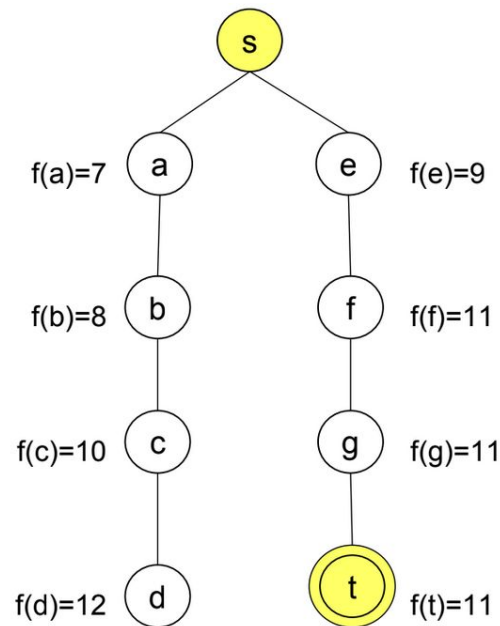
- Ou seja, o custo de um mesmo caminho nunca diminui
 - O exemplo ao lado explora dois caminhos distintos

$$f(b) = g(s, \dots, b) + h(b)$$

$$f(c) = g(s, \dots, b) + g(b, c) + h(c)$$

$$h(b) - h(c) \leq g(b, c) \text{ \#consistência}$$

$$h(b) \leq g(b, c) + h(c)$$



Implementação

- Busca genérica
- Fila de prioridade
 - heap
- No caso do A* o valor-f define a prioridade
- No caso do UCS, é o custo $g(n)$

```
170 class PriorityQueue:
171     """
172     Implements a priority queue data structure. Each inserted item
173     has a priority associated with it and the client is usually interested
174     in quick retrieval of the lowest-priority item in the queue. This
175     data structure allows O(1) access to the lowest-priority item.
176     """
177     def __init__(self):
178         self.heap = []
179         self.count = 0
180
181     def push(self, item, priority):
182         entry = (priority, self.count, item)
183         heapq.heappush(self.heap, entry)
184         self.count += 1
185
186     def pop(self):
187         (_, _, item) = heapq.heappop(self.heap)
188         return item
189
190     def isEmpty(self):
191         return len(self.heap) == 0
192
193     def update(self, item, priority):
194         # If item already in priority queue with higher priority, update its priority and rebuild the heap.
195         # If item already in priority queue with equal or lower priority, do nothing.
196         # If item not in priority queue, do the same thing as self.push.
197         for index, (p, c, i) in enumerate(self.heap):
198             if i == item:
199                 if p <= priority:
200                     break
201                 del self.heap[index]
202                 self.heap.append((priority, c, item))
203                 heapq.heapify(self.heap)
204                 break
205         else:
206             self.push(item, priority)
207
```

Implementação

- Busca genérica
 - Registro de custo $g(n)$ acumulado no caminho
 - `visited = dict(key=node, value=total_cost)`
 - Verificação se foi visitado ou se está sendo revisitado por um caminho melhor
 - Fila de prioridade com base no valor-f

```
26     for next_node, cost, new_state in problem.getValidMoves(curr_node.state):
27         if next_node not in visited or curr_node.total_cost+cost < visited[next_node]:
28
29             node = Node(name=next_node, state=new_state, parent=curr_node)
30             visited[next_node] = curr_node.total_cost+cost
31             valor_f = visited[next_node] + heuristic(next_node, goal)
32             priority_queue.push(next_node, priority=valor_f)
```

Complexidade do A*

- Apesar do A* usar a heurística para reduzir o espaço de busca, seu custo ainda é exponencial na profundidade.
- Uma heurística ruim fará o algoritmo explorar muitos caminhos
- Uma boa heurística levará o algoritmo direto no caminho da solução
- Vale repetir: se a heurística é desprezível, o algoritmo se assemelha ao UCS.

Algoritmo	Tempo	Espaço
DFS-Backtracking	$O(b^D)$	$O(D)$
DFS	$O(b^D)$	$O(D)$
DFS-I	$O(b^d)$	$O(D)$
BFS	$O(b^d)$	$O(b^d)$
UCS	$O(b^{C^*/e})$	$O(b^{C^*/e})$
A*	$O(b'^d)$	$O(b'^d)$

b' = ramificação efetiva da busca

quantos caminhos são explorados em paralelo

Complexidade do A*

- O mais crítico em seu caso é o custo em espaço, também exponencial.
- A memória estoura antes de você ficar entediado esperando a execução
- Temos alternativas:
 - IDA* (Iterative Deepening A*)
 - RBFS (Recursive Best-First Search)

Algoritmo	Tempo	Espaço
DFS-Backtracking	$O(b^D)$	$O(D)$
DFS	$O(b^D)$	$O(D)$
DFS-I	$O(b^d)$	$O(D)$
BFS	$O(b^d)$	$O(b^d)$
UCS	$O(b^{C^*/e})$	$O(b^{C^*/e})$
A*	$O(b'^d)$	$O(b'^d)$

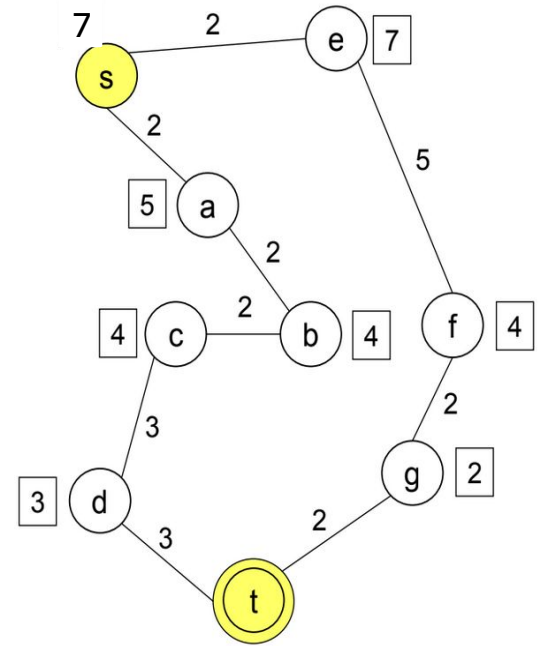
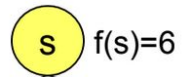
b' = ramificação efetiva da busca

quantos caminhos são explorados em paralelo

IDA*

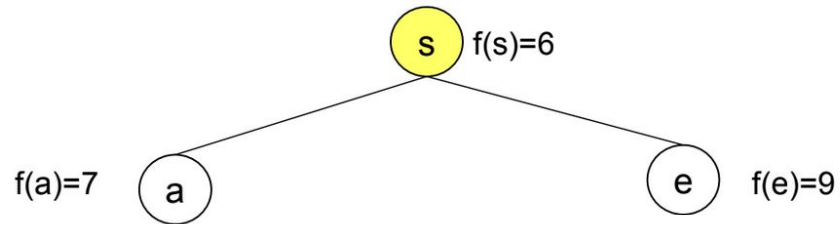
- Iterative Deepening A*
- Similar à DFS-I (profundidade iterativa). Nesse caso, não se limita pela profundidade, mas sim pelo valor-f.
- Limite inicial é o valor-f do nó inicial
 - $f(n) = g(n) + h(n)$

IDA*

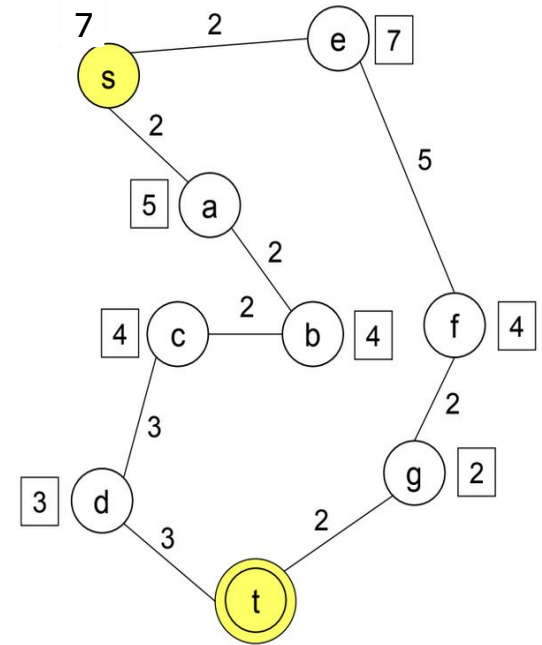


Limite =

IDA*



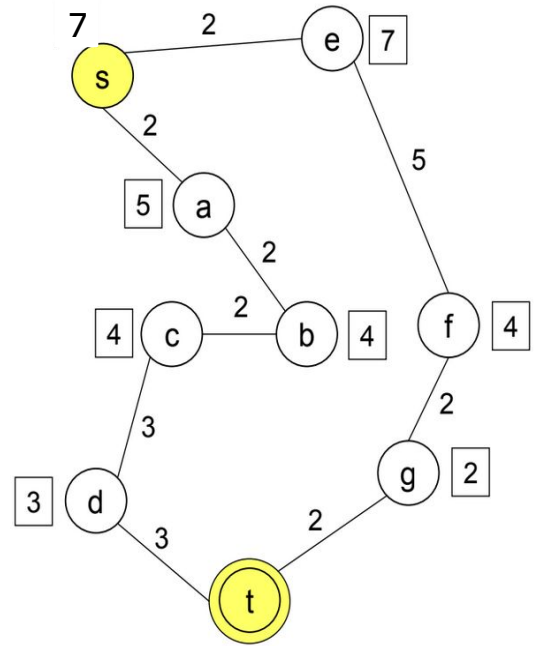
Limite = 6



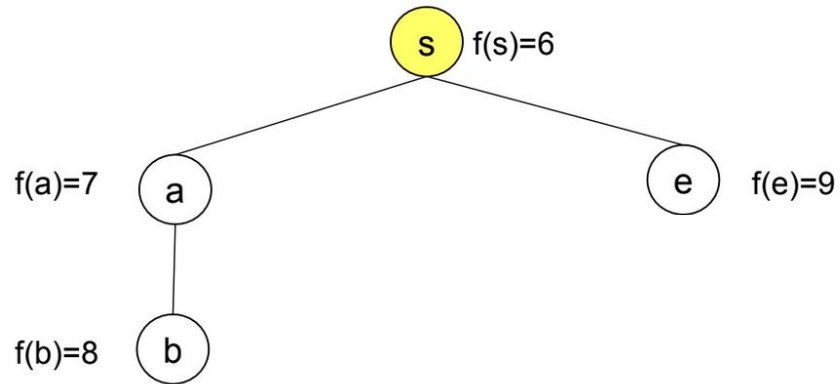
IDA*



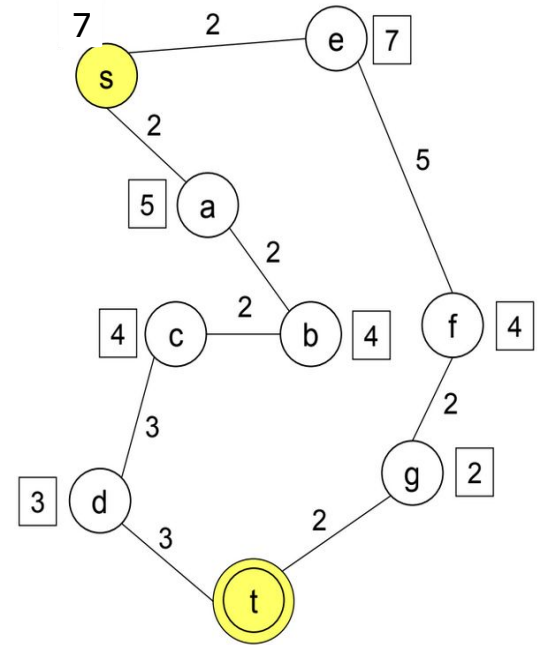
Limite = 7



IDA*



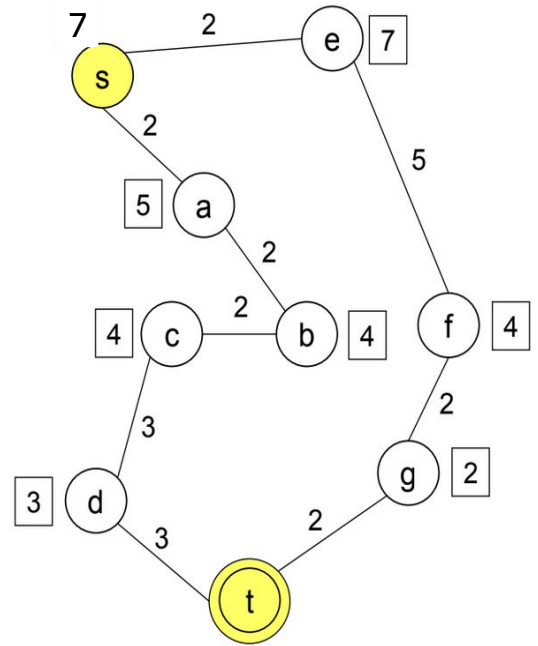
Limite = 7



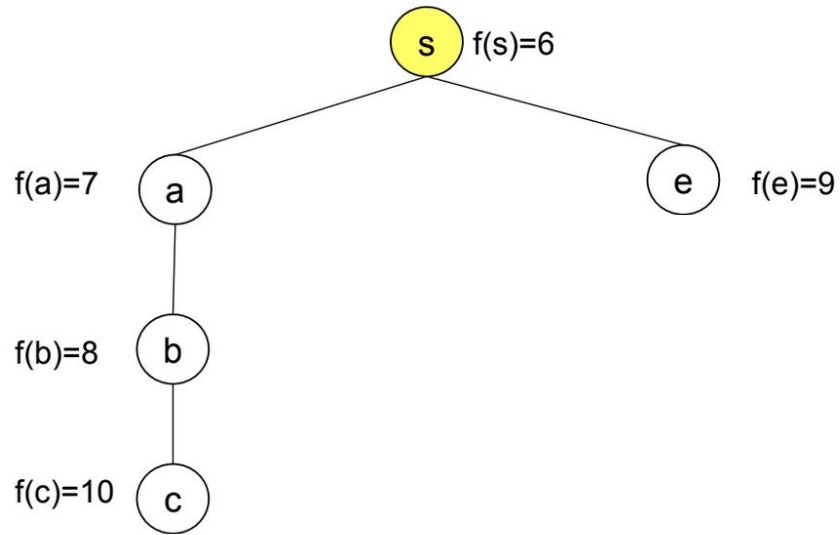
IDA*



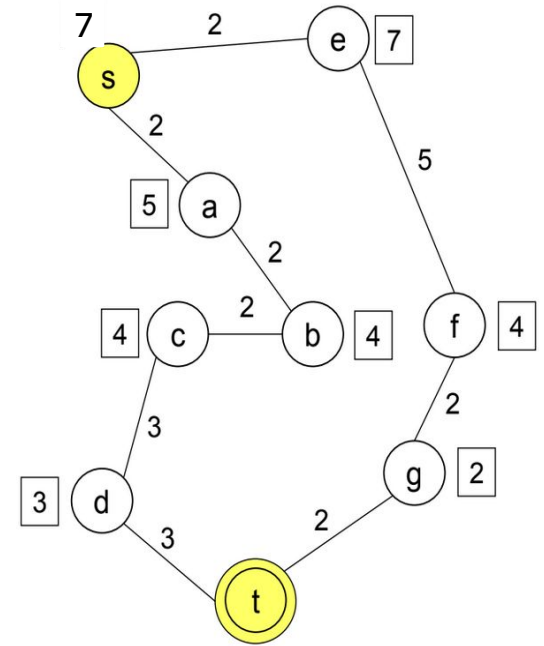
Limite = 8



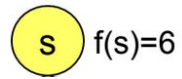
IDA*



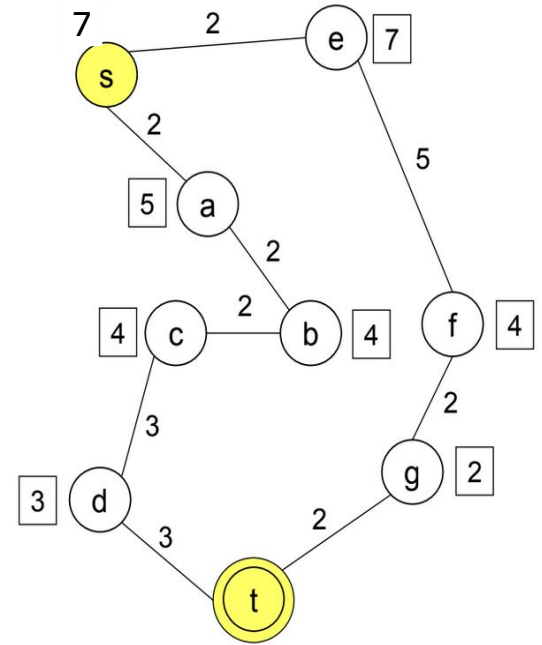
Limite = 8



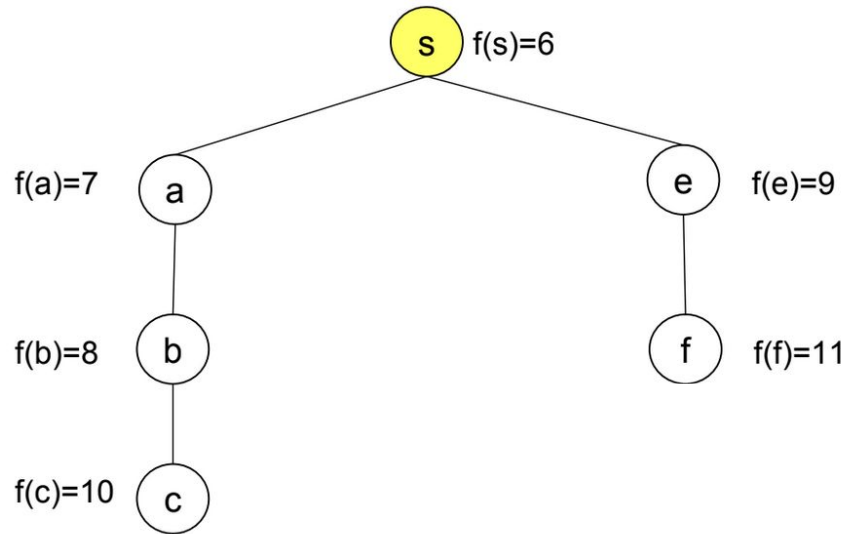
IDA*



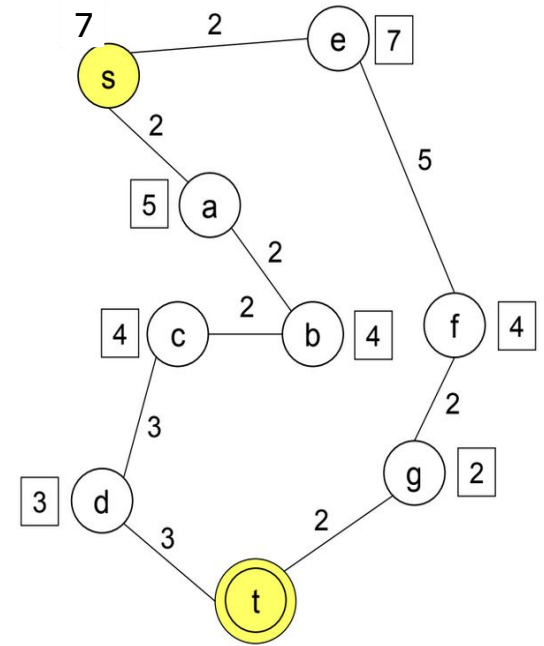
Limite = 9



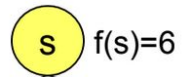
IDA*



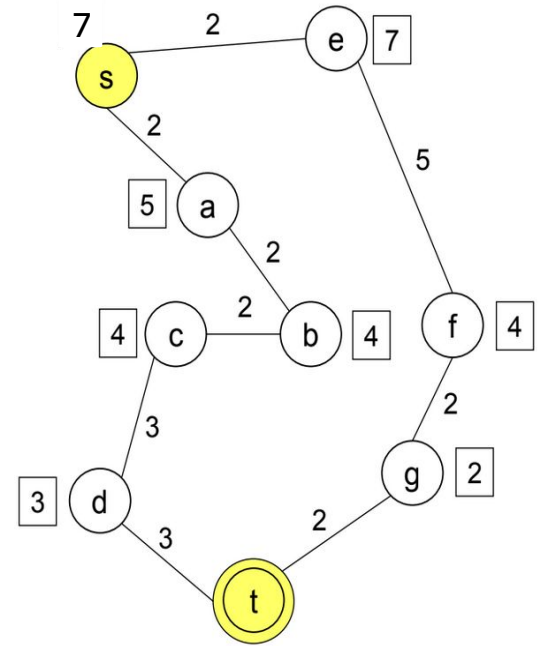
Limite = 9



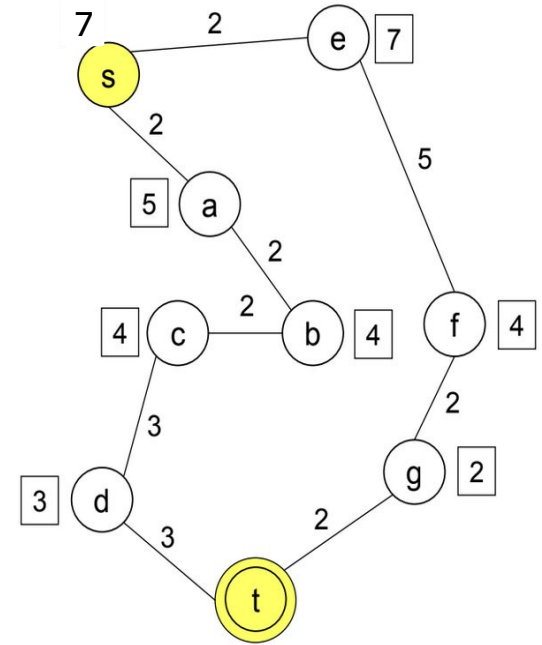
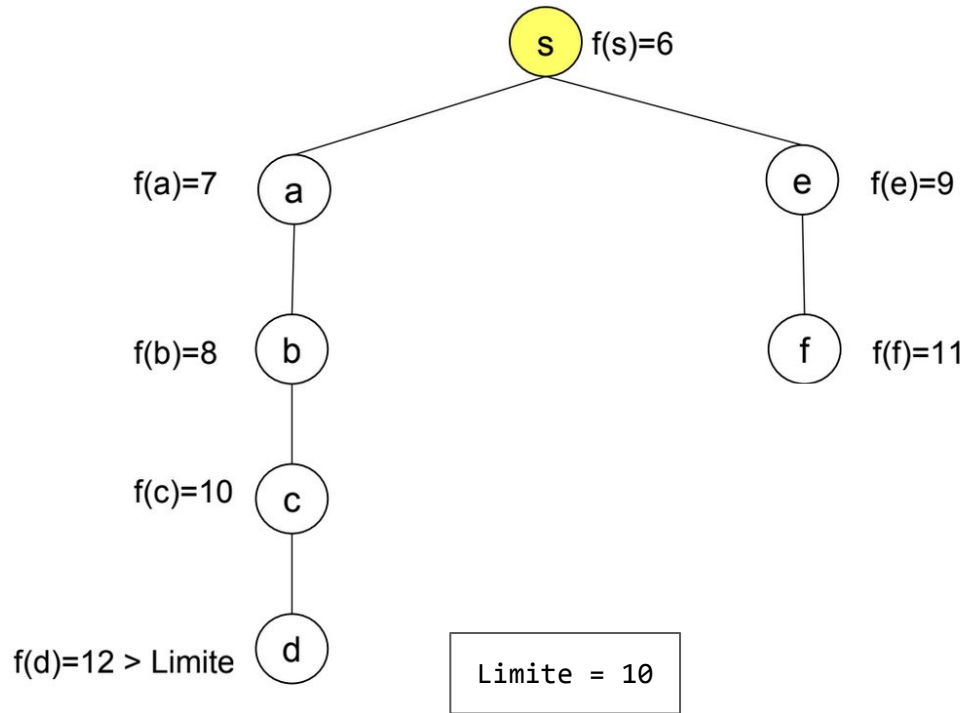
IDA*



Limite = 10



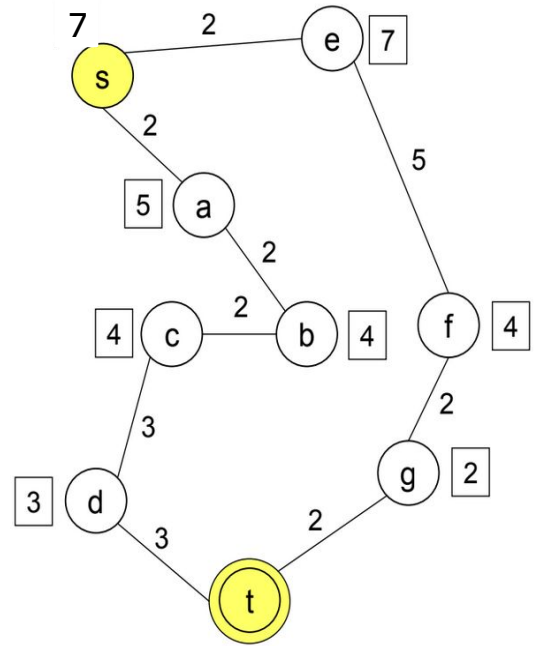
IDA*



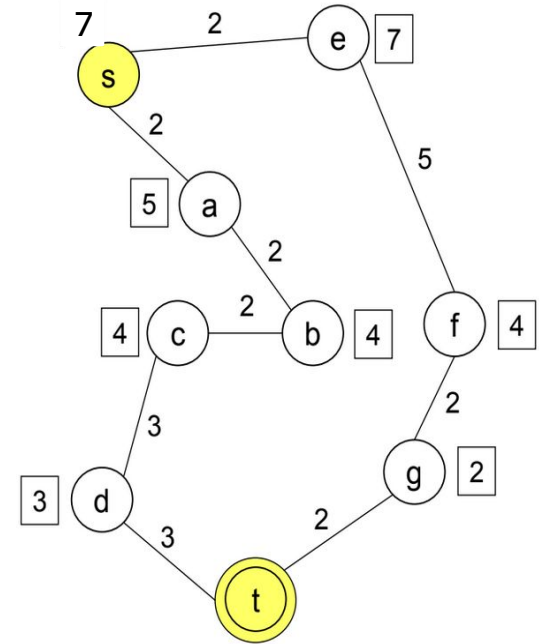
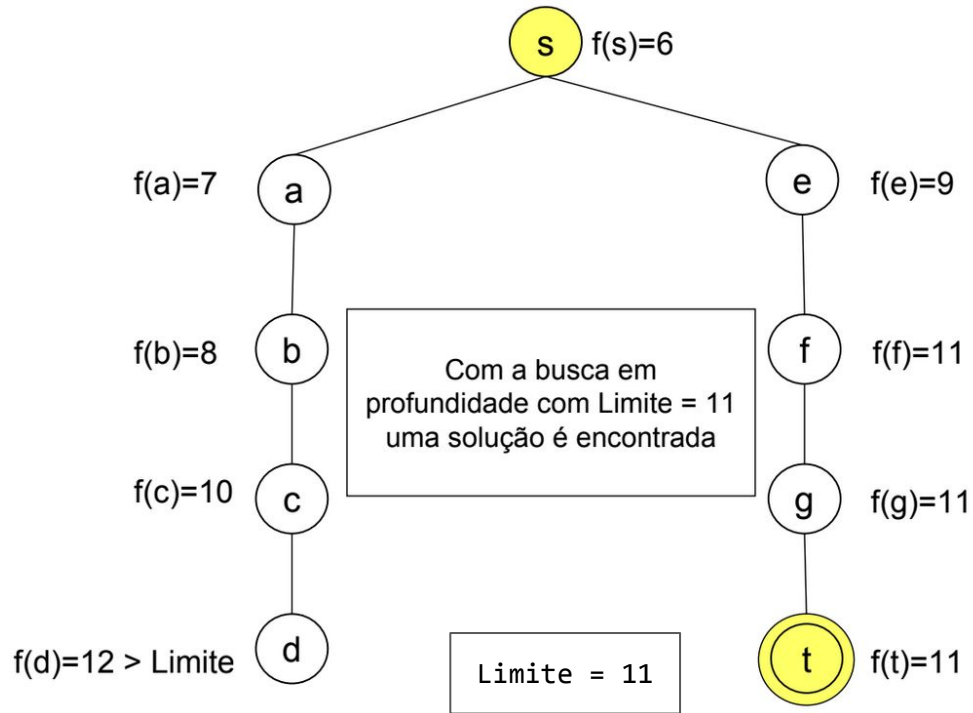
IDA*



Limite = 11



IDA*



IDA*


- IDA* gasta muito espaço re-expandindo nós
 - Se os valores-f variam muito entre sub-árvores, ele gasta mais tempo re-expandindo que descobrindo novos nós
- RBFS resolve esse problema

RBFS

- Recursive Best-First Search
- O algoritmo “esquece” as sub-árvores que não estão sendo exploradas no momento, registrando somente o valor-f do caminho
- Torna-se linear na complexidade do espaço
- Também explora com base em **limites de valor-f** que serão incrementados ao longo da exploração
- O limite é determinado pelos valores-f dos filhos ao longo do caminho atual
- Exceder esse limite é o critério para esquecer o caminho, atualizar o limite e refazer a exploração

RBFS

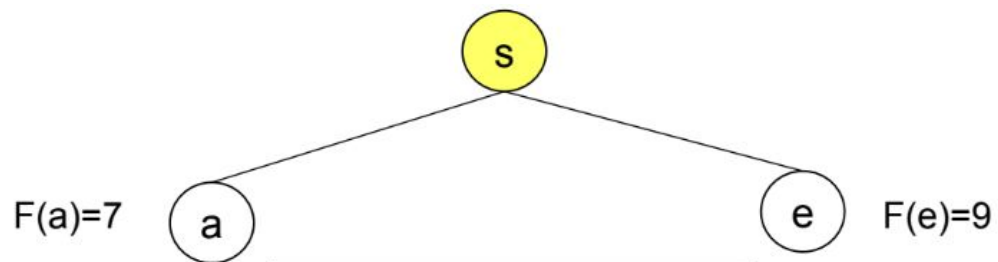
- É importante distinguir entre valores-f reais e valores-f de registros temporários. Usaremos
 - $f(n)$: valor-f estimado para o nó n
 - $F(n)$: valor-f armazenado do caminho em espera passando por n
- $F(N) = f(n)$ se n nunca foi expandido
- $F(N) = \min\{F(n_i)\}$, sendo n_i um nó sucessor de n

 *O melhor caminho passando por n (até agora)*

Note que:

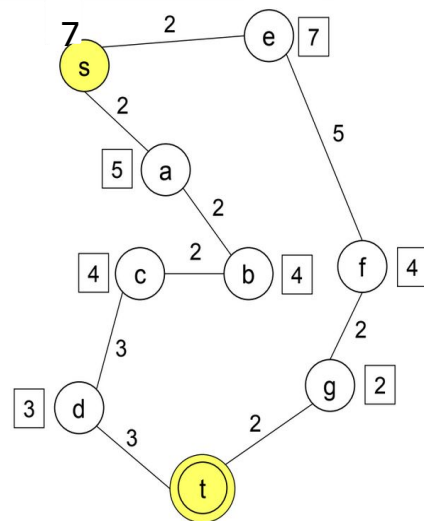
- Se $F(n) > f(n)$ então sabemos que n foi expandido anteriormente e $F(n)$ foi determinado a partir dos filhos de n , mas os filhos foram removidos da memória

RBFS

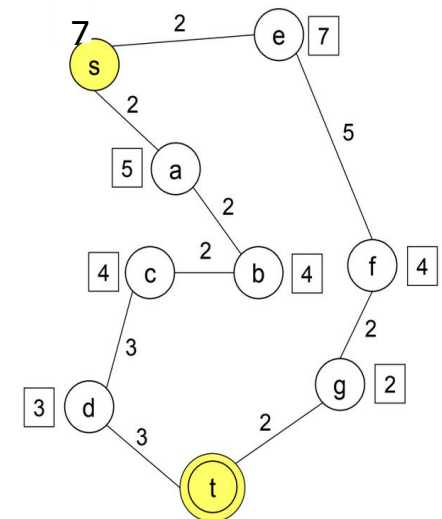
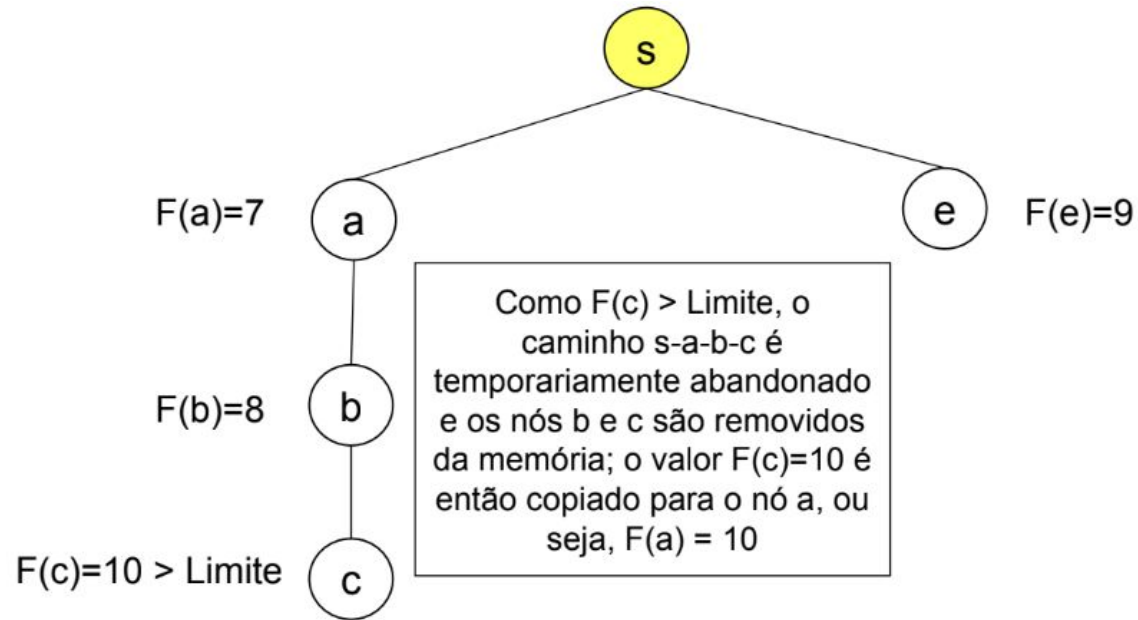


O melhor candidato é o nó **a**,
pois $F(a) < F(e)$. A busca
prossegue via **a**

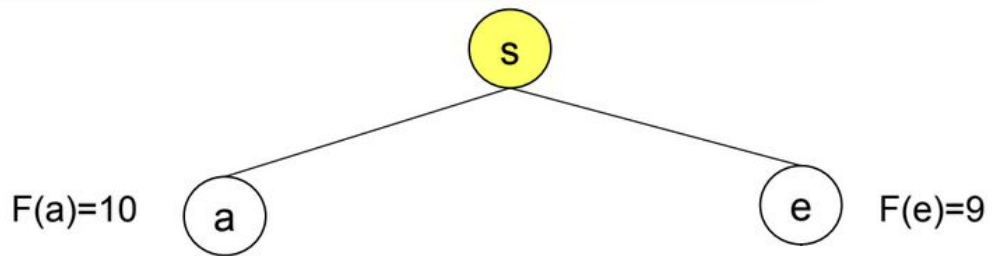
Limite = 9



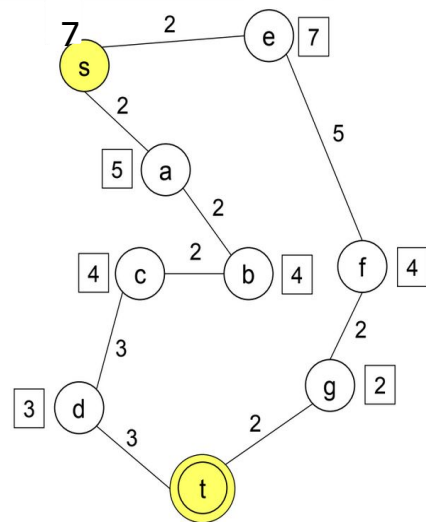
RBFS



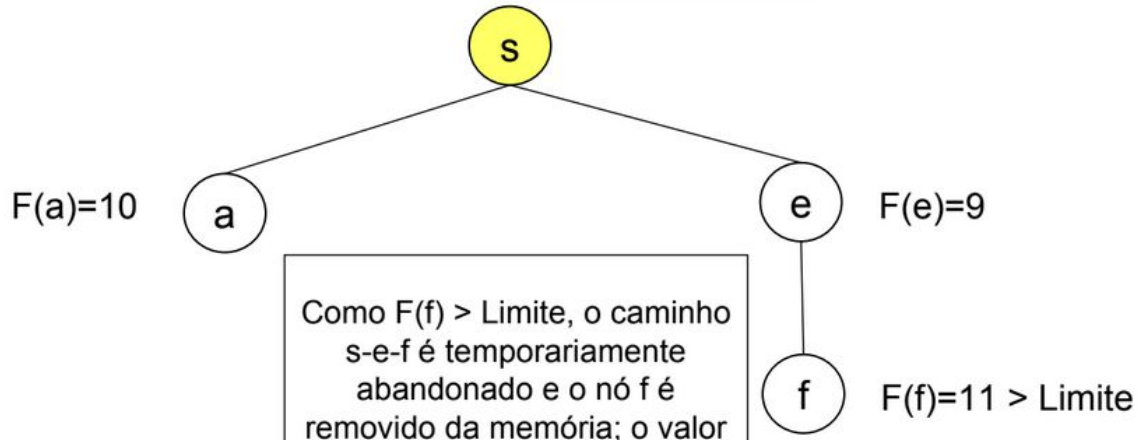
RBFS



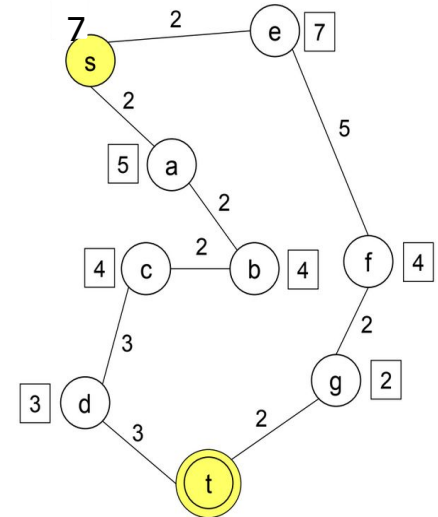
Limite = 10



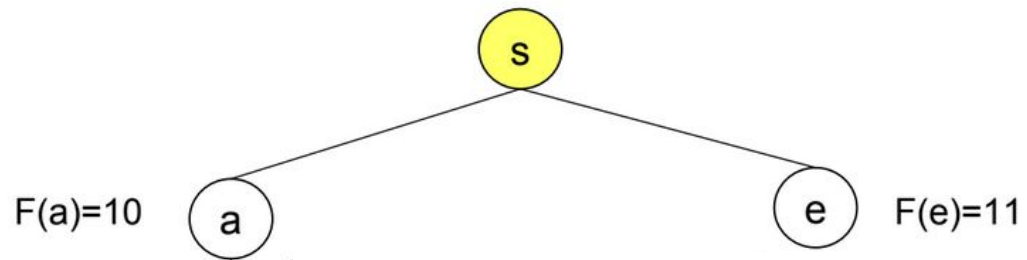
RBFS



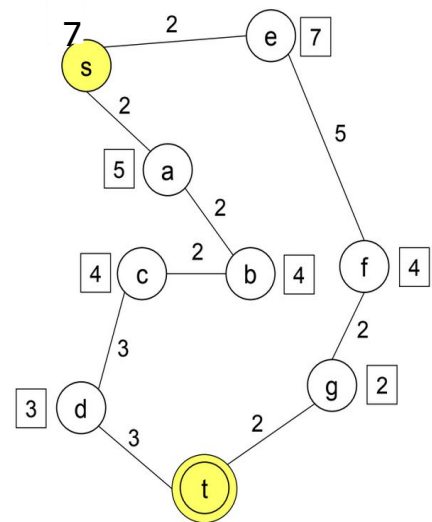
Limite = 10



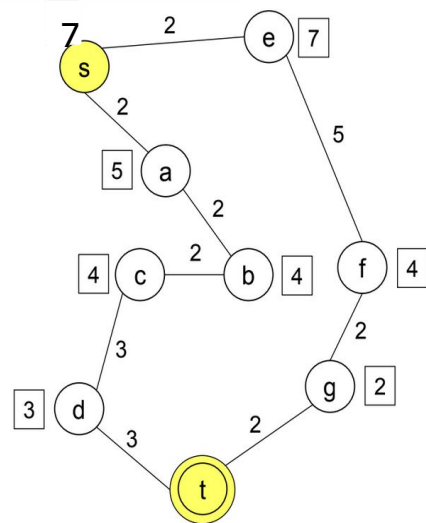
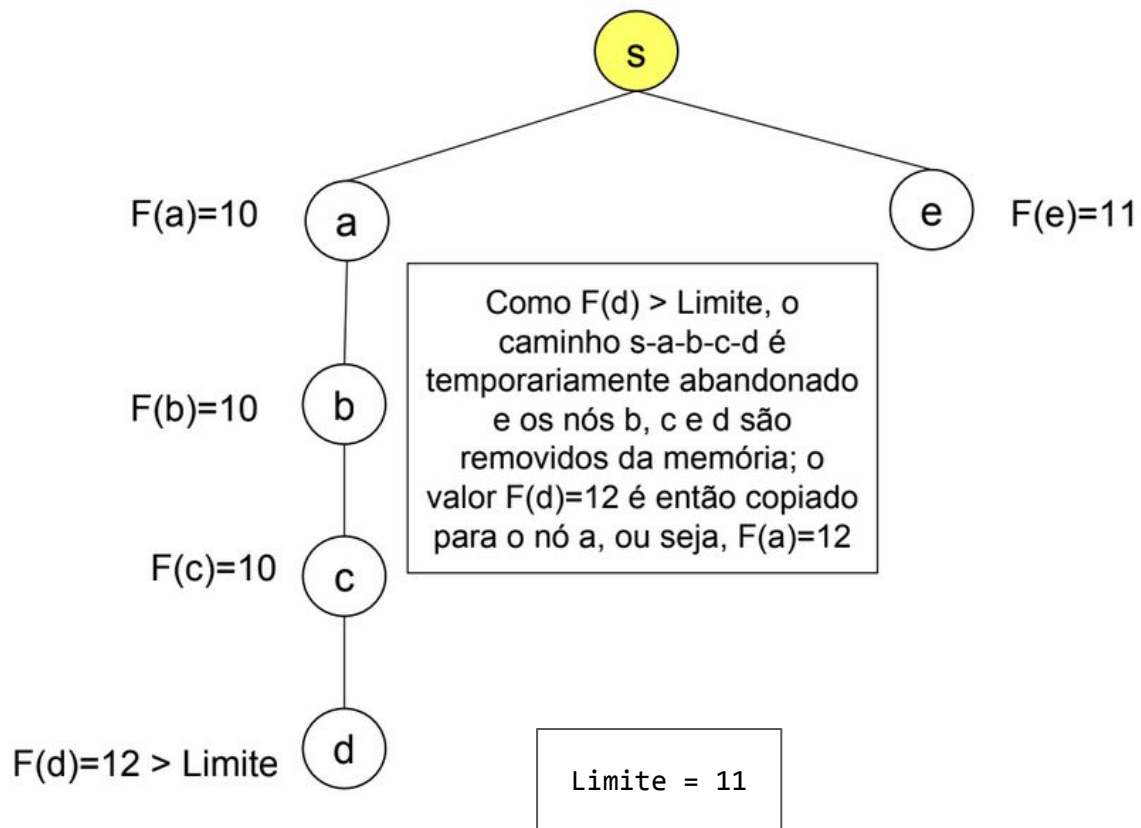
RBFS



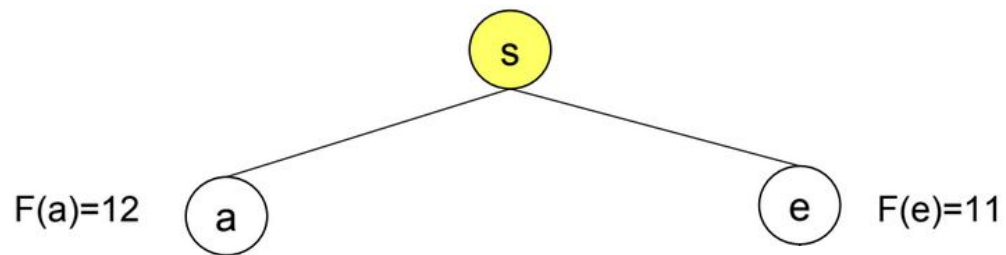
Limite = 11



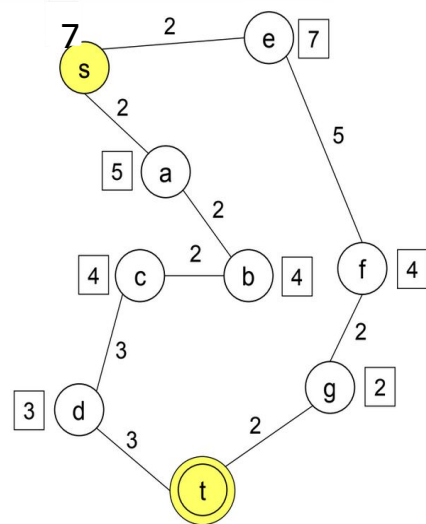
RBFS



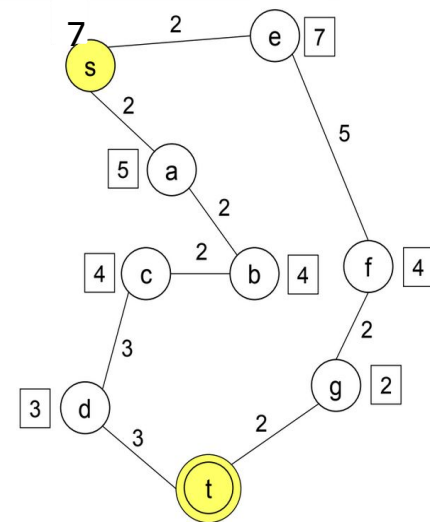
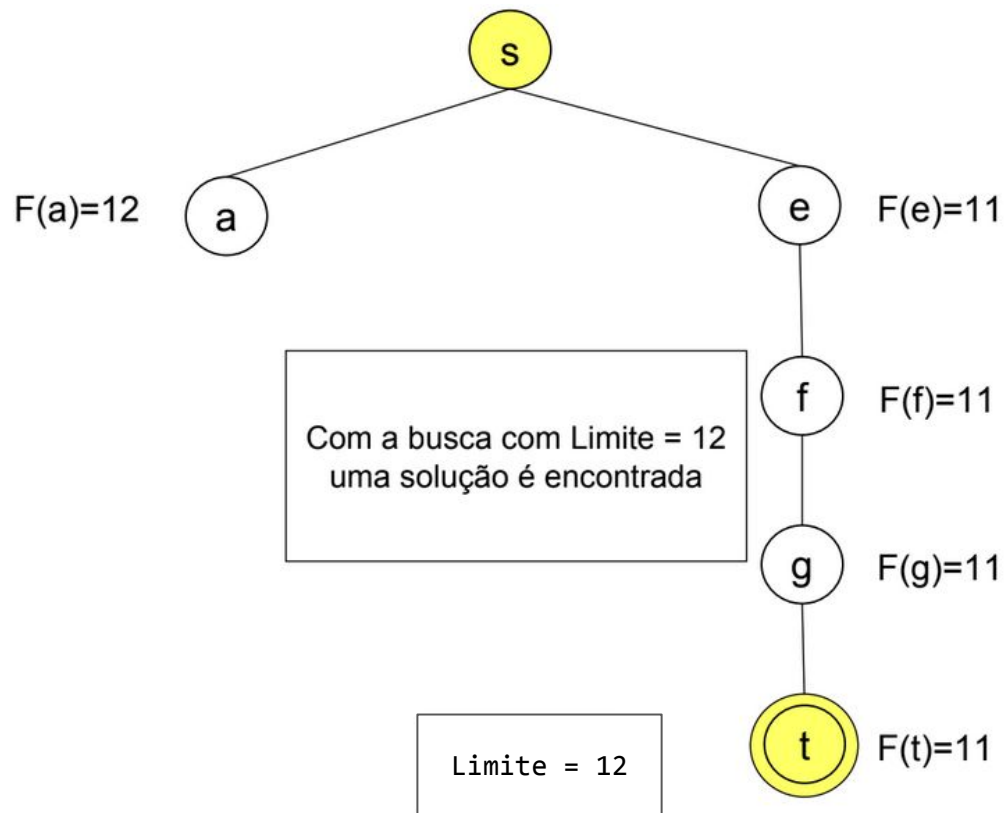
RBFS



Limite = 12



RBFS



Resumo

- Para uma dada heurística, A* irá expandir menos nós dentre todos os algoritmos.
- Sendo o A* proibitivo em espaço, IDA* e RBFS exploram abordagens iterativas para evitar o armazenamento de todo o grafo
- IDA* tende a desperdiçar computação em troca de pouco ganho de informação
- RBFS esquece as sub-árvores menos promissoras para resolver o ponto fraco do IDA*
- *A melhor escolha depende do quão complexo é o seu problema :)*

Pacman

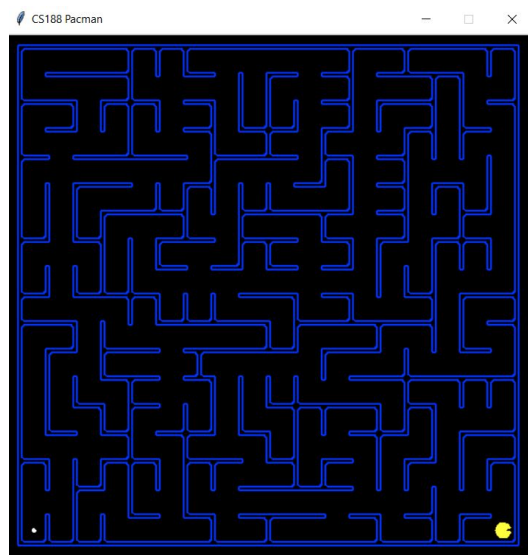
- Modelo de mundo do problema que estamos atacando no arquivo `searchAgents.py`

```
141 class PositionSearchProblem(search.SearchProblem):
142     """
143     A search problem defines the state space, start state, goal test, successor
144     function and cost function. This search problem can be used to find paths
145     to a particular point on the pacman board.
146
147     The state space consists of (x,y) positions in a pacman game.
148
149     Note: this search problem is fully specified; you should NOT change it.
150     """
151
152     def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None, warn=True, visualize=True):
153         """
154         Stores the start and goal.
155
156         gameState: A GameState object (pacman.py)
157         costFn: A function from a search state (tuple) to a non-negative number
158         goal: A position in the gameState
159         """
160         self.walls = gameState.getWalls()
161         self.startState = gameState.getPacmanPosition()
162         if start != None: self.startState = start
163         self.goal = goal
164         self.costFn = costFn
165         self.visualize = visualize
166         if warn and (gameState.getNumFood() != 1 or not gameState.hasFood(*goal)):
167             print('Warning: this does not look like a regular search maze')
```

Pacman (Ex 2)

- Implemente a **busca de custo uniforme** e a **busca A*** para solucionar um cenário limitado do Pacman, onde só há uma comida disponível na posição (1,1).
- Você vai editar apenas as funções `uniformCostSearch` e `aStarSearch` no arquivo `search.py`
 - Além de implementar a sua função heurística

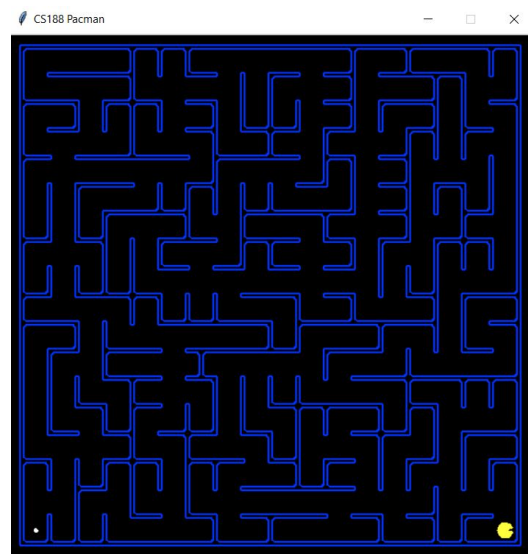
```
97 def uniformCostSearch(problem: SearchProblem):
98     """Search the node of least total cost first."""
99     """*** YOUR CODE HERE ***"""
100     util.raiseNotDefined()
101
102 def heuristic(state, problem=None):
103     """
104     A heuristic function estimates the cost from the current state to the nearest
105     goal in the provided SearchProblem. This heuristic is trivial.
106     """
107     return 0
108
109 def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
110     """Search the node that has the lowest combined cost and heuristic first."""
111     """*** YOUR CODE HERE ***"""
112     util.raiseNotDefined()
```



Pacman (Ex 2)

- Cada execução retorna o número de nós explorados. Com essa informação:
 - Compare o UCS e o A*
 - Compare as heurística baseadas na distância de manhattan e distância euclidiana
 - Sugira sua própria heurística (opcional)

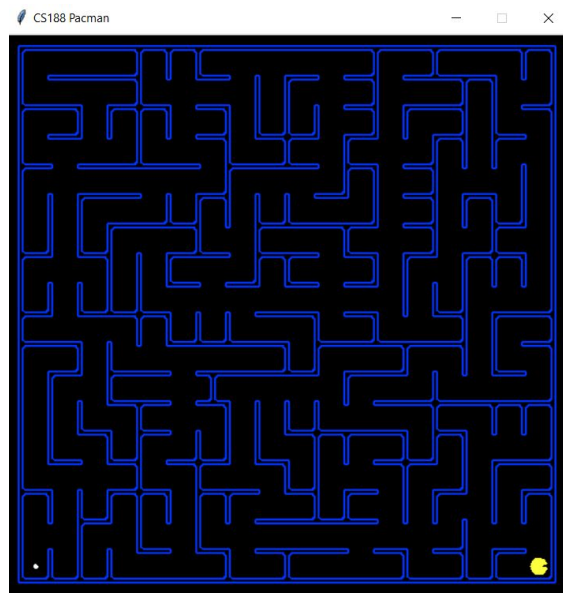
```
97 def uniformCostSearch(problem: SearchProblem):
98     """Search the node of least total cost first."""
99     """*** YOUR CODE HERE ***"""
100     util.raiseNotDefined()
101
102 def heuristic(state, problem=None):
103     """
104     A heuristic function estimates the cost from the current state to the nearest
105     goal in the provided SearchProblem. This heuristic is trivial.
106     """
107     return 0
108
109 def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
110     """Search the node that has the lowest combined cost and heuristic first."""
111     """*** YOUR CODE HERE ***"""
112     util.raiseNotDefined()
```



Pacman (Ex 2)

- Lembre-se: estrutura de dados já implementada em `util.py`

```
170 class PriorityQueue:
171     """
172     Implements a priority queue data structure. Each inserted item
173     has a priority associated with it and the client is usually interested
174     in quick retrieval of the lowest-priority item in the queue. This
175     data structure allows O(1) access to the lowest-priority item.
176     """
177     def __init__(self):
178         self.heap = []
179         self.count = 0
180
181     def push(self, item, priority):
182         entry = (priority, self.count, item)
183         heapq.heappush(self.heap, entry)
184         self.count += 1
185
186     def pop(self):
187         (_, _, item) = heapq.heappop(self.heap)
188         return item
189
190     def isEmpty(self):
191         return len(self.heap) == 0
192
193     def update(self, item, priority):
194         # If item already in priority queue with higher priority, update its priority and rebuild the heap.
195         # If item already in priority queue with equal or lower priority, do nothing.
196         # If item not in priority queue, do the same thing as self.push.
197         for index, (p, c, i) in enumerate(self.heap):
198             if i == item:
199                 if p <= priority:
200                     break
201                 del self.heap[index]
202                 self.heap.append((priority, c, item))
203                 heapq.heapify(self.heap)
204                 break
205         else:
206             self.push(item, priority)
207
```



Pacman (Ex 2)

- Seu código deve retornar a lista de ações necessárias para alcançar a comida. Lembra do tinyMazeSearch?

```
def tinyMazeSearch(problem):  
    """  
    Returns a sequence of moves that solves tinyMaze. For any other maze, the  
    sequence of moves will be incorrect, so only use this for tinyMaze.  
    """  
    from game import Directions  
    s = Directions.SOUTH  
    w = Directions.WEST  
    return [s, s, w, s, w, w, s, w]
```

- Você pode executar seus algoritmos, digitando no terminal:

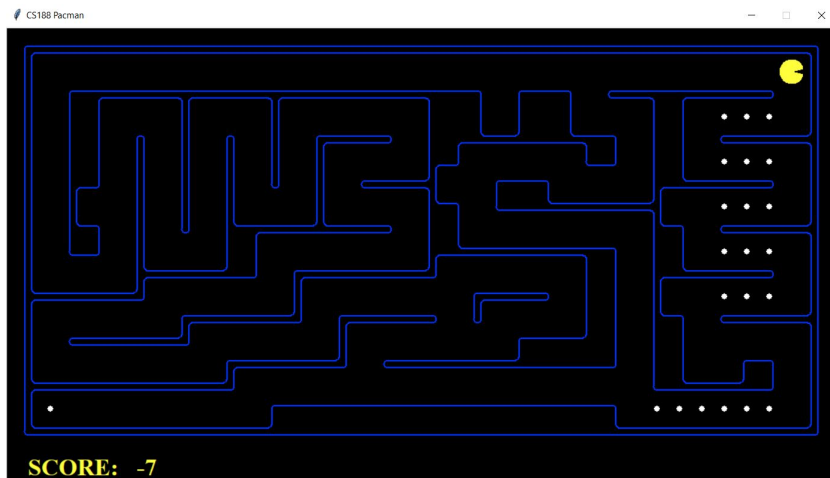
```
python pacman.py -l bigMaze -p SearchAgent -a fn=ucs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=astar,heuristic=myHeuristic
```

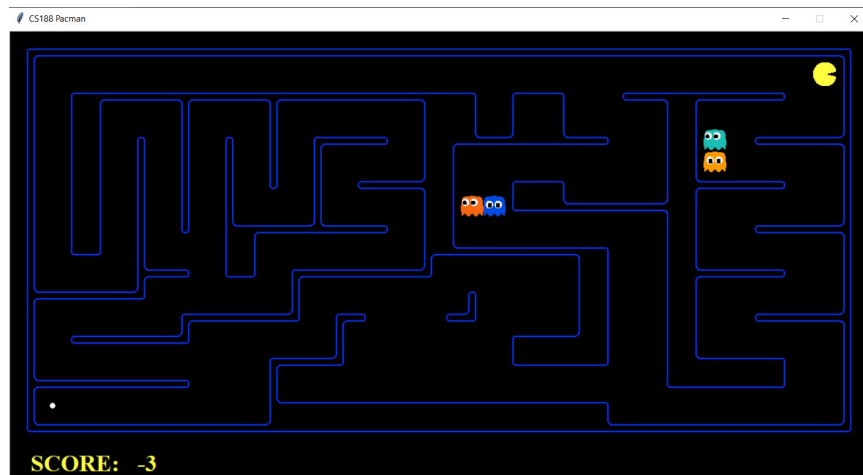
Pacman (Ex 3)

- Aplique sua **busca de custo uniforme** aos seguintes cenários

```
python pacman.py -l mediumDottedMaze
```



```
python pacman.py -l mediumScaryMaze
```



Pacman (Ex 3)

- Aplique sua busca de custo uniforme aos seguintes cenários e **reporte o score final considerando duas funções de custo**

Leia o script `searchAgents.py`

Penaliza movimentos mais à esquerda

```
python pacman.py -l mediumDottedMaze  
-p StayEastSearchAgent
```

```
234 class StayEastSearchAgent(SearchAgent):  
235     """  
236     An agent for position search with a cost function that penalizes being in  
237     positions on the West side of the board.  
238  
239     The cost function for stepping into a position (x,y) is 1/2^x.  
240     """  
241     def __init__(self):  
242         self.searchFunction = search.uniformCostSearch  
243         costFn = lambda pos: .5 ** pos[0]  
244         self.searchType = lambda state: PositionSearchProblem(state, costFn, (1, 1), None, False)  
245
```

Penaliza movimentos mais à direita

```
python pacman.py -l mediumDottedMaze  
-p StayWestSearchAgent
```

```
246 class StayWestSearchAgent(SearchAgent):  
247     """  
248     An agent for position search with a cost function that penalizes being in  
249     positions on the East side of the board.  
250  
251     The cost function for stepping into a position (x,y) is 2^x.  
252     """  
253     def __init__(self):  
254         self.searchFunction = search.uniformCostSearch  
255         costFn = lambda pos: 2 ** pos[0]  
256         self.searchType = lambda state: PositionSearchProblem(state, costFn)
```

Pacman: Entregas

https://docs.google.com/document/d/1zj4r0FCO6K4gbWjzTxK-E2X6THR_OIVNfG-3BrjYH3c/edit?usp=sharing

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Instituto de Ciências Exatas e Informática

Curso de Ciência da Computação - Coração Eucarístico

Profa.: Camila Laranjeira - mila.laranjeira@gmail.com

Disciplina: Inteligência Artificial / 1o Semestre de 2022

Aluna(o):

Exercício Prático 01 - Pacman #1

Instruções:

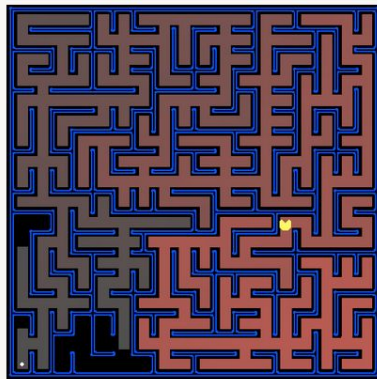
- Consulte os slides da disciplina para maiores detalhes sobre a implementação
- O código fonte base está no Canvas sob o título `pacman.zip`
- Você deve entregar seu código em um `.zip` que inclua esse documento preenchido

Pacman

- Essa atividade é uma pequena parte do projeto de Berkeley University:
 - <https://inst.eecs.berkeley.edu/~cs188/sp22/project1/>
- Fique a vontade para explorar outros desafios!

Project 1: Search

Due: Thursday, Feb 3 at 10:59 pm



All those colored walls,
Mazes give Pacman the blues,
So teach him to search.