CEFET/RJ

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

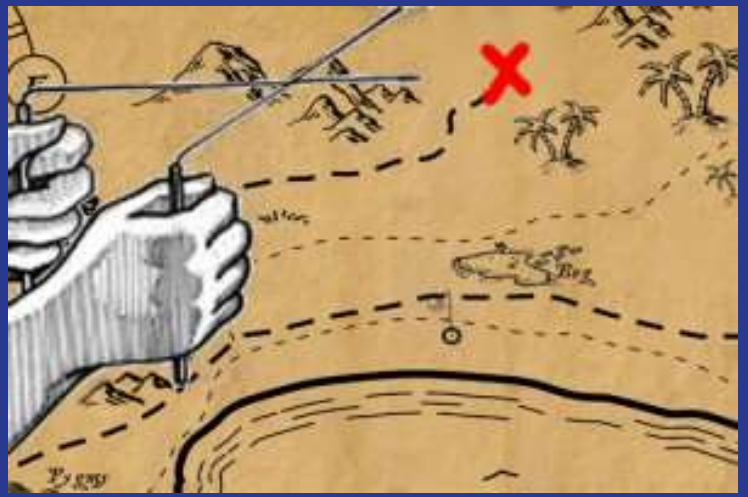
GCC1734 - INTELIGÊNCIA ARTIFICIAL

Eduardo Bezerra (CEFET/RJ)
ebezerra@cefet-rj.br

Créditos

- Essa apresentação é uma tradução e/ou adaptação feita pelo prof. Eduardo Bezerra (ebezerra@cefet-rj.br) do material cuja autoria é dos professores Dan Klein e Pieter Abbeel (UC Berkeley).
- O material original é usado no curso CS188 (Introduction to Artificial Intelligence).
  - https://inst.eecs.berkeley.edu/~cs188

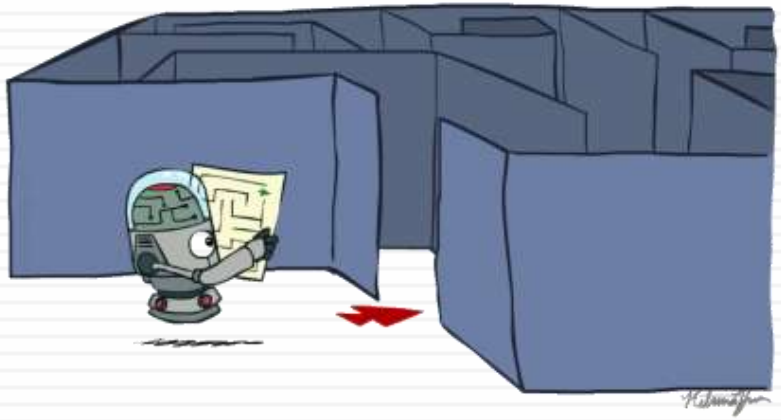
BUSCA COM HEURÍSTICAS



Visão Geral

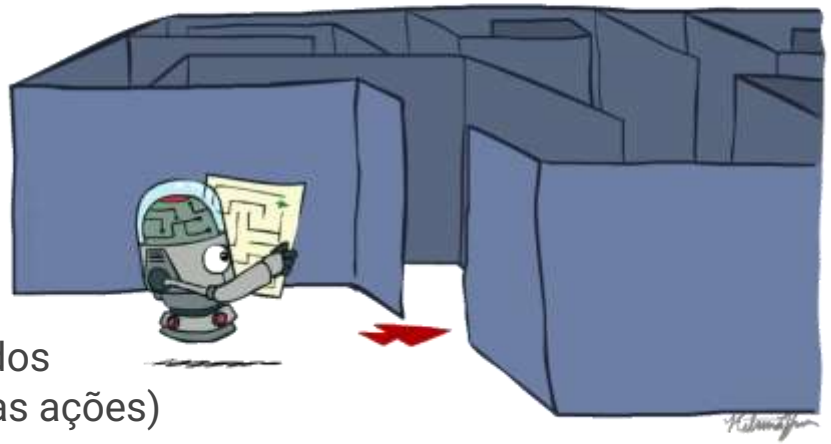
- Revisão
- Busca com heurísticas
  - Conceito de heurística
  - Busca Gulosa (*Greedy Search*)
  - Busca A* (*A-Star Search*)
- Tree Search vs Graph Sesarch
- Otimalidade e propriedades do A*
- Criação de heurísticas

Revisão: Busca



Busca

- Problema de Busca:
  - Estados (configurações do mundo)
  - Ações e custos
  - Função sucessora (dinâmica do mundo)
  - Estado inicial e teste de objetivo



- Árvore de Busca:
  - Nós: representam planos para alcançar estados
  - Planos possuem custos (soma dos custos das ações)
- Algoritmo de Busca:
  - Sistematicamente constrói uma árvore de busca
  - Define uma ordem sobre os elementos na borda (nós não explorados)
  - É ótimo se encontra um plano de custo mínimo

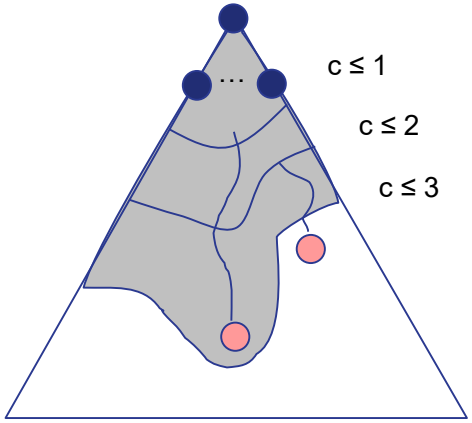
Filas, filas e mais filas...

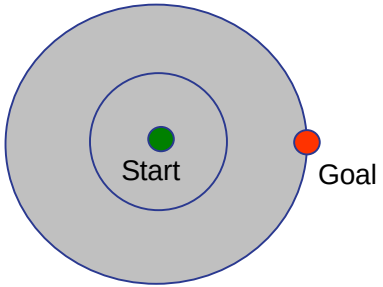
- Todos os algoritmos que vimos são idênticos, a menos de suas estratégias de manipulação da fronteira.
  - Conceitualmente, todas as froteiras são filas de prioridades (*priority queues*).
  - Na prática, para DFS e BFS, é possível evitar o custo log(n) de uma fila de prioridades, ao usar pilhas e filas.
  - É até possível realizar uma implementação genérica que receba um objeto FilaPrioridades.

Revisão: UCS

Revisão: UCS

- Estratégia: expandir nó com o menor custo de caminho (i.e., menor valor da função g(n)).
- Vantagem: UCS é completa e ótima!
- Desvantagem:
  - Explora espaço de busca em todas as “direções”
  - Não usa qualquer informação acerca da localização do objetivo para guiar a busca.
    - É um tipo de *busca sem informação*



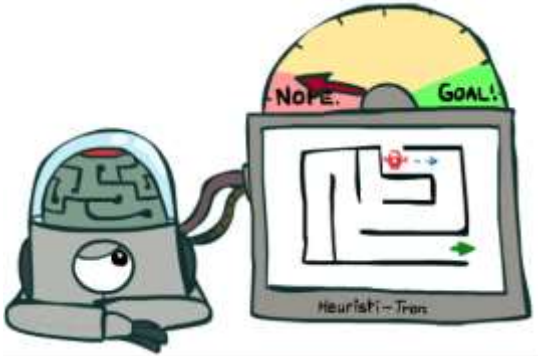


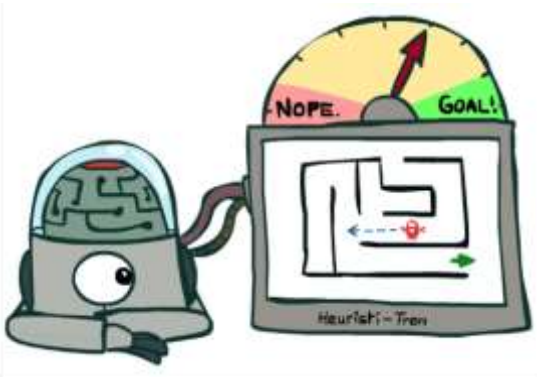
[Demo: contours UCS empty (L3D1)]
[Demo: contours UCS pacman small maze (L3D3)]

Busca com Informação (*Informed Search*)

Função Heurística

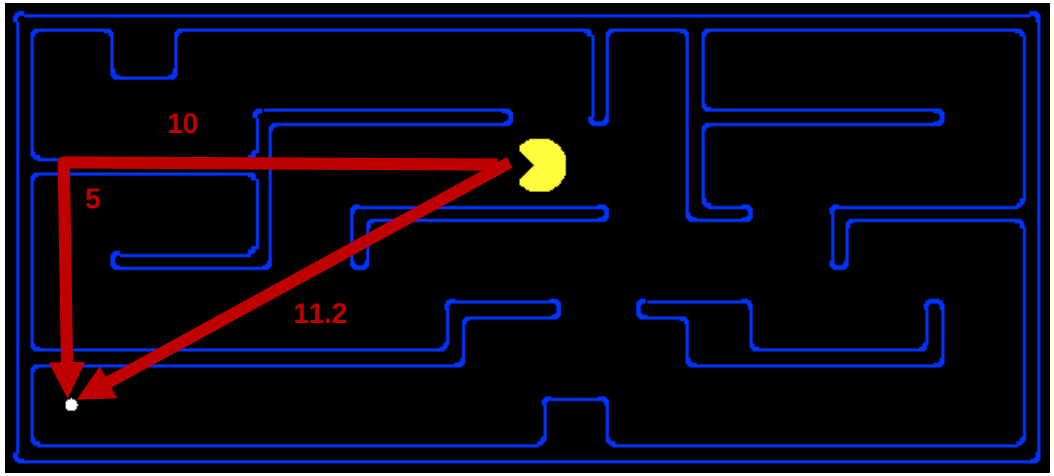
- Uma **função heurística**...
  - ...provê uma **estimativa** de quão próximo um estado se encontra de um objetivo.
  - ...é projetada (personalizada) para cada problema de busca em particular.





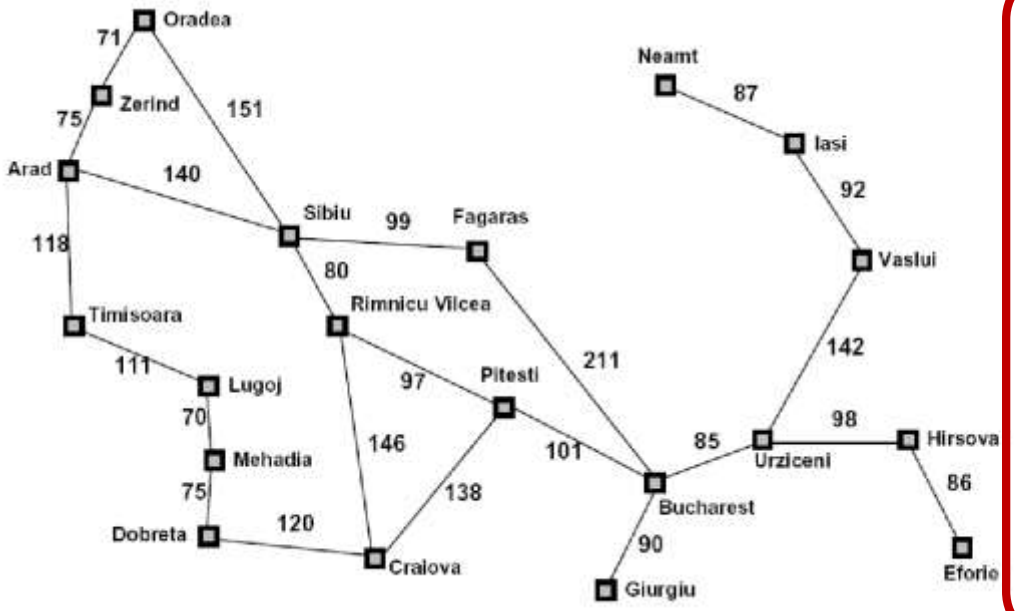
Exemplo: Função Heurística (Pac-Man)

- h(n): distância de Manhatan ou distância Euclidiana



Exemplo: Função Heurística (Cidades da Romênia)

h(n): distância em linha reta desde n até Bucareste.



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(n)

Busca com Informação (ou Busca Heurística)

- Utiliza conhecimento específico sobre o problema para encontrar soluções de forma mais eficiente do que a busca sem informação.
  - Conhecimento específico: além da especificação do problema.
- Abordagem geral: Busca pela Melhor Escolha (*Best-First Search*).
  - Utiliza uma **função de avaliação**, aplicável a cada nó.
  - Expande o nó que tem o valor mais baixo para a função de avaliação (i.e., expande o nó considerado mais promissor).
  - Dependendo da função de avaliação escolhida, a estratégia de busca muda.

Busca pela Melhor Escolha (*Best-First Search*)

- Ideia: usar uma função de avaliação *f(n)* para cada nó.
  - estimativa do quanto aquele nó é desejável
  - Expandir nó mais desejável que ainda não foi expandido
- Implementação: ordenar nós na borda em ordem decrescente do valor da função de avaliação.
  - Borda é uma lista de prioridades, ordenada de acordo com o valor de *f(n)*.
- Casos especiais:
  - **Busca Gulosa (*greedy best-first search, GBFS*)**
  - **Busca A* (*A* search*)**

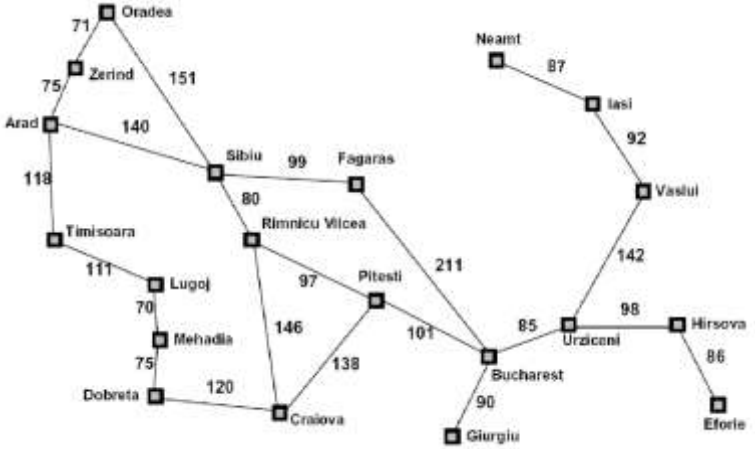
Busca Gulosa *(Greedy Search)*

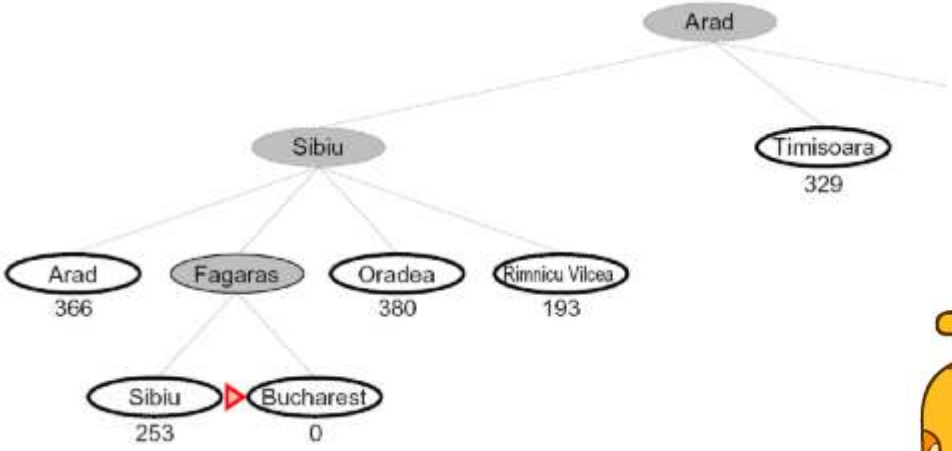
Busca Gulosa (*Greedy Search*)

- Estratégia: expandir o nó que parece ser o mais próximo ao objetivo, de acordo com uma função heurística, *h(n)*.
  - A esperança é que esse nó leve ao objetivo mais rapidamente.
- A função *h(n)* é uma estimativa do custo desde o nó *n* até o nó objetivo.
- Na busca gulosa, *f(n) = h(n)*.

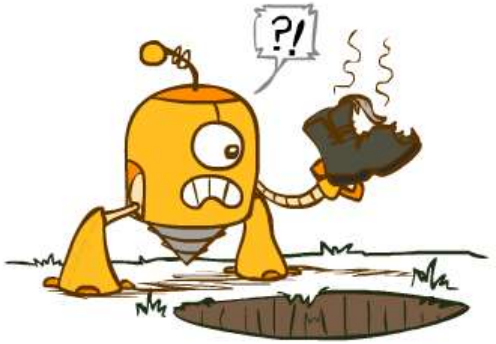
Exemplo: Romênia

- Busca gulosa: expandir o nó que *parece* mais próximo ao objetivo…





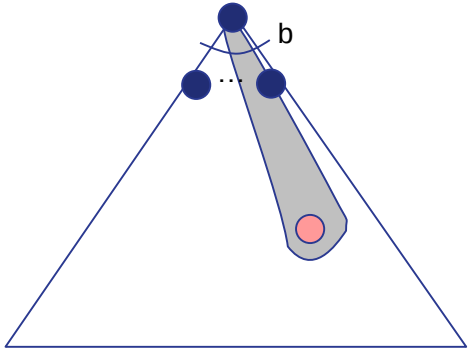
- O que pode dar errado?



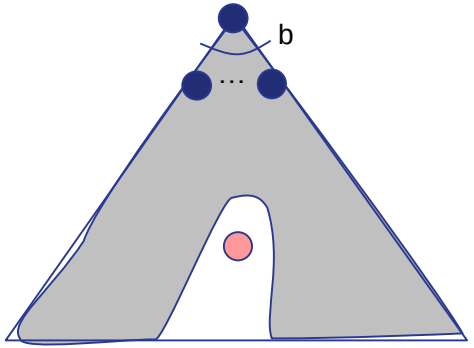
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Busca Gulosa

- Estratégia: expandir o nó que parece estar mais próximo de um estado objetivo.
  - Heurística: estimativa da distância ao objetivo mais próximo, para cada estado.



- Um caso comum:
  - GBFS leva diretamente a um objetivo subótimo



- No pior caso: similar a um DFS "piorado".

Busca Gulosa: propriedades

- Completa? Não – pode ficar presa em loops (ex., ir para Fagaras: Iasi → Neamt → Iasi → Neamt → ...)
- Tempo? $O(b^m)$ no pior caso, mas uma boa função heurística pode levar a uma redução substancial no tempo de execução
- Espaço? $O(b^m)$ – mantém todos os nós na memória
- Ótima? Não
  - Pois escolhe ação **considerando somente o estado atual.**
  - Pode haver um plano globalmente melhor que siga algumas opções localmente piores em

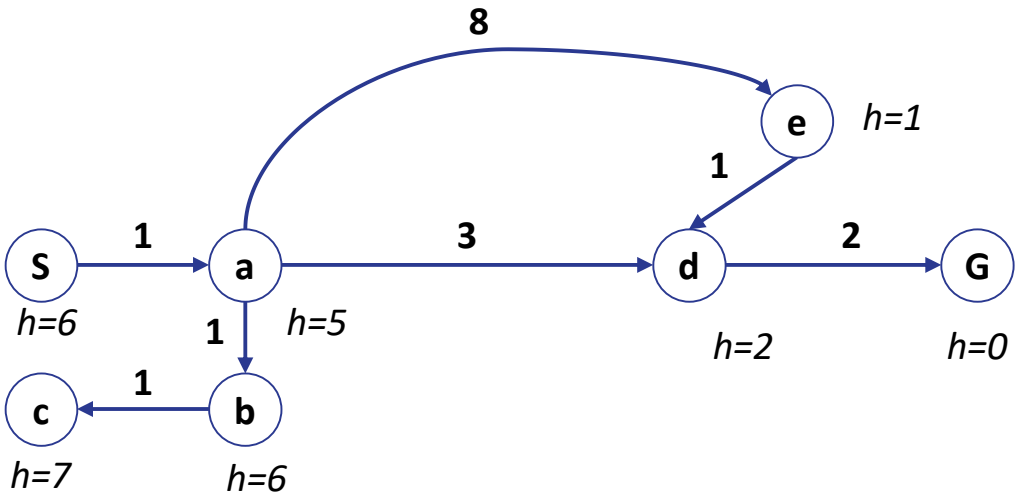
Busca A* (A* Search)

Busca A*

- Usa uma função de avaliação da forma $f(n) = g(n) + h(n)$
  - $g(n)$ = custo até o momento para alcançar n
  - $h(n)$ = custo estimado de n até o objetivo
  - $f(n)$ = custo total estimado do caminho até o objetivo através de n.
- Justificativa: se queremos encontrar a solução menos custosa, uma ação razoável é expandir o nó com o menor valor de f(n).
- A* expande nós de maneira similar à UCS.

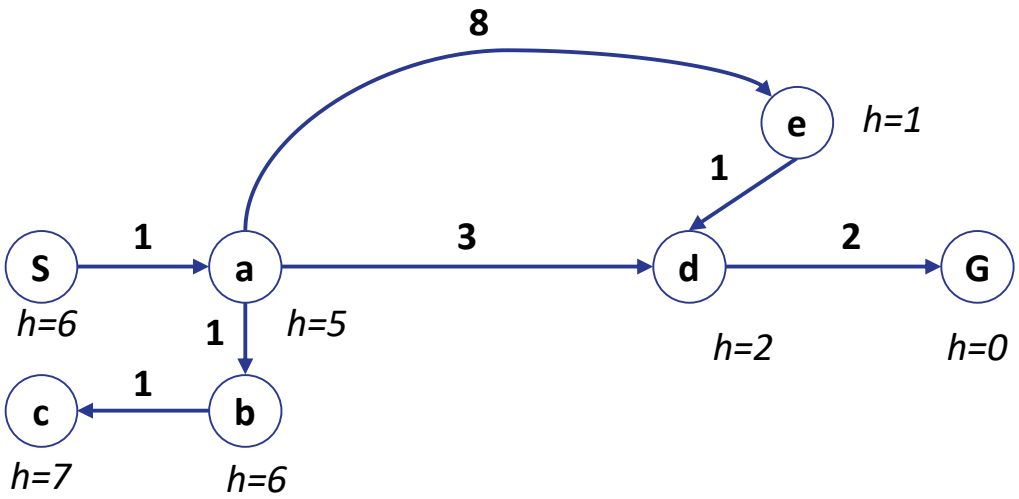
A* combina UCS e Busca Gulosa

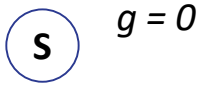
- Busca de Custo Uniforme ordena por custo de caminho, g(n)
- Busca Gulosa ordena por proximidade estimada, h(n)
- Busca A* ordena pela soma: f(n) = g(n) + h(n)



A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

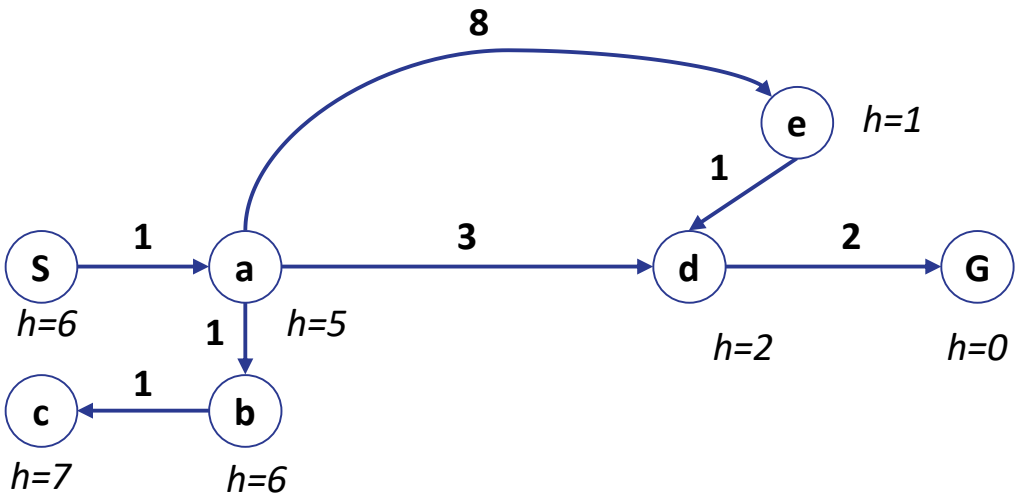


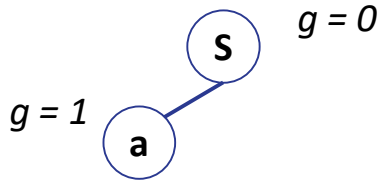


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

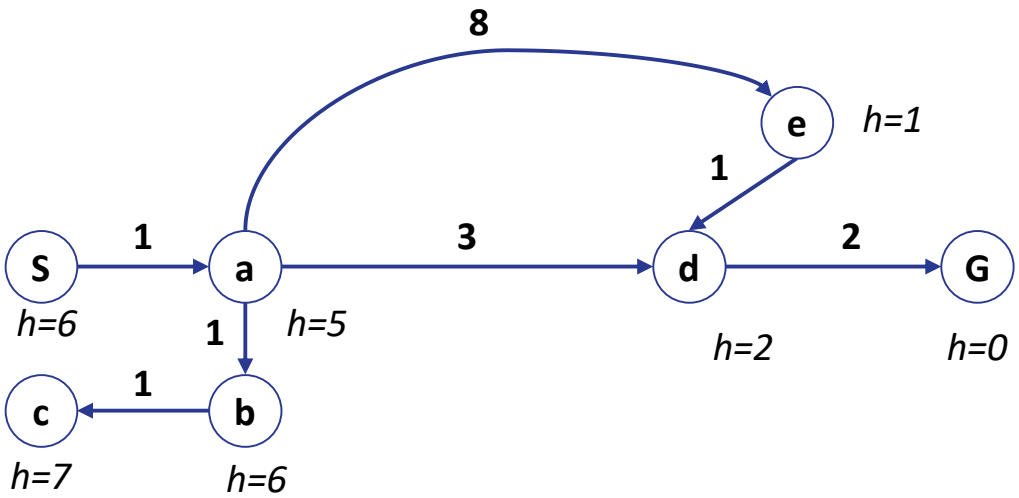


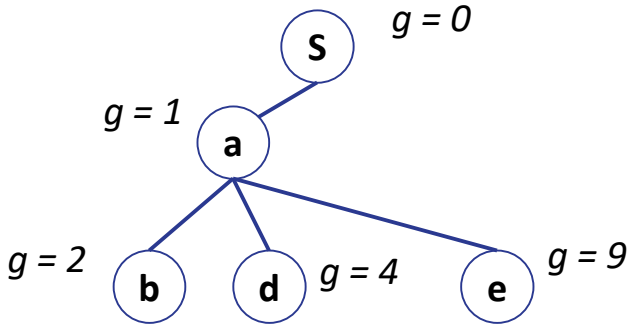


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

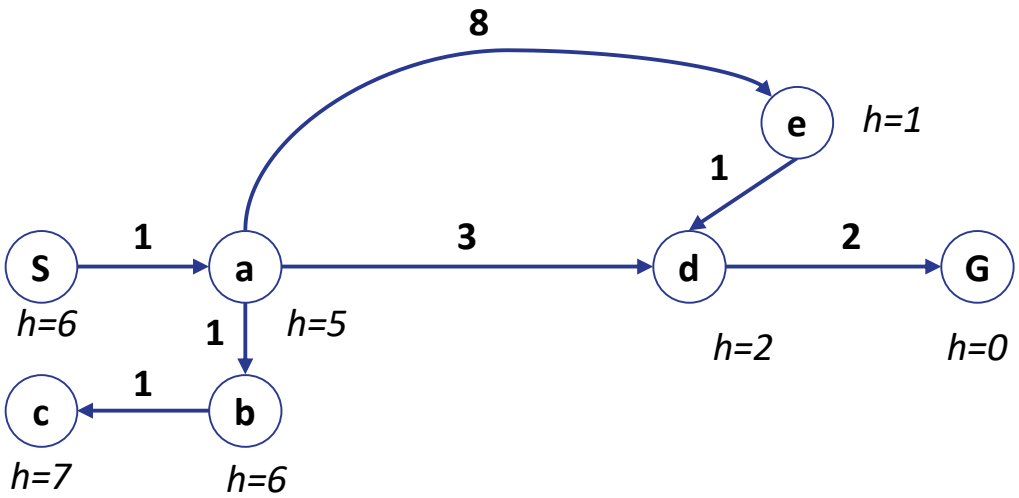


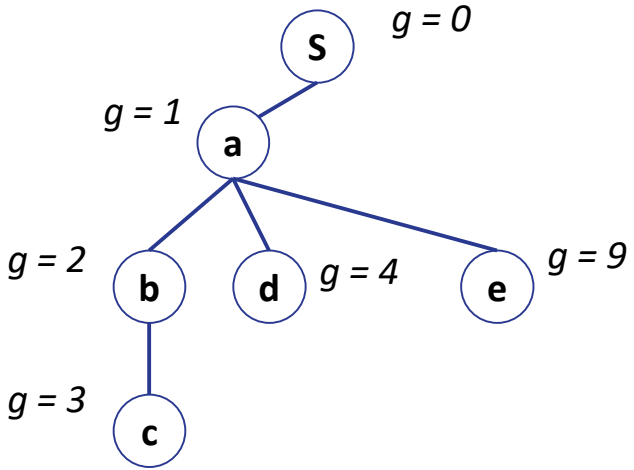


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

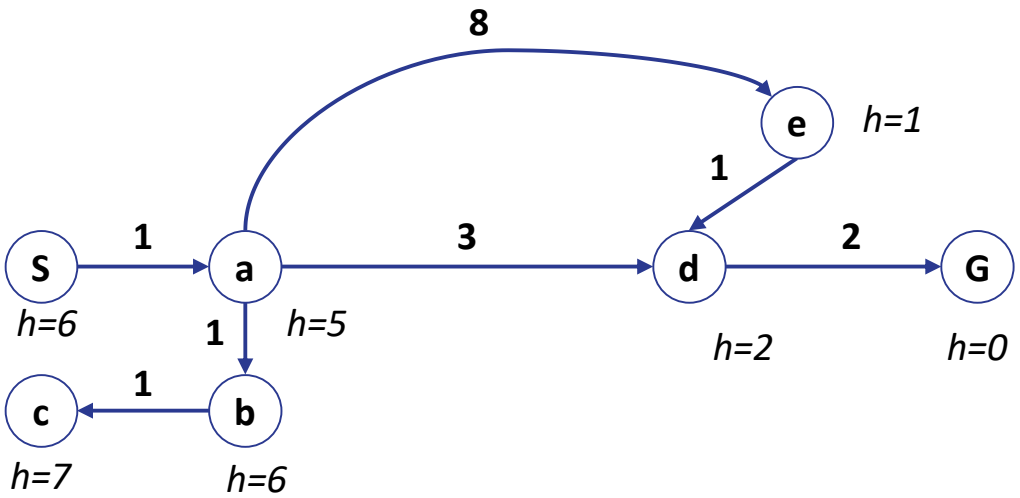


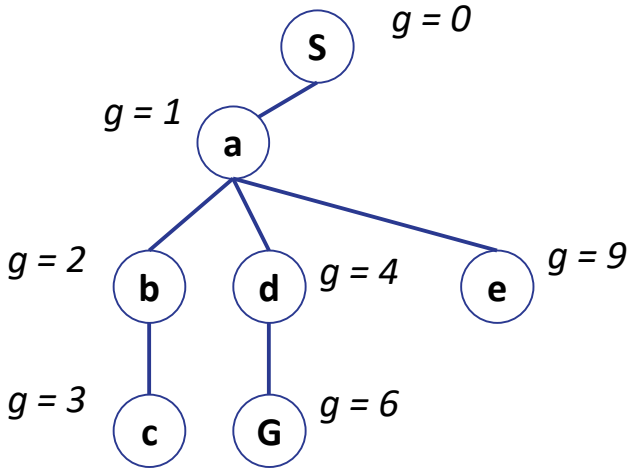


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

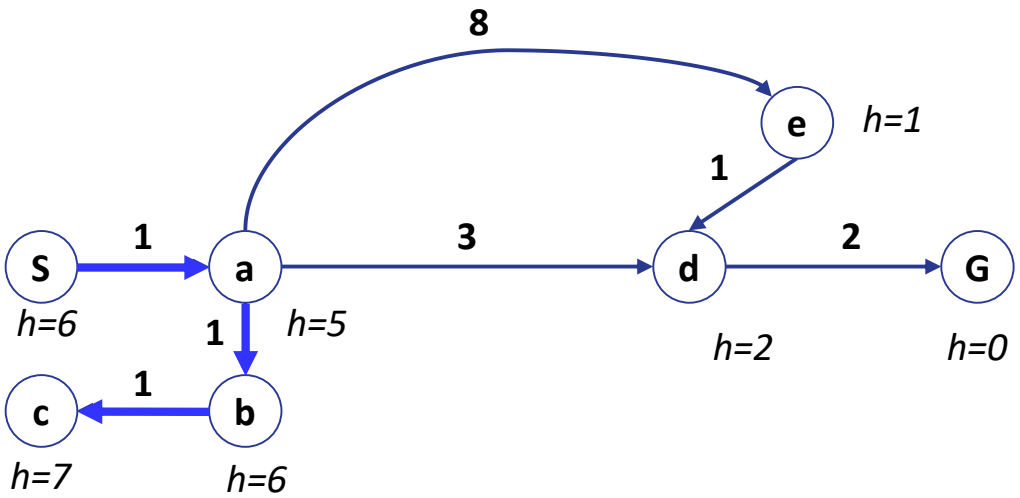


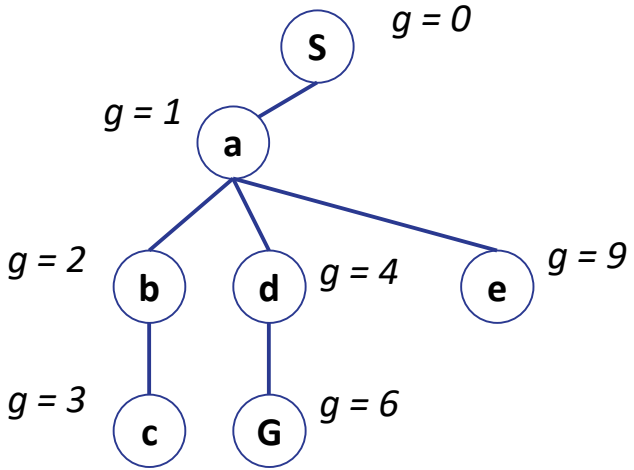


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca de Custo Uniforme ordena por custo de caminho, g(n)

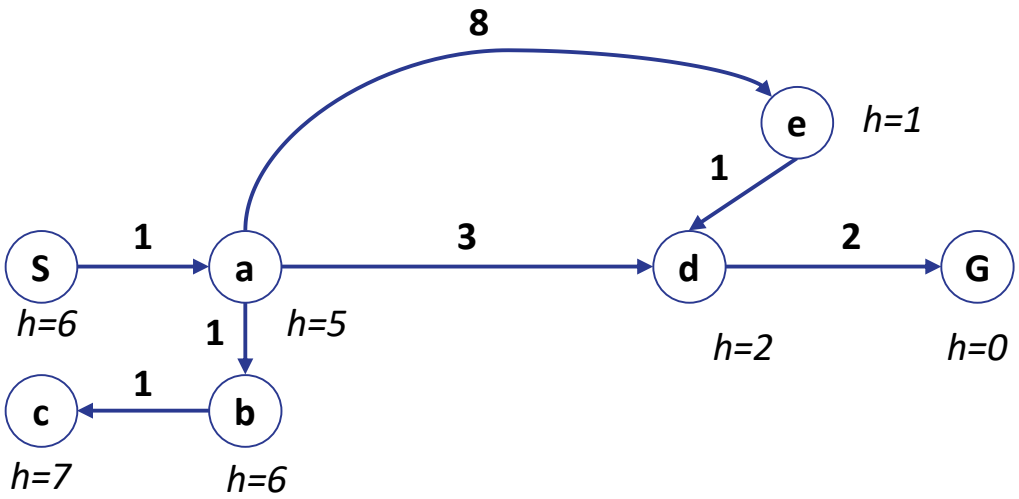




Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

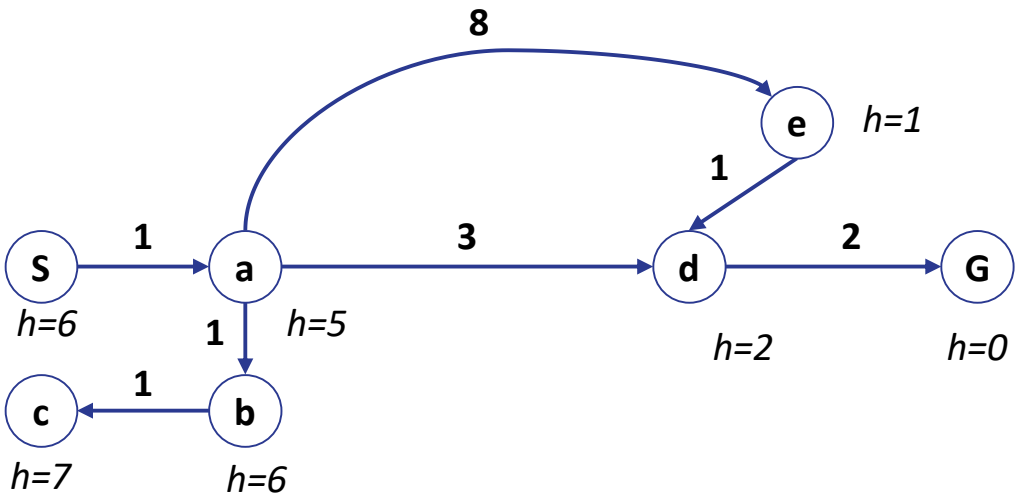
- Busca Gulosa ordena por proximidade estimada, h(n)

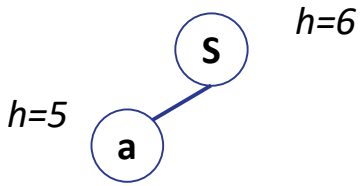


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca Gulosa ordena por proximidade estimada, h(n)

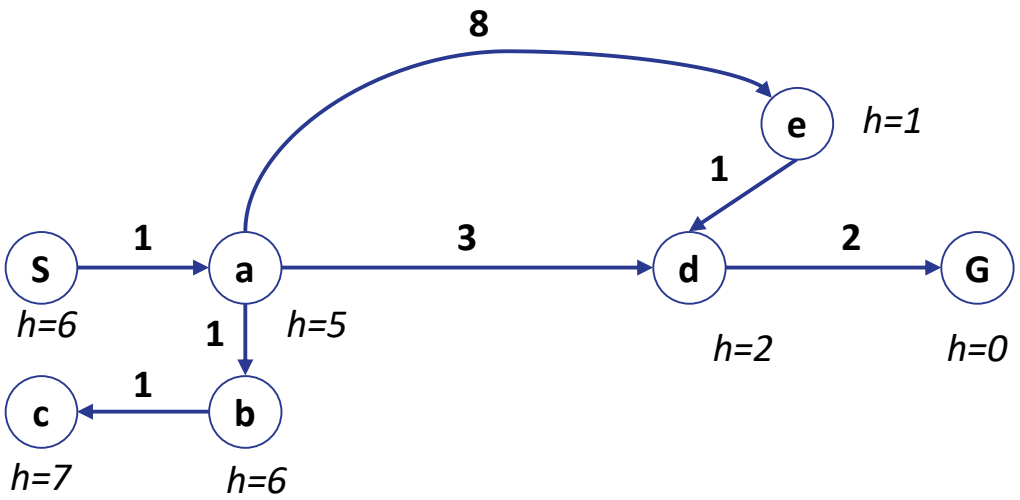


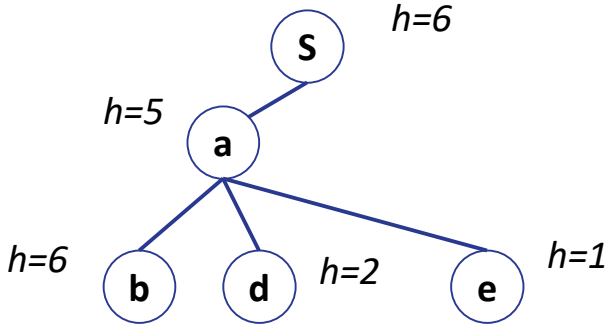


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca Gulosa ordena por proximidade estimada, h(n)

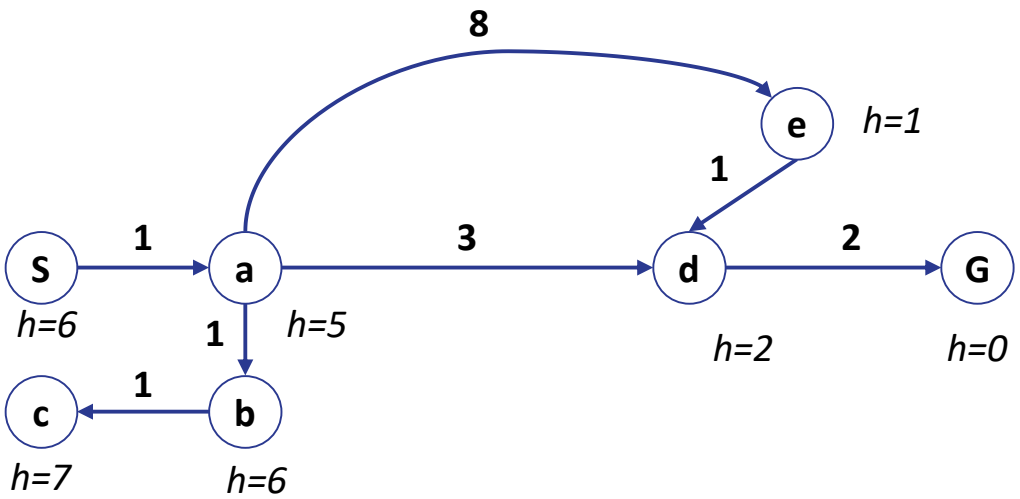


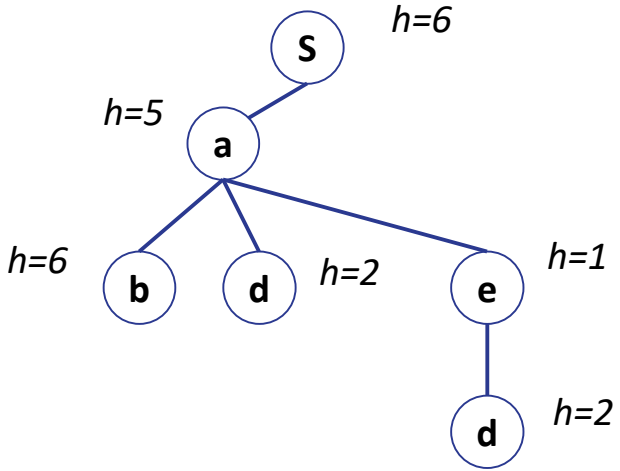


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca Gulosa ordena por proximidade estimada, h(n)

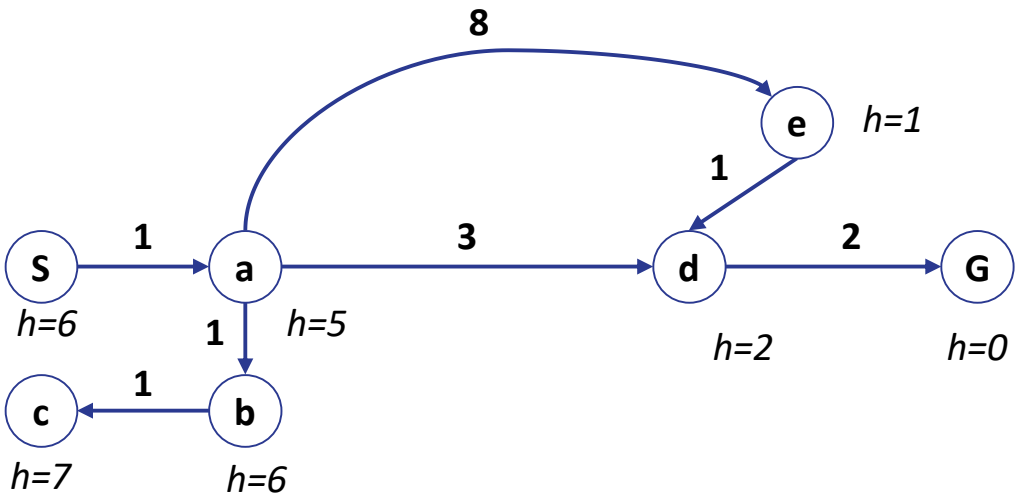


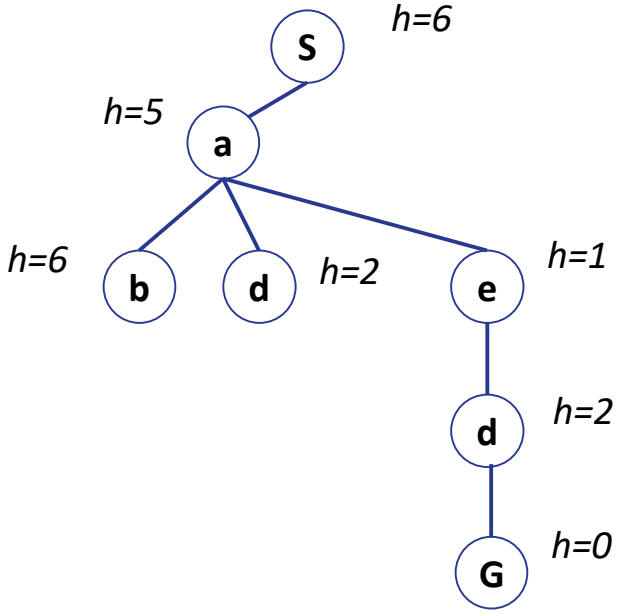


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca Gulosa ordena por proximidade estimada, h(n)

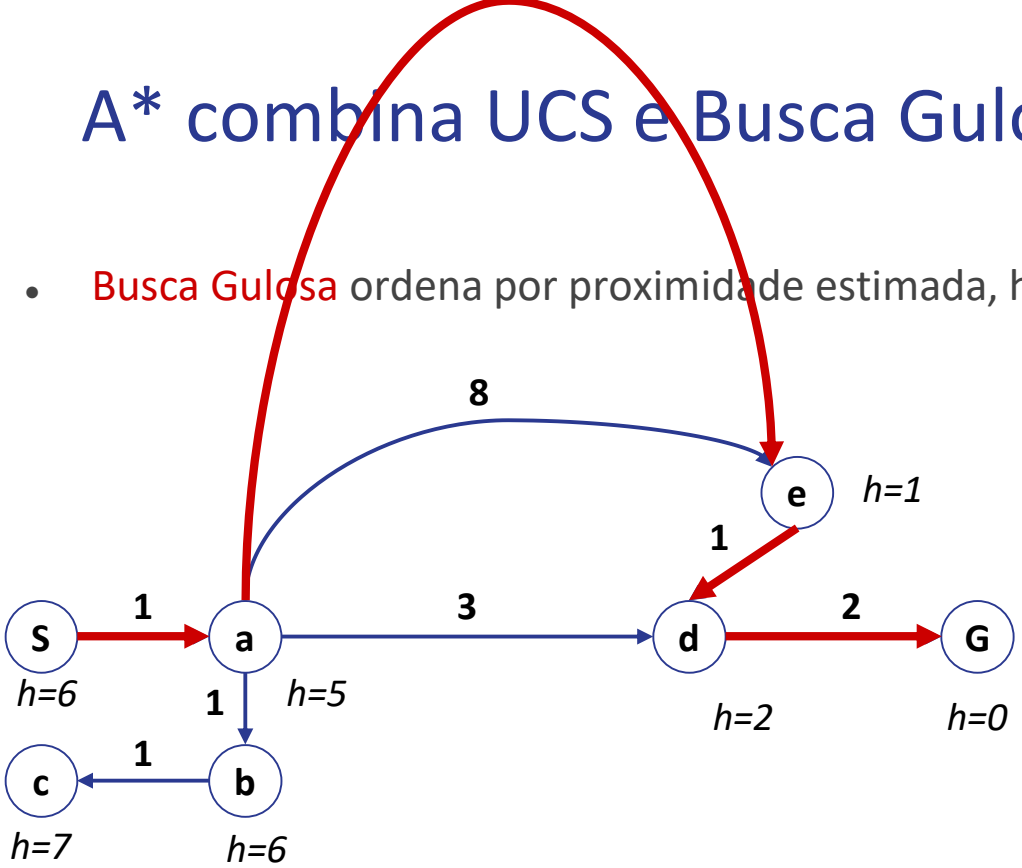


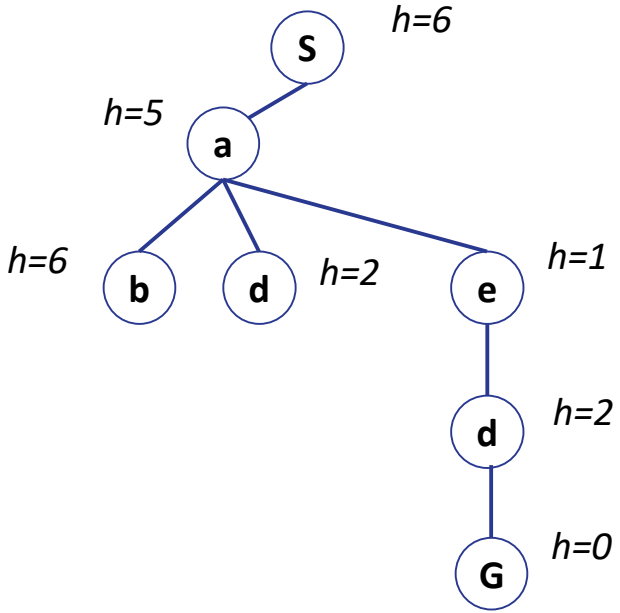


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca Gulosa ordena por proximidade estimada, h(n)

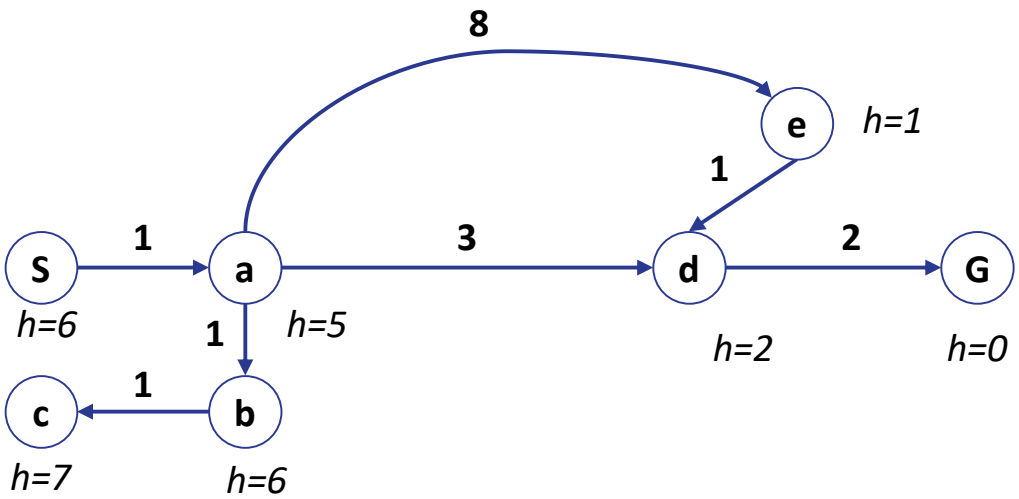


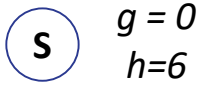


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca A* ordena pela soma: f(n) = g(n) + h(n)

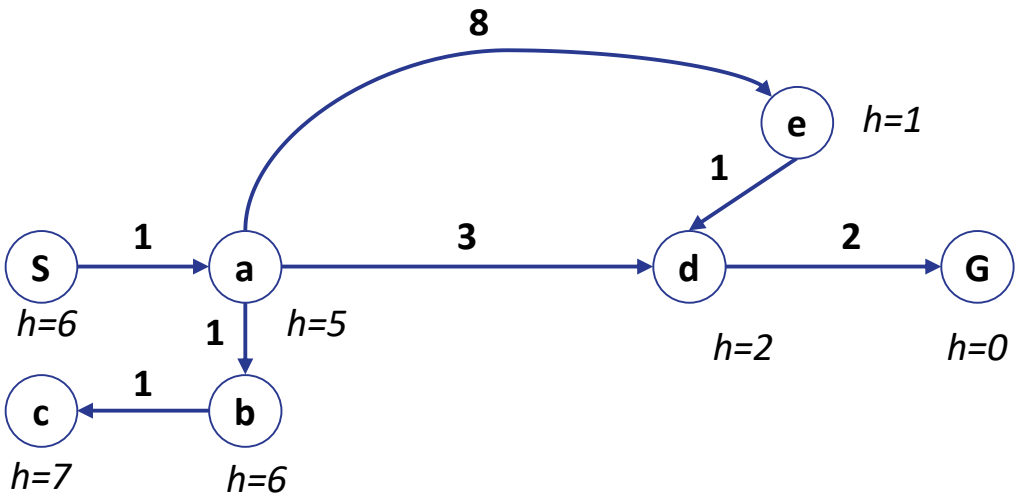


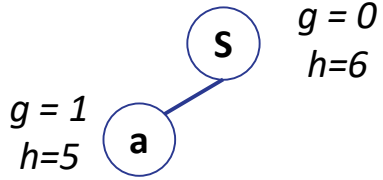


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca A* ordena pela soma: f(n) = g(n) + h(n)

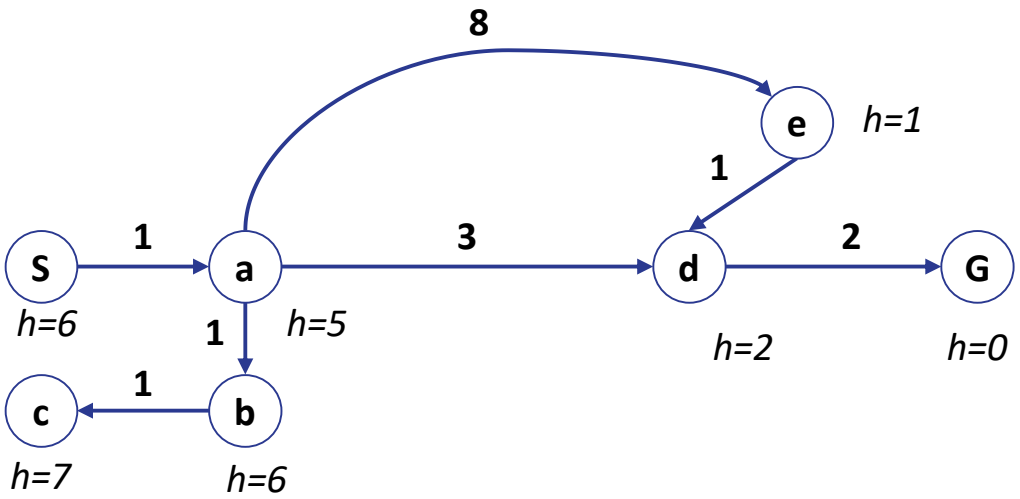


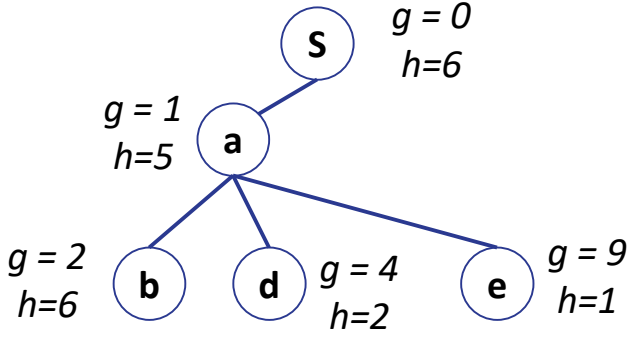


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca A* ordena pela soma: f(n) = g(n) + h(n)

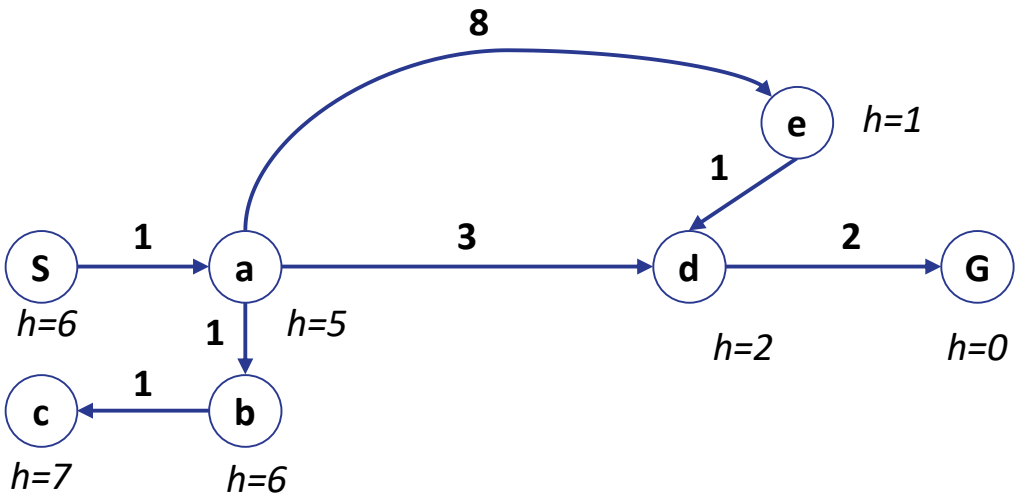


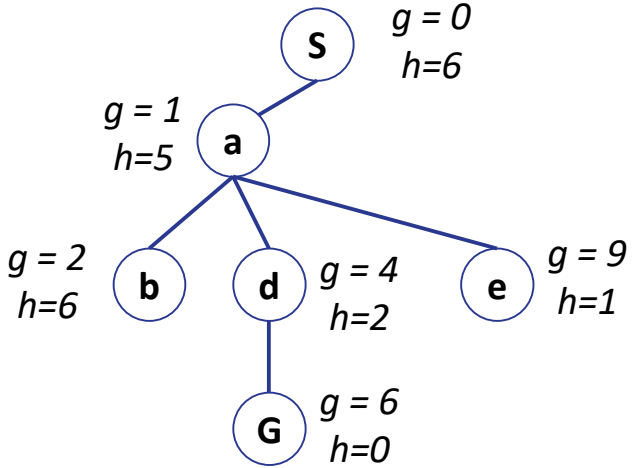


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca A* ordena pela soma: f(n) = g(n) + h(n)

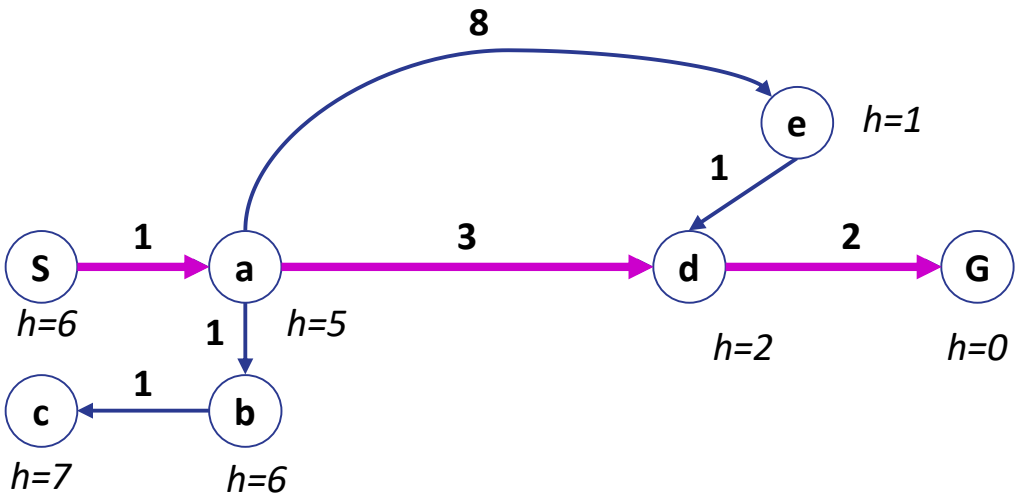


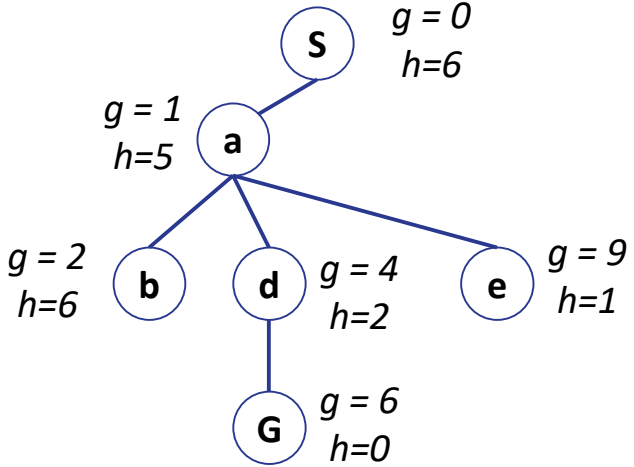


Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

- Busca A* ordena pela soma: f(n) = g(n) + h(n)

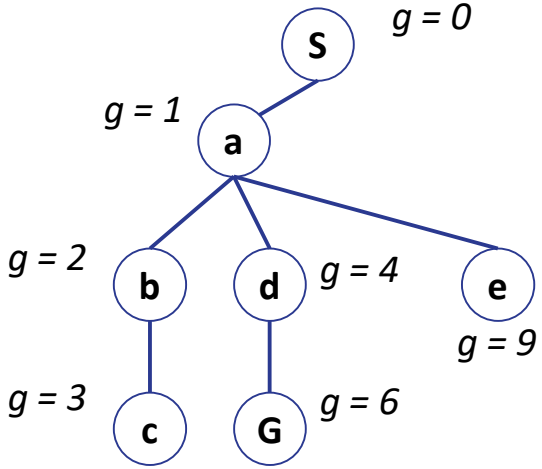




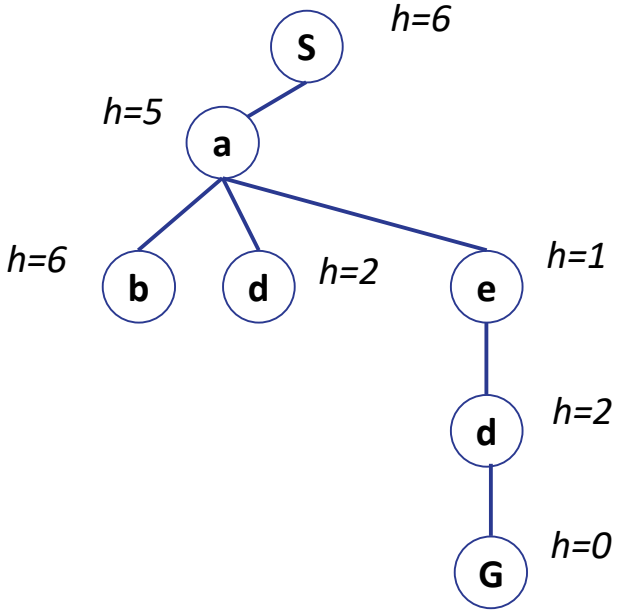
Autor do exemplo: Teg Grenager

A* combina UCS e Busca Gulosa

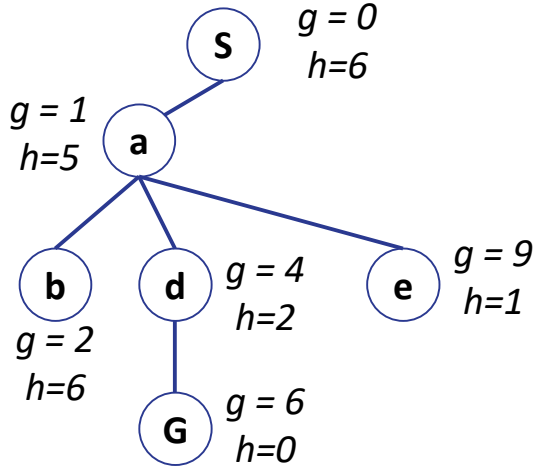
Busca UCS



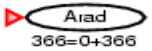
Busca gulosa

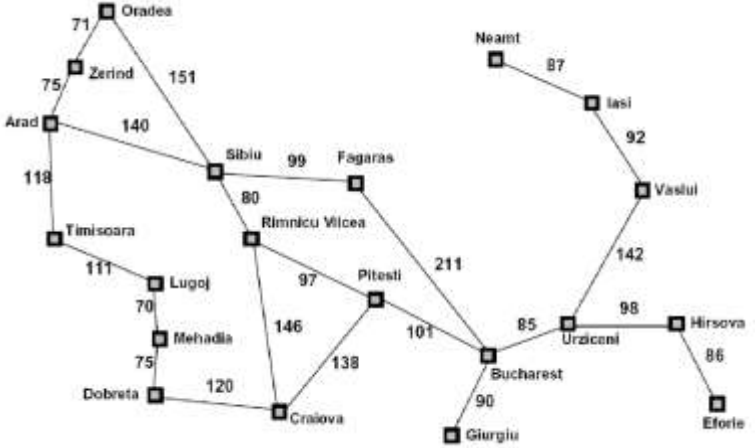


Busca A*



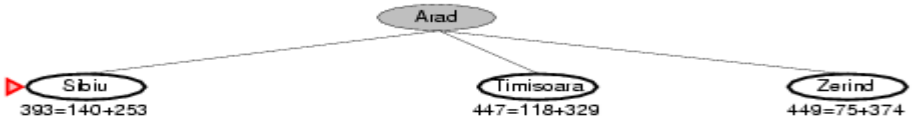
Busca A*: Exemplo (Romênia)

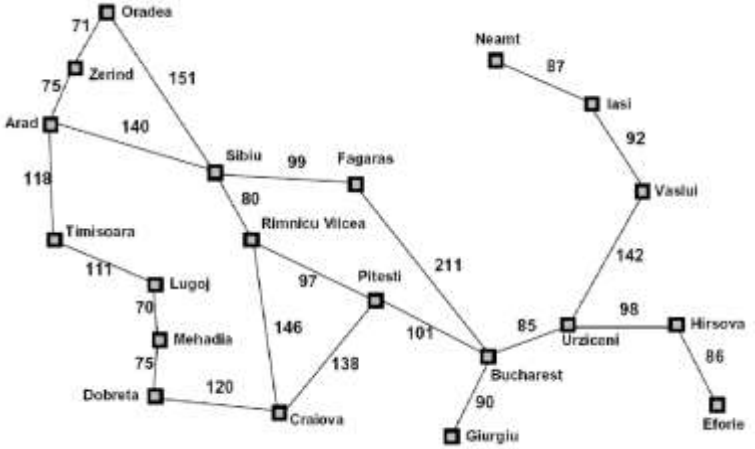




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

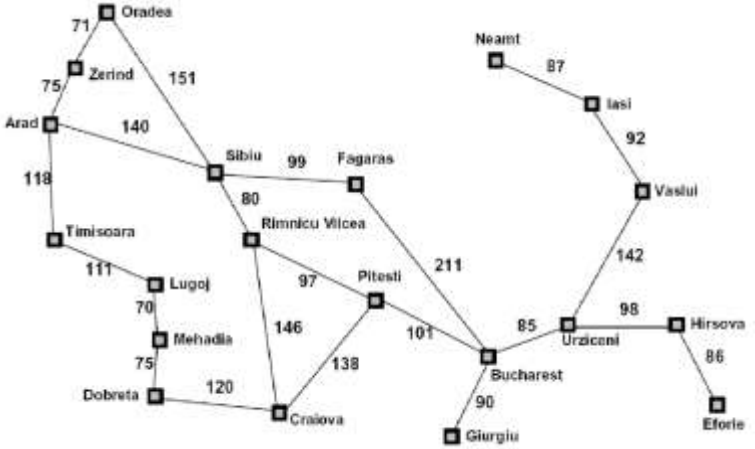
Busca A*: Exemplo (Romênia)





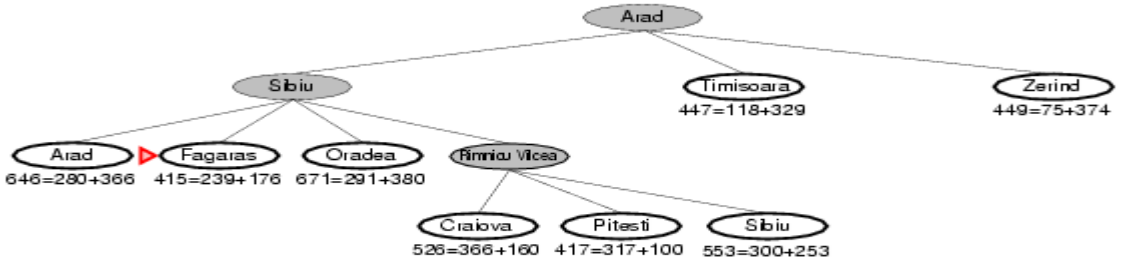
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

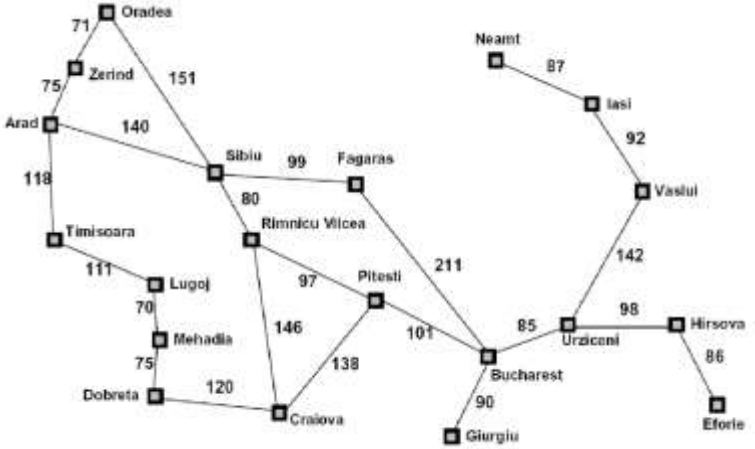
Busca A*: Exemplo (Romênia)



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

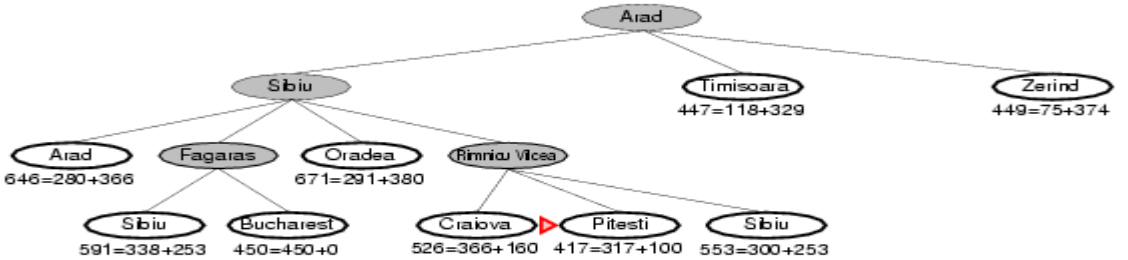
Busca A*: Exemplo (Romênia)

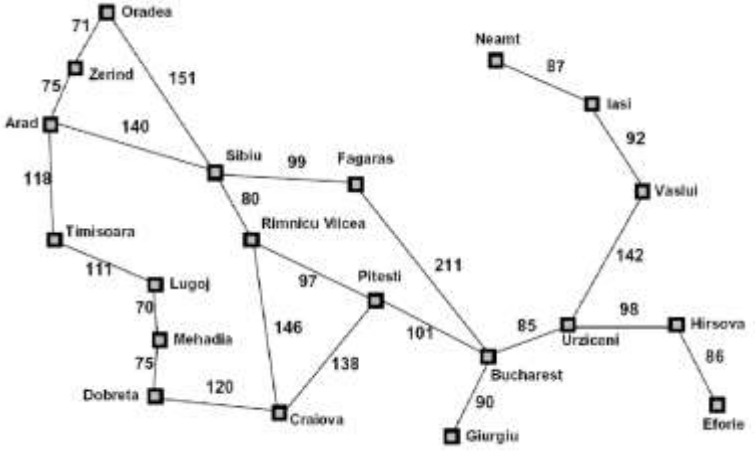




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

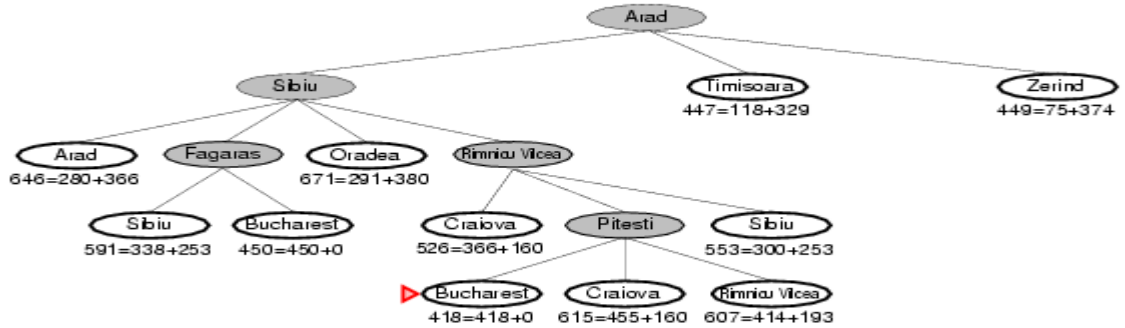
Busca A*: Exemplo (Romênia)

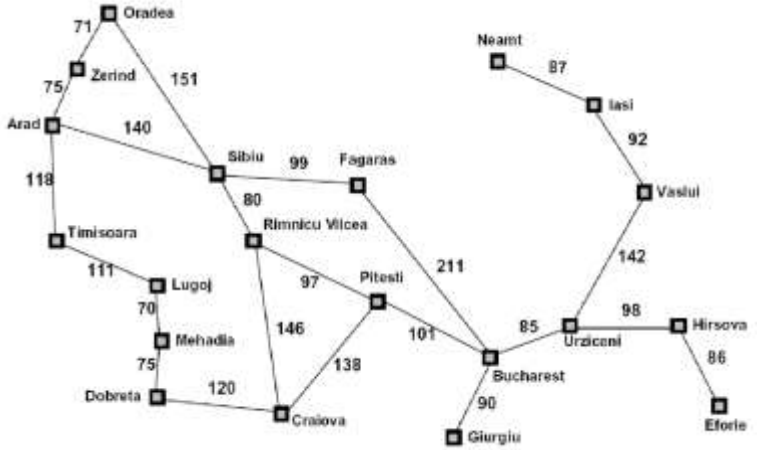




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Busca A*: Exemplo (Romênia)

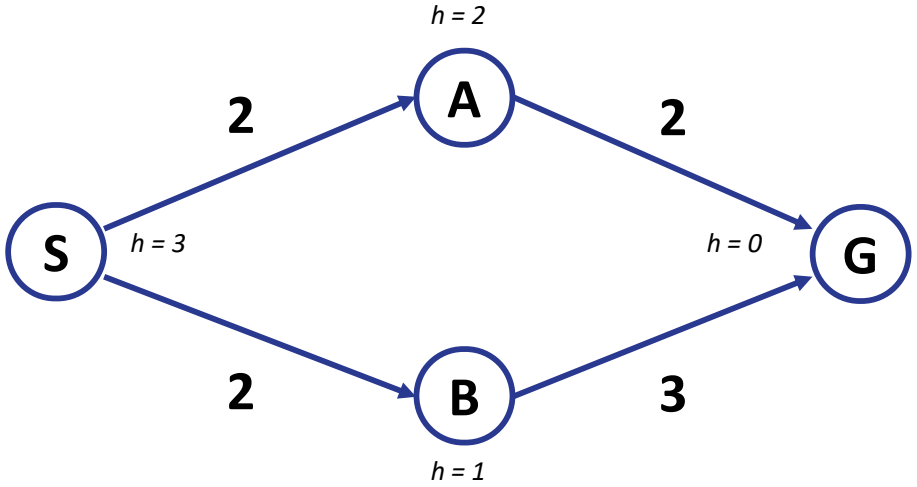




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Quando a busca A* deve terminar?

- Deve terminar quando um nó objetivo é adicionado na fila?



- **Não**! Deve terminar apenas quando um objetivo é retirado da fila.

Tree Search versus *Graph Search*

Tree Search vs Graph Search

- Duas formas de percorrer o espaço de estados de um problema de busca.
  - *Tree search*
  - *Graph search*
- São também duas formas possíveis de implementar quaisquer das estratégias de busca que estudamos até aqui.
- **Importante**: ambas geram uma árvore de busca!

Tree Search (pseudocódigo)

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Graph Search

- A busca em grafo (*graph search*) é uma extensão da busca em árvore (*tree search*) para detectar estados repetidos durante a busca.
- Ideia da busca em grafo: nunca expandir um mesmo estado mais do que uma vez.

Graph Search

- Implementação:
  - Busca em árvore + conjunto de estados já expandidos (coleção “closed” no pseudocódigo)
  - Expande a árvore de busca nó por nó, mas…
    - Antes de expandir um nó, se assegura de que seu estado ainda não foi expandido antes
    - Se o estado já foi visto, nó não é expandido (i.e., é ignorado); se o estado é novo, adiciona esse estado à coleção “closed“
  - Importante: a coleção “closed” deve ser um “set”, e não um “list”

Graph Search (pseudocódigo)

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Tree Search vs Graph Search

Tree search

```
open <- []
next <- start
while next isn't goal {
  open += successors of next
  next <- select from open
  remove next from open
}
return next
```

Graph search

```
open <- []
closed <- []
next <- start
while next isn't goal {
  closed += next
  open += successors of next, which are not in closed
  next <- select from open
  remove next from open
}
return next
```

Fonte: https://stackoverflow.com/questions/10680180/what-is-the-difference-between-graph-search-and-tree-search

Tree Search vs Graph Search

```
def depthFirstSearch(problem):
    """

    node = getStartNode(problem)
    frontier = util.Stack()
    frontier.push(node)

    closed = set()

    while not frontier.isEmpty():
        node = frontier.pop()

        if node['STATE'] in closed:
            continue

        closed.add(node['STATE'])

        if problem.isGoalState(node['STATE']):
            return getActionSequence(node)

        for sucessor in problem.expand(node['STATE']):
            child_node = getChildNode(sucessor, node)
            frontier.push(child_node)

    return []
```

Tree Search vs Graph Search

```
def depthFirstSearch(problem):
    """

    node = getStartNode(problem)
    frontier = util.Stack()
    frontier.push(node)

    closed = set()

    while not frontier.isEmpty():
        node = frontier.pop()

        if node['STATE'] in closed:
            continue

        closed.add(node['STATE'])

        if problem.isGoalState(node['STATE']):
            return getActionSequence(node)

        for sucessor in problem.expand(node['STATE']):
            child_node = getChildNode(sucessor, node)
            frontier.push(child_node)

    return []
```

Tree Search vs Graph Search

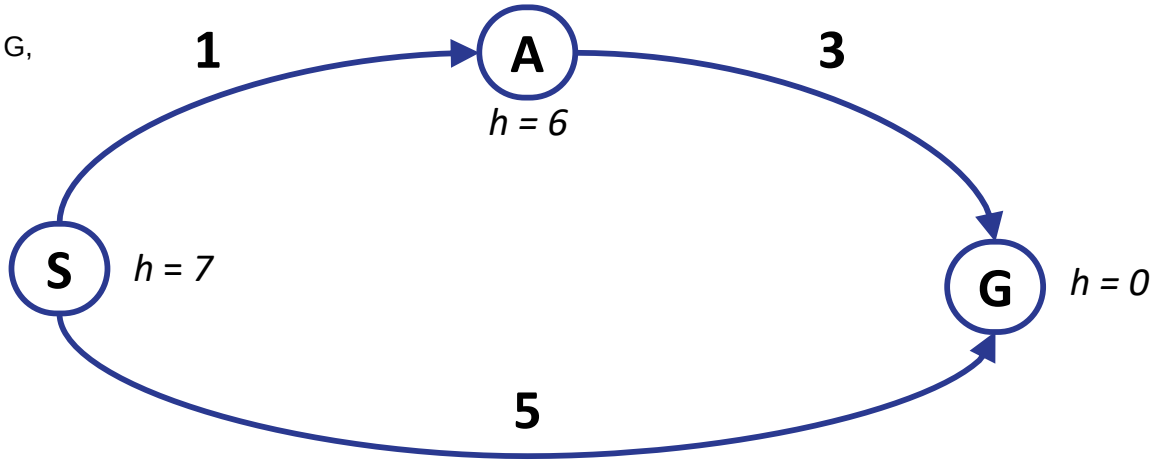
- Cuidado com a nomenclatura!
  - Tanto *graph search* quanto *tree search* produzem uma árvore de busca (*search tree*).
  - O espaço de estados sempre é representado como um grafo (embora possa não ter ciclos, caso em que ele é uma árvore).

Fonte: https://ai.stackexchange.com/questions/6426/what-is-the-difference-between-tree-search-and-graph-search/9554

A*: otimalidade

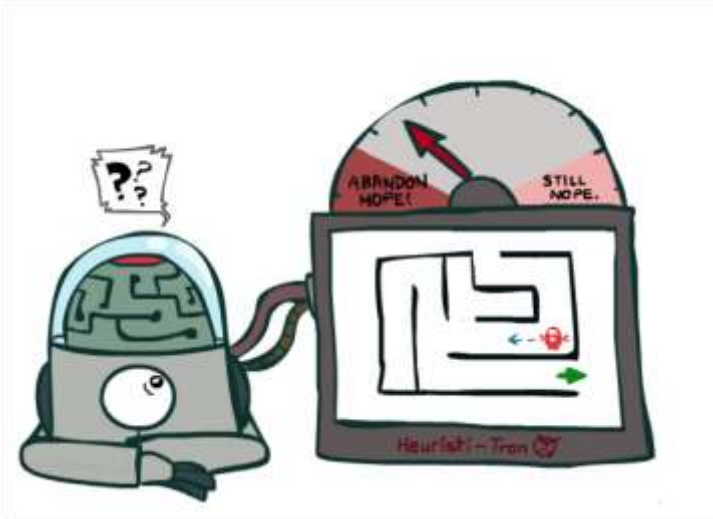
A busca A* é ótima?

$BORDA_1$ = { (S, 0+7) }
$BORDA_2$ = { (S□ A, 1+6), (S□ G, 5+0) }

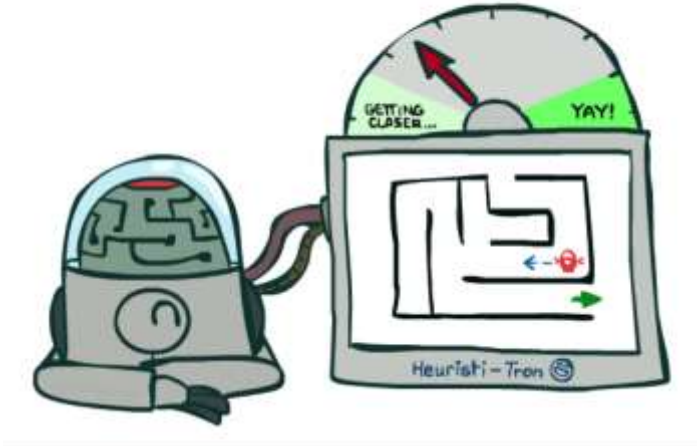


- O que deu errado nesse caso?
  - h(A) **>** custo efetivo para chegar ao objetivo, a partir de A
- Em geral: h(n) deve sempre produzir estimativas que sejam menores do que os custos efetivos!

Heurísticas admissíveis



Heurísticas inadmissíveis (i.e., pessimistas) comprometem (i.e., não garantem) a otimalidade porque “prendem” bons planos na borda



Heurísticas admissíveis (i.e., otimistas) desaceleram planos ruins, e nunca superestimam os custos verdadeiros.

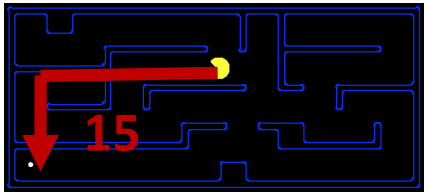
Heurísticas admissíveis: definição

- (*Definição*) Uma heurística $h(n)$ é *admissível* (i.e., otimista) se:

$$0 \leq h(n) \leq h^*(n)$$

onde $h^*(n)$ é o custo efetivo para o objetivo mais próximo

- Exemplos:

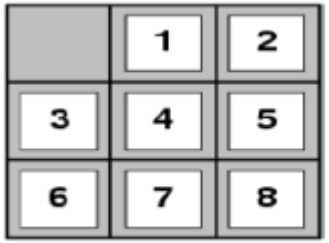


- Projetar heurísticas admissíveis é a parte mais difícil de usar o algoritmo A* em situações práticas.

Heurísticas admissíveis: exemplo

- Para o quebra-cabeça de 8 peças, considere duas heurísticas:
  - $h_1(n)$ = número de peças fora da posição
  - $h_2(n)$ = distância “Manhattan” total (para cada peça calcular a distância em “quadras” até a sua posição)

Start State



Goal State

$h_1(S)$ = ?

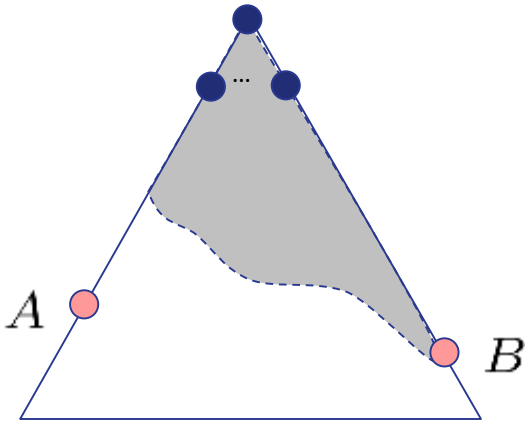
8

$h_2(S)$ = ?

3+1+2+2+2+3+3+2 = 18

A* com TREE-SEARCH

- **Teorema**: *Se h(n) é admissível, então A* usando TREE-SEARCH é ótima.*
- Considere que:
  - **A** e **B** são nós objetivos;
  - **A** corresponde a um plano ótimo;
  - **B** corresponde a um plano subótimo;
  - **h** é admissível.



- Proposição (suficiente para provar otimalidade):
  - **A** será expandido (i.e., irá sair da borda) antes de **B**

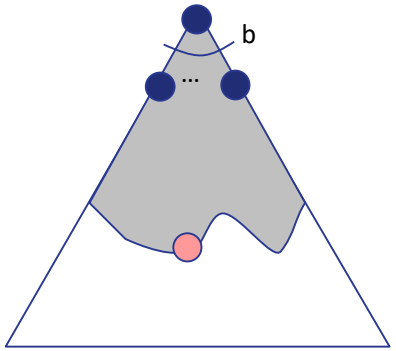
Propriedades do A*

Busca A*: propriedades

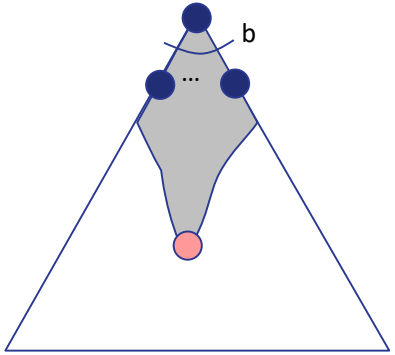
- Completa?
  - Sim, a não ser que exista uma quantidade infinita de nós com f ≤ *f(G)*
- Tempo?
  - Exponencial no pior caso
- Espaço?
  - Exponencial também (mantém todos os nós na memória)
- Ótima?
  - Sim. (TREE-SEARCH, com h admissível; GRAPH-SEARCH, com h consistente)
- Teorema: A* é otimamente eficiente
  - Nenhum outro algoritmo de busca ótimo tem garantia de expandir menos nós do que A*. Isso porque qualquer algoritmo que não expande todos os nós com f(n) < C* corre o risco de omitir uma solução ótima.

Propriedades do A*

UCS

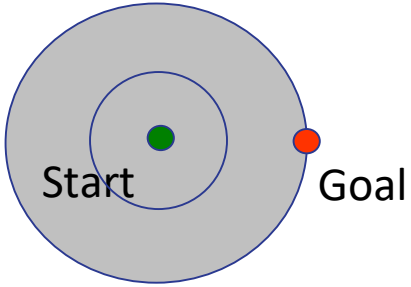


A*

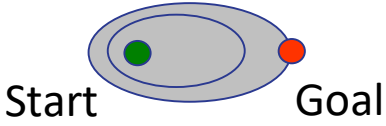


UCS vs A*

- UCS expande igualmente em todas as direções (do grafo de espaço de estados).



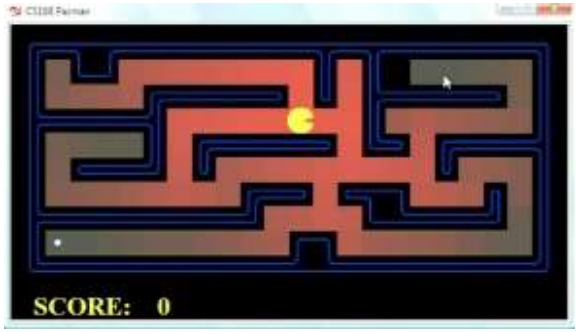
- A* expande principalmente em direção ao objetivo.



[Demo: contours UCS / greedy / A* empty (L3D1)]
[Demo: contours A* pacman small maze (L3D5)]

Comparação

Guloso



Uniform Cost

A*

Aplicações do A*

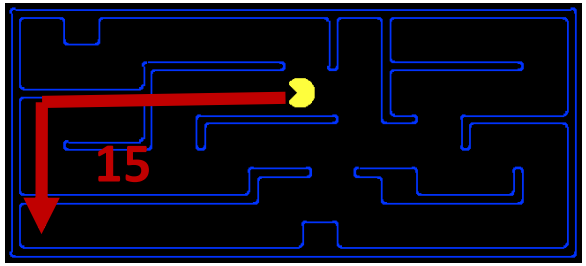
Aplicações do A*

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

Criação de Heurísticas

Criação de Heurísticas Admissíveis

- A maior parte do esforço envolvido na solução de problemas de busca complexos está na criação de heurísticas admissíveis.
- Frequentemente, heurísticas admissíveis são soluções para problemas relaxados, nos quais novas ações estão disponíveis.



- Heurísticas não admissíveis são úteis também em contextos específicos.