



EA 072 Inteligência Artificial em Aplicações Industriais

4-Estruturas e Estratégias de Busca

Introdução

- Algoritmos de busca:
 - não informados (*depth-first*, *breadth-first*)
 - usam somente definição do problema (*problem*)
 - informados (*best-first*)
 - usam conhecimento sobre o domínio além de *problem*
 - conhecimento na forma de heurísticas
- Aplicações
 - sistemas baseados em conhecimento
 - sequenciamento de produção
 - busca internet

■ Desempenho e complexidade de algoritmos de busca

– desempenho

- completeza: garantia de encontrar uma solução, se existir
- otimalidade: estratégia busca encontra solução ótima
- complexidade temporal: tempo para achar uma solução
- complexidade espacial: quantidade de memória para a busca

– complexidade

- fator de ramificação (b): no. máximo de sucessores de um nó
- profundidade (d): profundidade do nó meta mais raso
- comprimento trajetória (m): maior entre todas trajetórias

Algoritmos de busca não Informados

- Características

- busca cega
- utiliza somente informação contida em *problem*
- definidos pela ordem em que os nós são expandidos
- necessário eliminar ciclos (organizar soluções em uma árvore)
- detectar estados redundantes
- complexidade

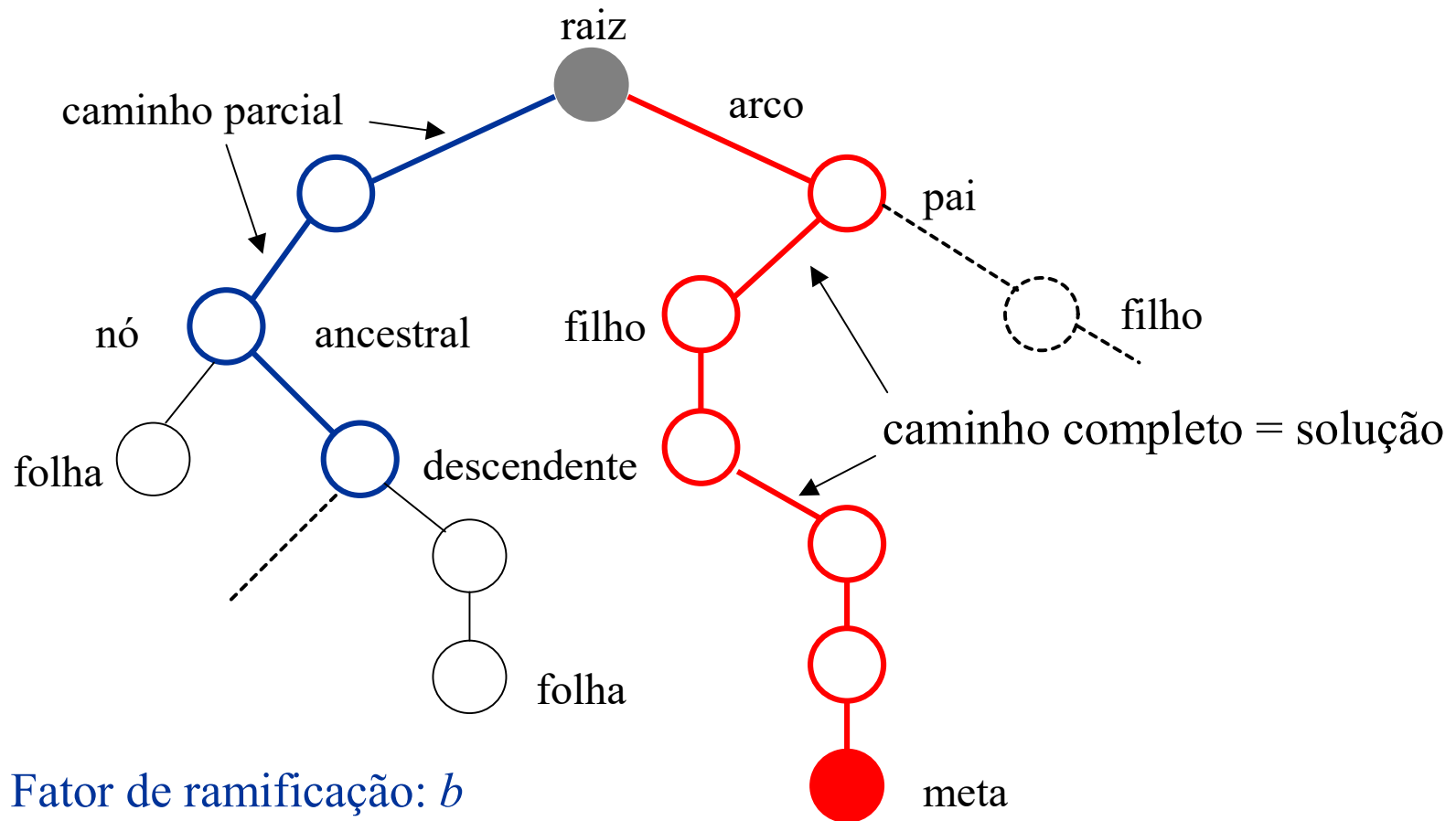
Componentes de um problema (*problem*)

- estado inicial
- ações disponíveis ao agente
- modelo de transição (efeito ações)
- teste de meta
- função de avaliação de um caminho (custo)

Ordenação dos nós para expansão

- FIFO : pops o elemento mais antigo da fila
- LIFO: pops o elemento mais novo da fila (stack)
- PRIORITY: pops elemento com maior prioridade

Árvores de busca



Fator de ramificação: b
Profundidade: d
Total caminhos: b^d

Nó em algoritmos de busca

Estrutura de dados com os seguintes componentes

Estado (STATE): estado, elemento de um espaço de estado

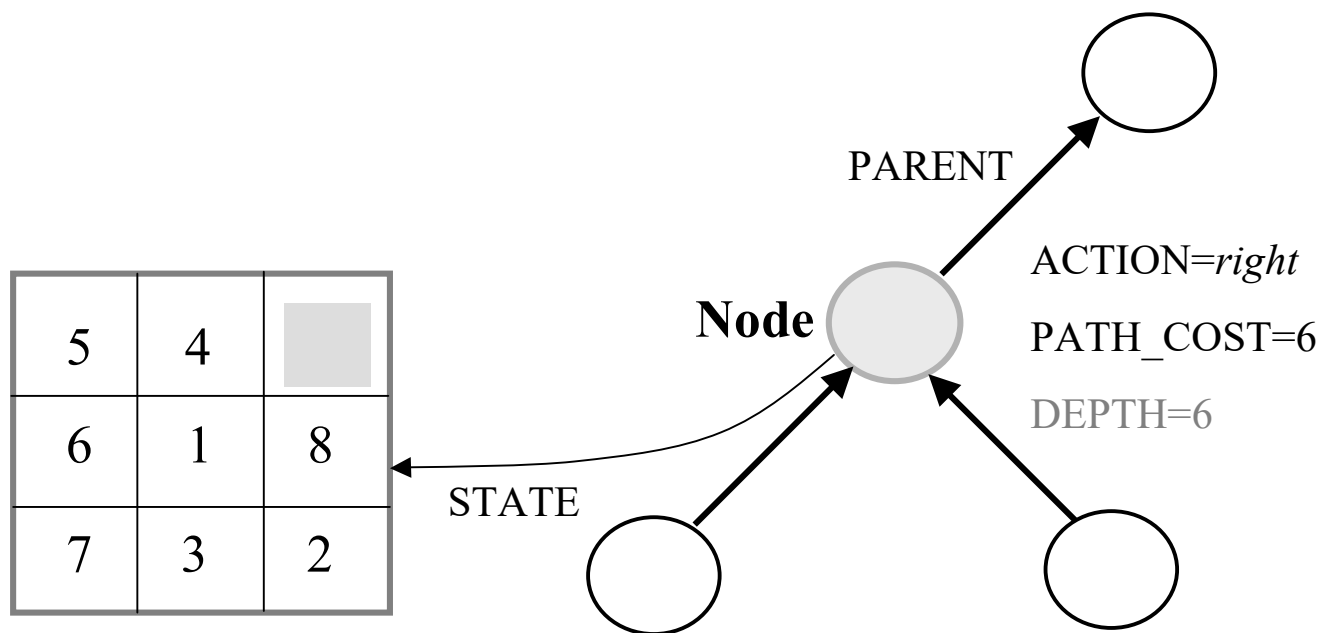
Nó pai (PARENT): pai de um nó filho

Ação (ACTION): ação que, aplicada a um nó pai, gera seus filhos

Custo (PATH-COST): $g(n)$ valor do caminho da raiz até o nó n

Profundidade (DEPTH): número de arcos no caminho da raiz até nó n

Nó em algoritmos de busca



Infraestrutura para algoritmos de busca

Estrutura com componentes

n .STATE: estado correspondente ao nó n

n .PARENT: nó da árvore que gerou nó n

n .ACTION: ação aplicada ao pai que gerou n

n .PATH-COST: $g(n)$ custo do estado inicial até n

p .STEP-COST = $c(s, a, n)$ custo de um passo para problema p

p .RESULT = RESULT(s, a) modelo (de transição) (sucessor) de p

Geração de filho de um nó

function CHILD_NODE (*problem, parent, action*) **returns** a node

return a node with

STATE = *problem.RESULT(parent.STATE, action)*

PARENT = *parent*

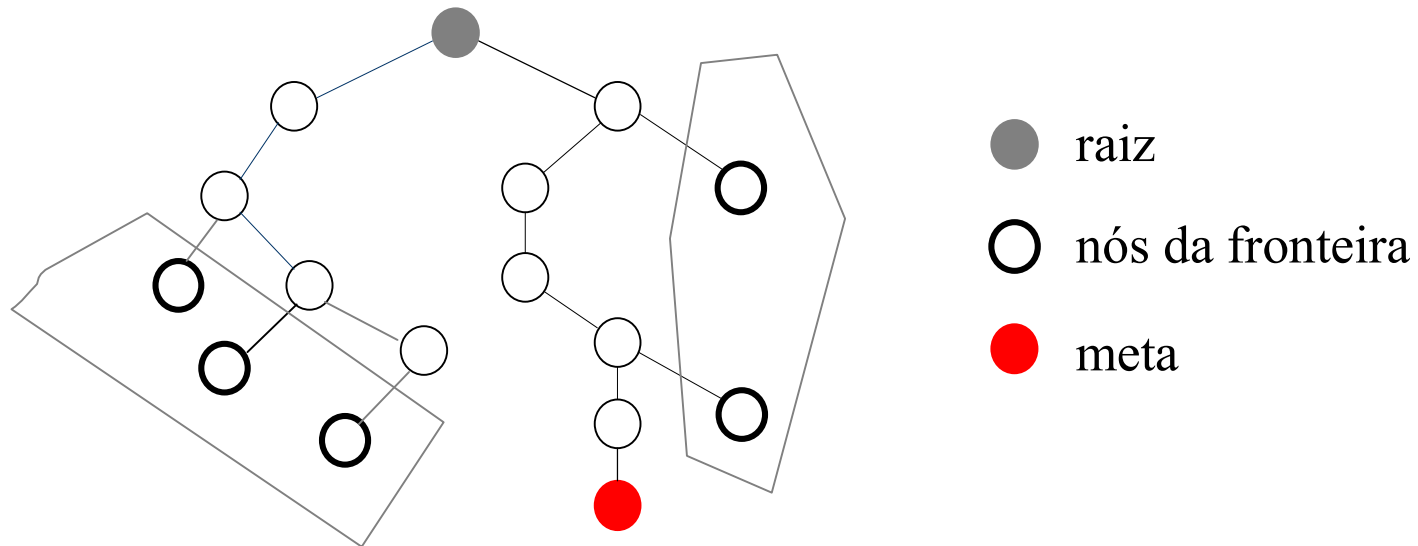
ACTION = *action*

PATH-COST = *parent.PATH-COST*

+ *problem.STEP-COST(parent.STATE, action)*

Fronteira (*frontier*)

Estrutura dados é uma fila (*queue*)



Operações com filas

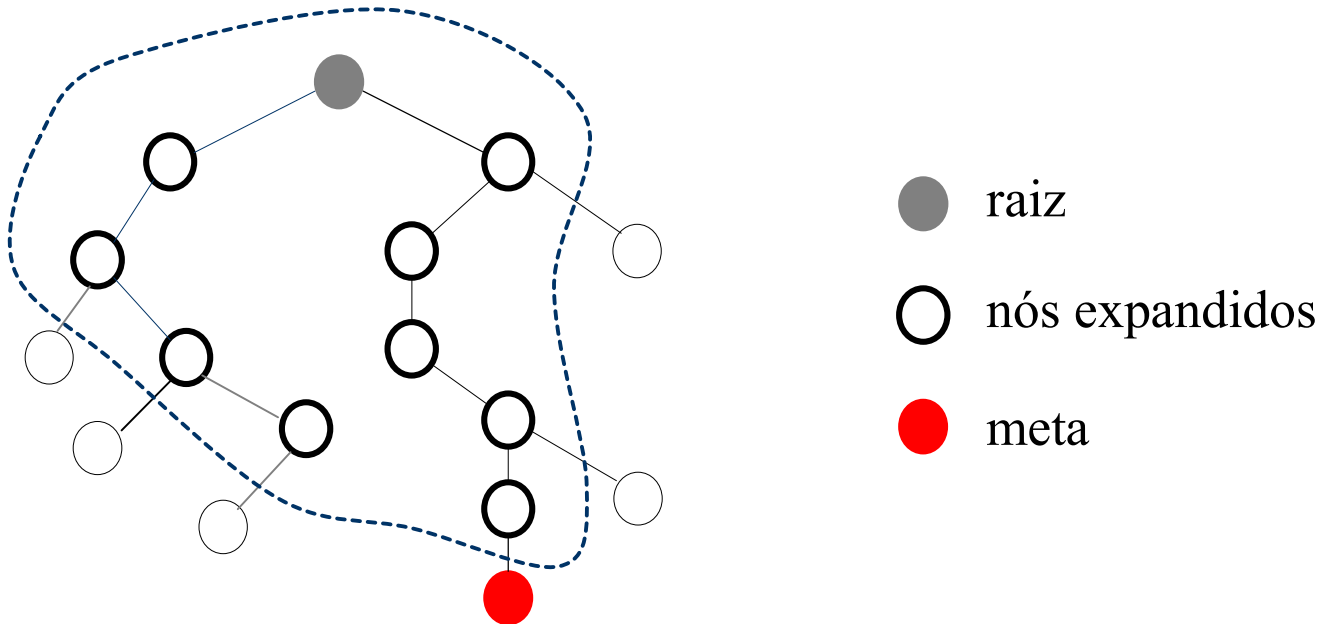
EMPTY?(*queue*): retorna *true* somente se fila é vazia

POP(*queue*): remove e retorna primeiro elemento da fila

INSERT(*element*, *queue*): insere elemento e retorna fila resultante

SOLUTION(*n*): retorna a sequência de ações de *n* até a raiz

Conjunto de nós expandidos (*explored set*)



Conjunto nós expandidos = *hash table*

Propósito: verificar estados repetidos

Igualdade de conjuntos: {Bucharest, Vaslui} = {Vaslui, Bucharest}

Algoritmo de busca em grafos

function GRAPH_SEARCH (*problem*) **returns** a solution or failure

frontier ← a node with STATE = *problem*.INITIAL-STATE
explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses a node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

 add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

Busca em largura (*breadth-first search*)

function BREADTH_FIRST_SEARCH (*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

if *problem*.GOAL-TEST (*node*.STATE) **then return** SOLUTION(*node*)

frontier ← a FIFO queue with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the shallowest node in *frontier* */

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

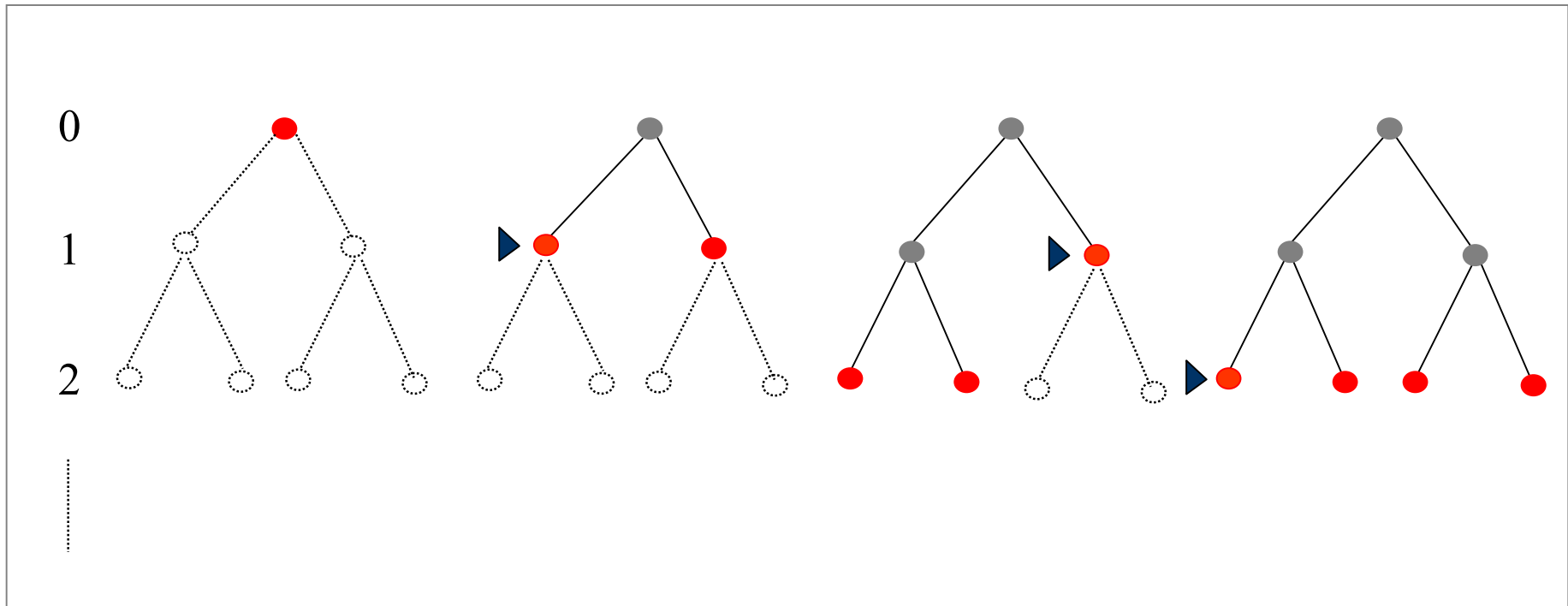
child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)

frontier ← INSERT (*child*, *frontier*)

Exemplo:



FIFO_QUEUE (First In–First Out) = ENQUEUE_AT_END

Complexidade busca em largura

- completo (se b é finito)
- não necessariamente ótimo
 - a menos que custo trajetória seja função não decrescente da profundidade
- tempo e memória (profundidade da meta = d)

$$b + b^2 + \dots + b^d = O(b^d)$$

Profundidade	Nós	Tempo	Memória
4	11.110	11 ms	10.6 MB
8	10^8	2 m	103 GB
12	10^{12}	13 dias	1 PB(10^{15})
16	10^{16}	350 anos	10 EB (10^{18})

$b = 10$, 1.000.000 nós/s, 1000 bytes/nó

Busca uniforme (*uniform search*)

- expande nó com menor $g(n)$ (*priority queue* ordenada por g)
 - nó no caminho com menor custo
- teste meta aplicado quando um nó é selecionado para expansão
 - ao invés de quando o nó é gerado
 - porque ? : nó pode estar em um caminho sub-ótimo
- teste para verificar se existe nó na fronteira com melhor custo
- expande nós desnecessariamente se custo dos passos são iguais
- ótimo com qualquer STEP-COST, $c(s, a, s')$

Algoritmo de busca uniforme

function UNIFORM_COST_SEARCH (*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a priority queue ordered by PATH-COST with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the lowest-cost node in *frontier* */

if *problem*.GOAL-TEST (*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

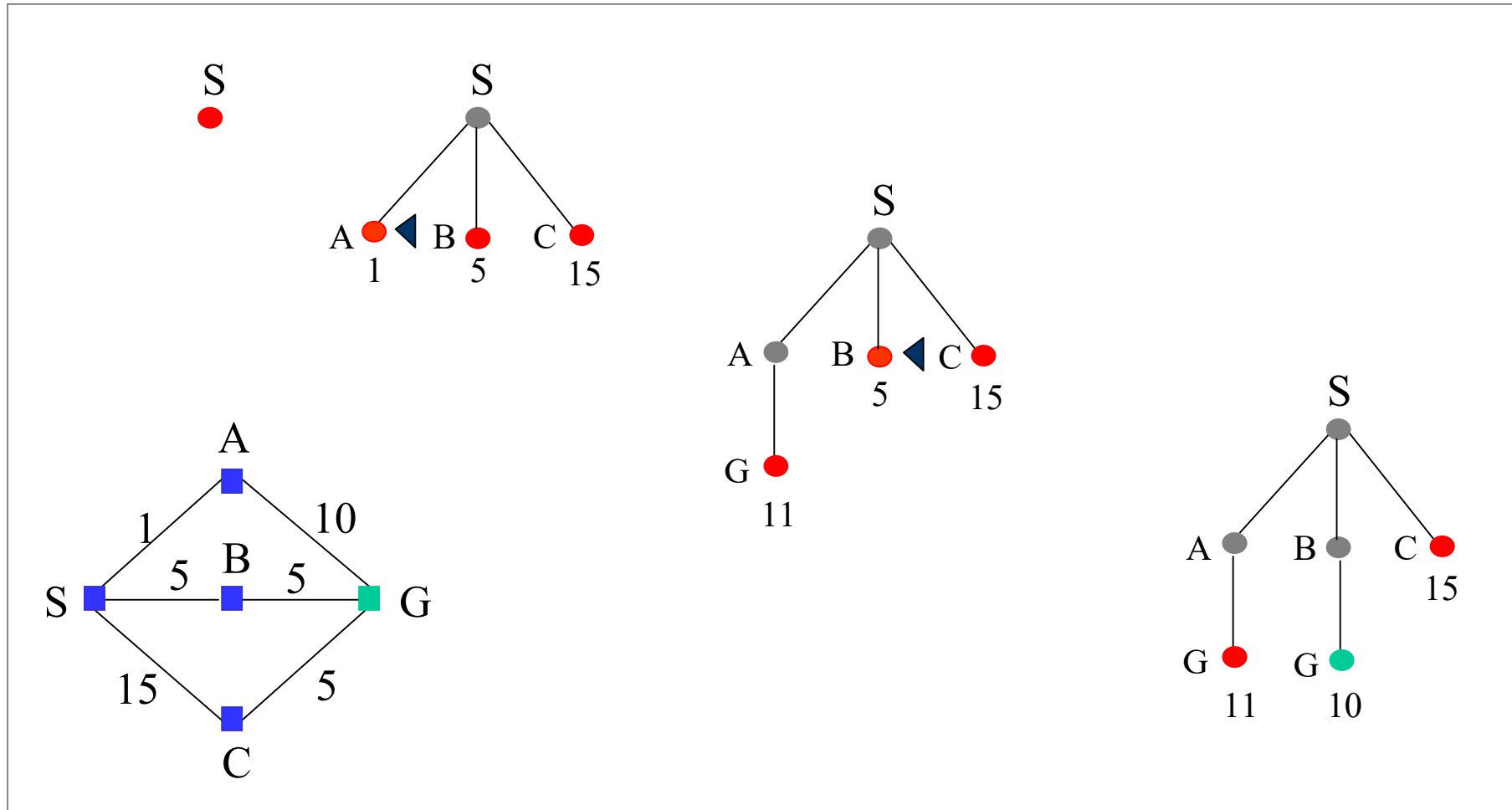
if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher PATH-COST **then**

replace that *frontier* node with *child*

Exemplo



ENQUEUE_BEST_AT_FRONT

Complexidade busca uniforme

- completo (se cada passo tem custo $\varepsilon > 0$)
- ótimo em geral
- C^* custo da solução ótima
- tempo e memória

$$O(b^{1+\lfloor C^*/\varepsilon \rfloor}) \geq O(b^d)$$

$$\text{se custos passos iguais} \rightarrow b^{1+\lfloor C^*/\varepsilon \rfloor} = b^{d+1}$$

Busca em profundidade (*depth-first search*)

function DEPTH_FIRST__SEARCH (*problem*) **returns** a solution or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a LIFO queue with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the deepest node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

 add *node*.STATE to *explored*

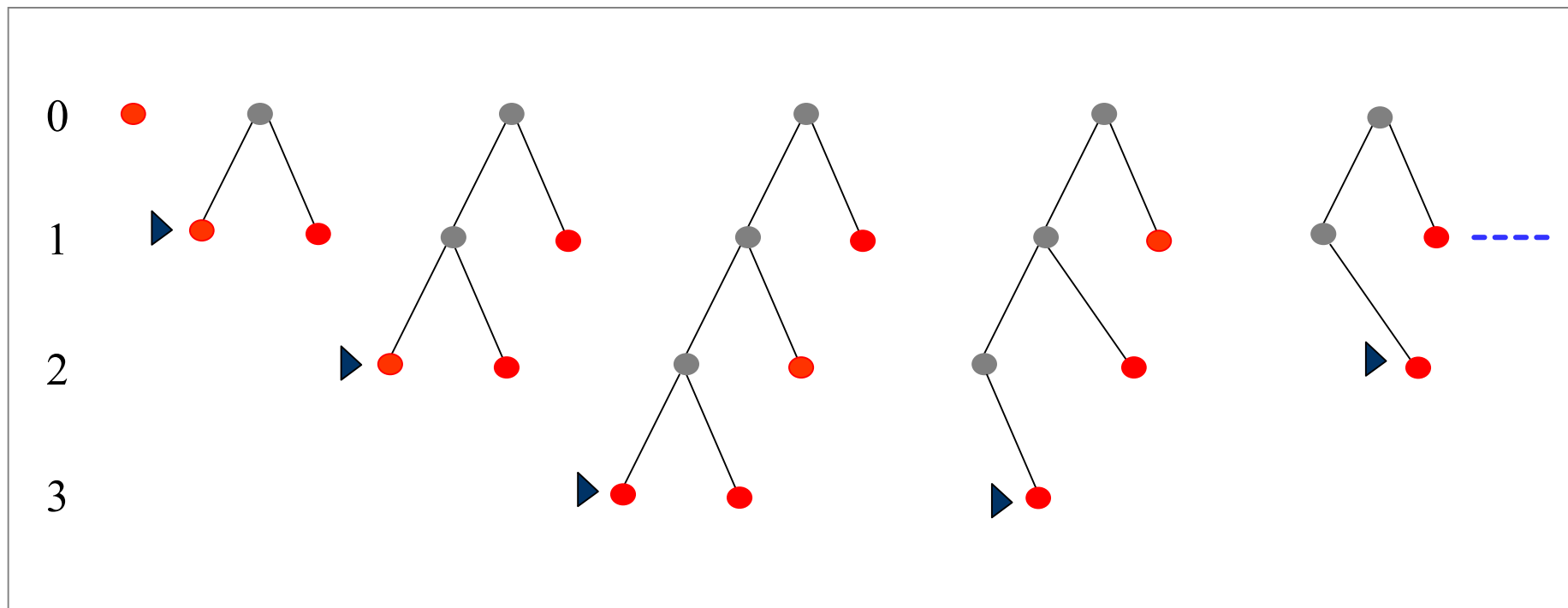
for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

Exemplo:



LIFO (Last In–First Out) = ENQUEUE_AT_FRONT

Exemplo: assume nós com profundidade 3 sem sucessores

Complexidade busca em profundidade

- não é completo (árvore), completo (grafo, espaço estado finito)
 - não é ótimo em ambos casos
 - complexidade temporal
 - grafo: limitada pelo tamanho espaço de estado (que pode ser ∞)
 - árvore: $O(b^m)$, m profundidade máxima de um nó
 - complexidade espacial
 - grafo: limitada pelo tamanho espaço de estado (que pode ser ∞)
 - árvore: memória modesta: bm nós
- exemplo: meta sem sucessores, $d = 16$
- $b = 10$, 1.000.000 nós/s, 1000 bytes/nó
- 156 KB (10 EB na busca em largura)
- fator: 7 trilhões menos memória !

Busca profundidade limitada (*depth-limited search*)

- ideia: usar busca em problemas com caminhos infinitos
- não é completo se $\ell < d$ (d : profundidade nó meta mais raso)
- não é ótimo se $\ell > d$
- complexidade temporal: $O(b^\ell)$
- complexidade espacial: $O(b\ell)$
- busca profundidade = busca profundidade limitada com $\ell = \infty$
- conhecimento do domínio da aplicação ajuda determinar limite

Algoritmo de busca em profundidade limitada

function DEPTH_LIMITED_SEARCH (*problem*, *limit*) **returns** a solution, or failure/cutoff

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a LIFO queue with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return failure**

node ← POP (*frontier*) /* chooses the deepest node in frontier */

if *problem*.GOAL-TEST(*node*.STATE) **then return SOLUTION**(*node*)

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

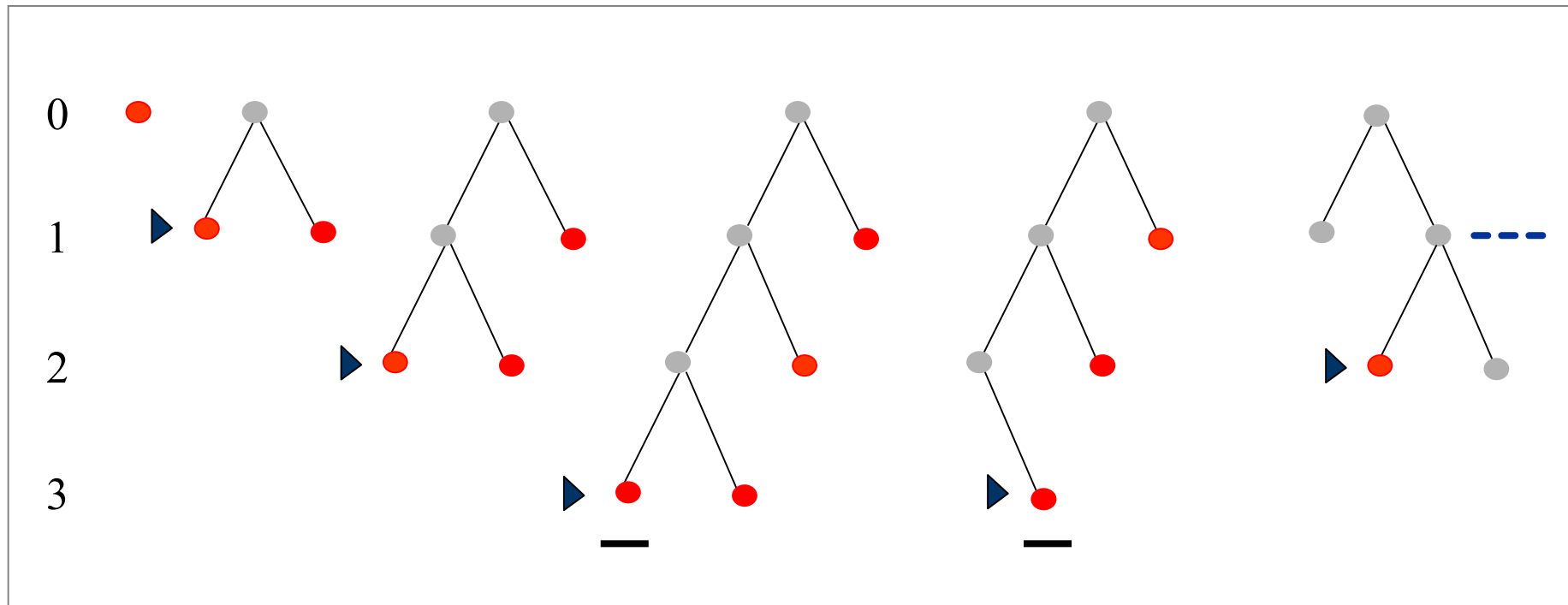
child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

if DEPTH(*node*) = *limit* **then return cutoff**

frontier ← INSERT (*child*, *frontier*)

Exemplo:



Exemplo supõe profundidade máxima = 3 ($\ell = 3$)

Busca profundidade progressiva (*iterative-deepening search*)

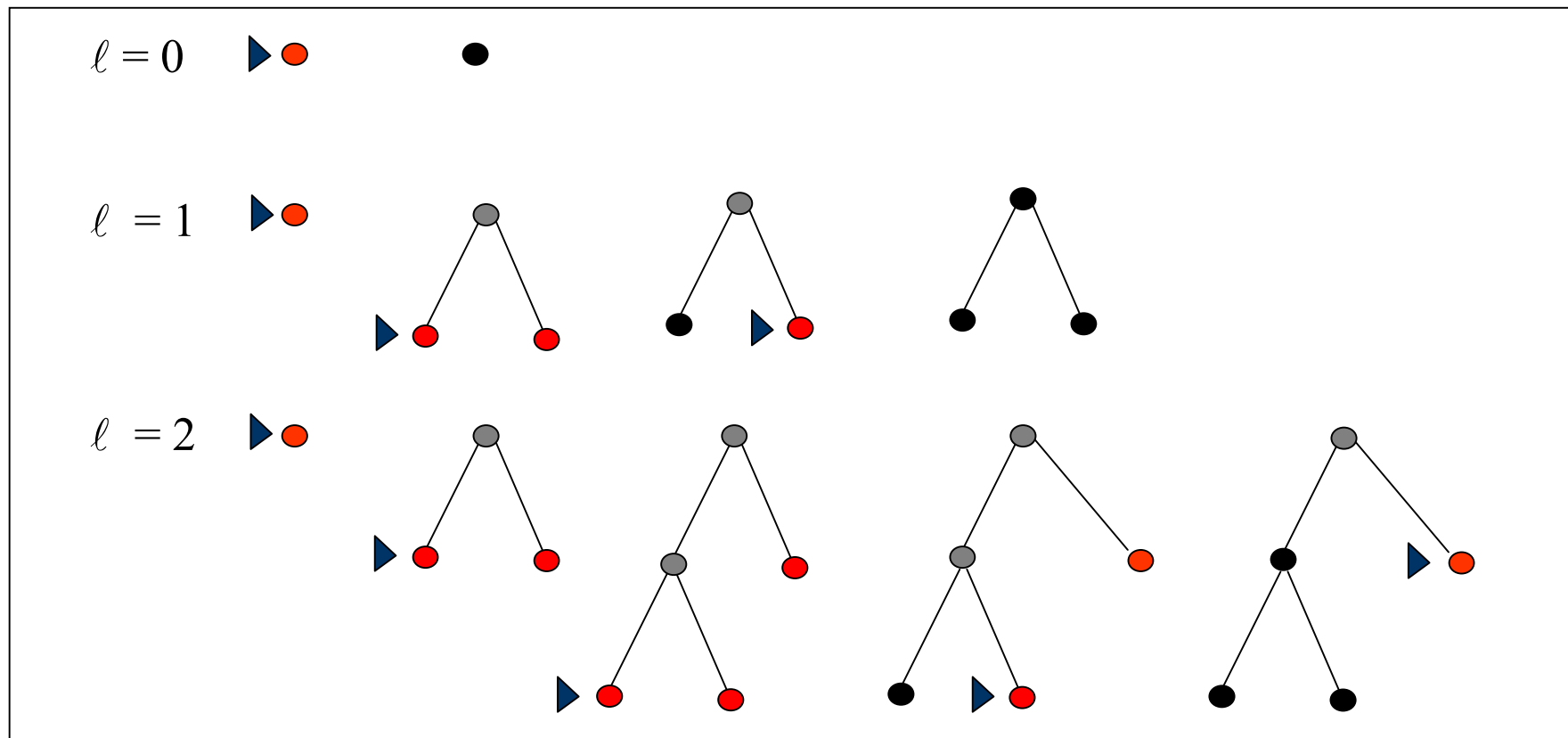
- ideia: aumentar limite de profundidade gradualmente até encontrar meta
- combina busca profundidade com busca em largura
- completo se b é finito
- ótimo se custo caminho não diminui com a profundidade
- complexidade espacial: $O(bd)$

function ITERATIVE_DEEPENING_SEARCH (*problem*) **returns** a solution, or failure

for *depth* \leftarrow 0 **to** ∞ **do**

result \leftarrow DEPTH_LIMITED_SEARCH (*problem*, *depth*)

if *result* \neq cutoff **then return** *result*



$$N(\text{IDS}) = (d)b + (d-1)b^2 + \dots + b^d$$

$$N(\text{BFS}) = b + b^2 + \dots + b^d$$

$$b = 10, d = 5 \rightarrow N(\text{IDS}) = 123.450 \quad N(\text{BFS}) = 111.110$$

$N(\text{IDS})$ não é muito maior que $N(\text{BFS})$!

- **IDS:** método de escolha quando espaço busca é grande
profundidade da solução não é conhecida *a priori*

Complexidade dos algoritmos de busca (em árvore)

Critério	Tempo	Memória	Ótimo ?	Completo ?
Largura	$O(b^d)$	$O(b^d)$	sim (custos iguais)	sim ($b < \infty$)
Uniforme	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	sim	sim ($b < \infty, c \geq \varepsilon > 0$)
Profundidade	$O(b^m)$	$O(bm)$	não	não
Profundidade limitada	$O(b^\ell)$	$O(b\ell)$	não	não
Profundidade progressiva	$O(b^d)$	$O(bd)$	sim (custos iguais)	sim (b finito)
Bidirecional	$O(b^{d/2})$	$O(b^{d/2})$	sim (custos iguais)	sim (b finito)

b : fator de ramificação

d : profundidade da solução

m : profundidade máxima da árvore de busca

ℓ : limite da profundidade

C^* : custo da solução ótima

ε : menor custo de uma ação

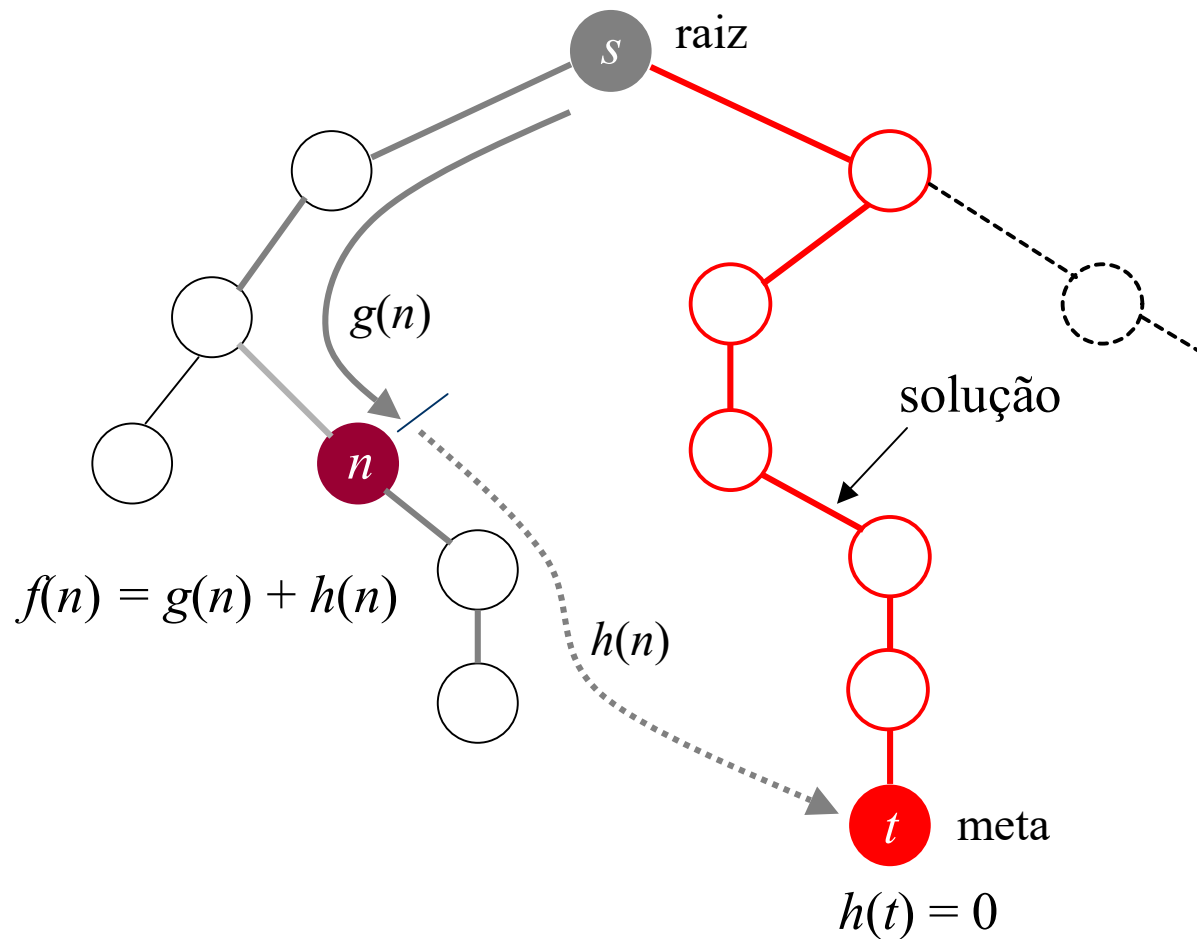
Algoritmos de busca informados

■ Características

- conhecimento domínio + *problem*
- função avaliação $f(n)$
- função heurística $h(n)$
- conhecimento na forma de heurísticas
- algoritmos do tipo *best-first*

■ Algoritmos do tipo *best-first*

- busca uniforme: $f(n) = g(n)$
- *greedy best-first*: $f(n) = h(n)$
- A* : $f(n) = g(n) + h(n)$



n : nó da árvore

$f(n)$: valor de f em n (estimativa custo mínimo através de n)

$g(n)$: custo do caminho da raiz até n

$h(n)$: *estimativa* do custo mínimo de n até a meta

function BEST_FIRST_SEARCH (*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a priority queue ordered by PATH-COST with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the **node with lowest cost** in *frontier* */

if *problem*.GOAL-TEST (*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

else if *child*.STATE is in *frontier* with **higher cost** **then**

replace that *frontier* node with *child*

Greedy best-first search

function GREEDY_BEST_FIRST_SEARCH (*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a priority queue ordered by PATH-COST with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the node with lowest $h(n)$ in *frontier* */

if *problem*.GOAL-TEST (*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

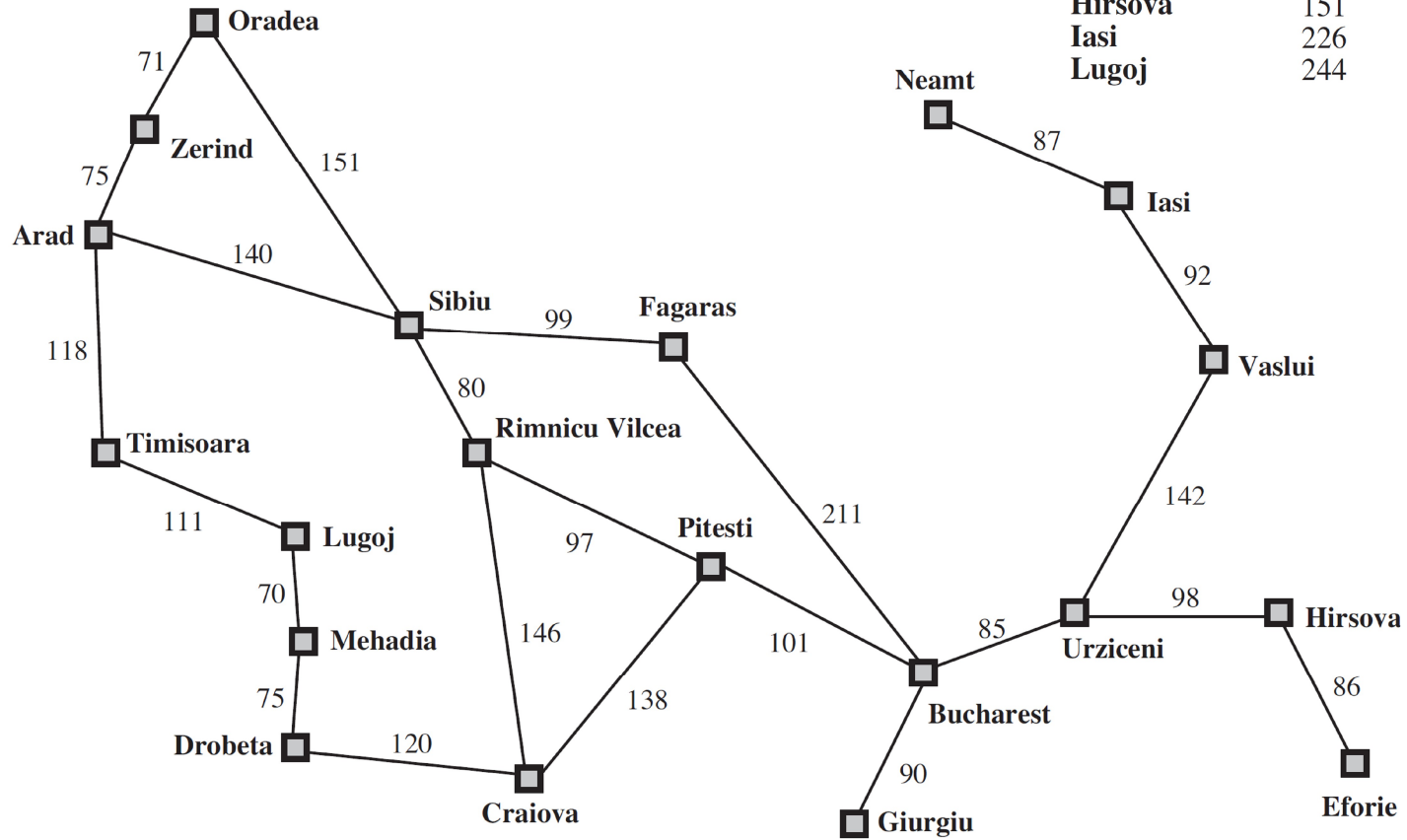
if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher cost **then**

replace that *frontier* node with *child*

Exemplo



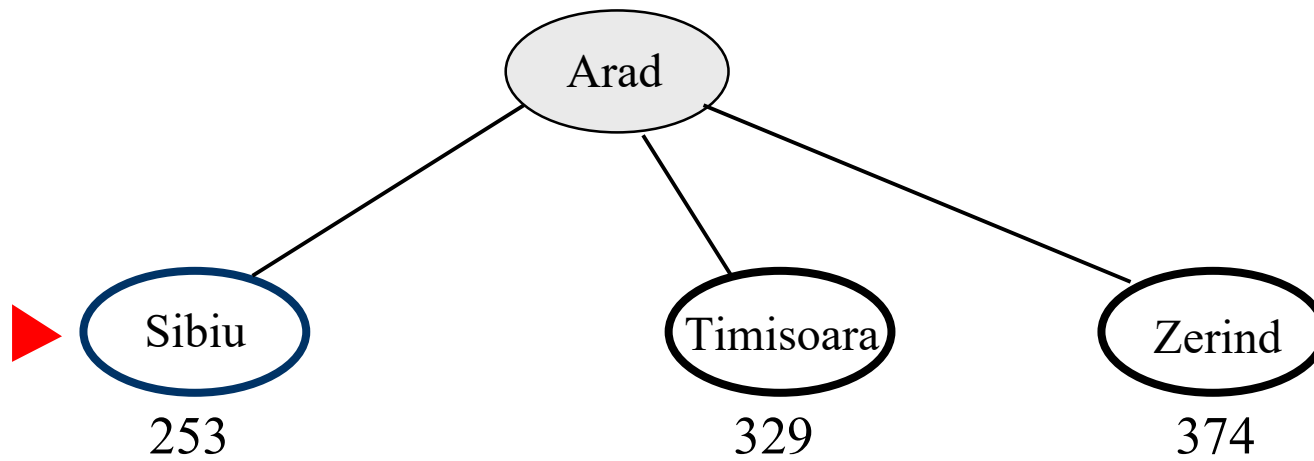
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

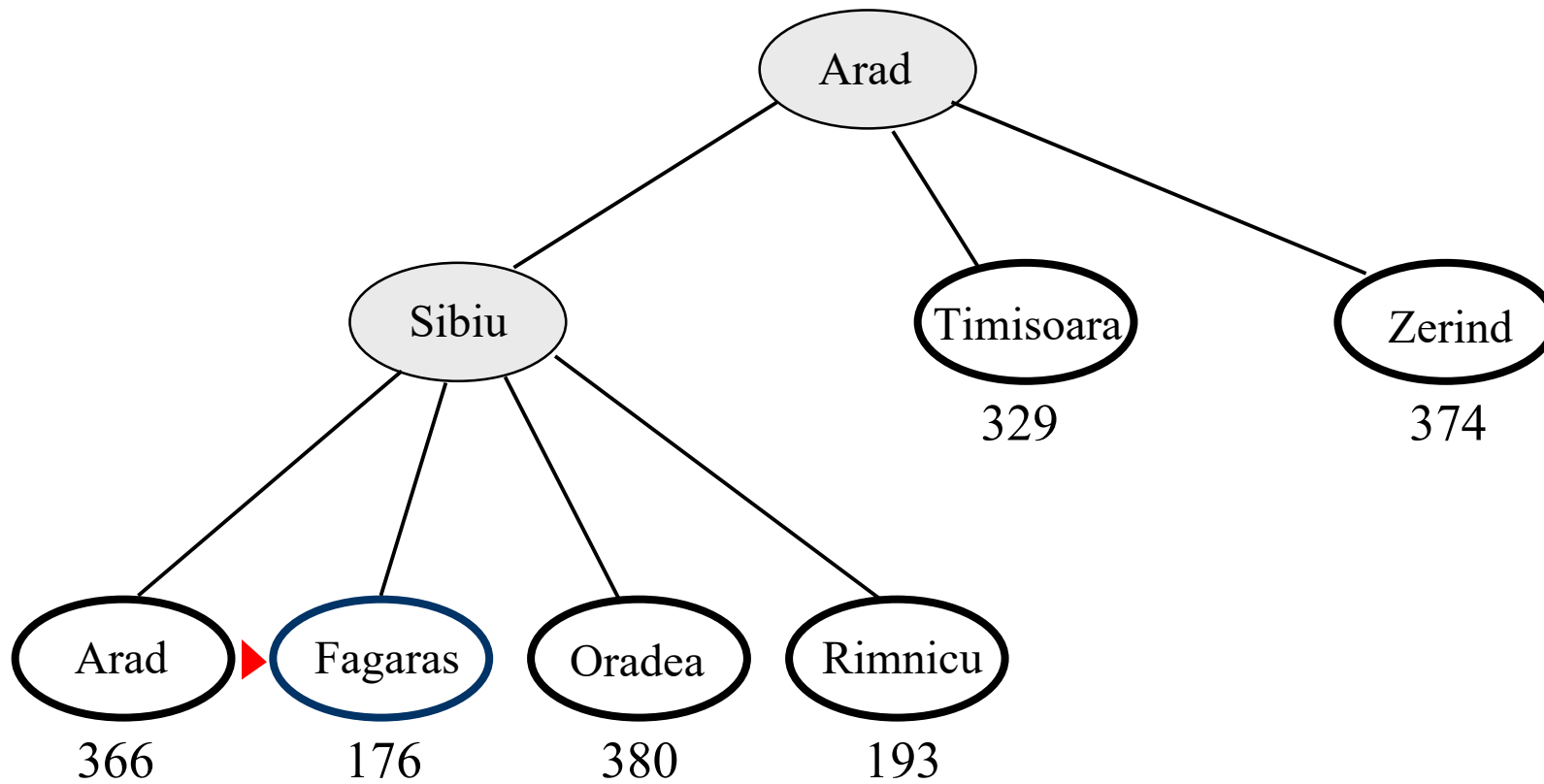
Greedy best-first search

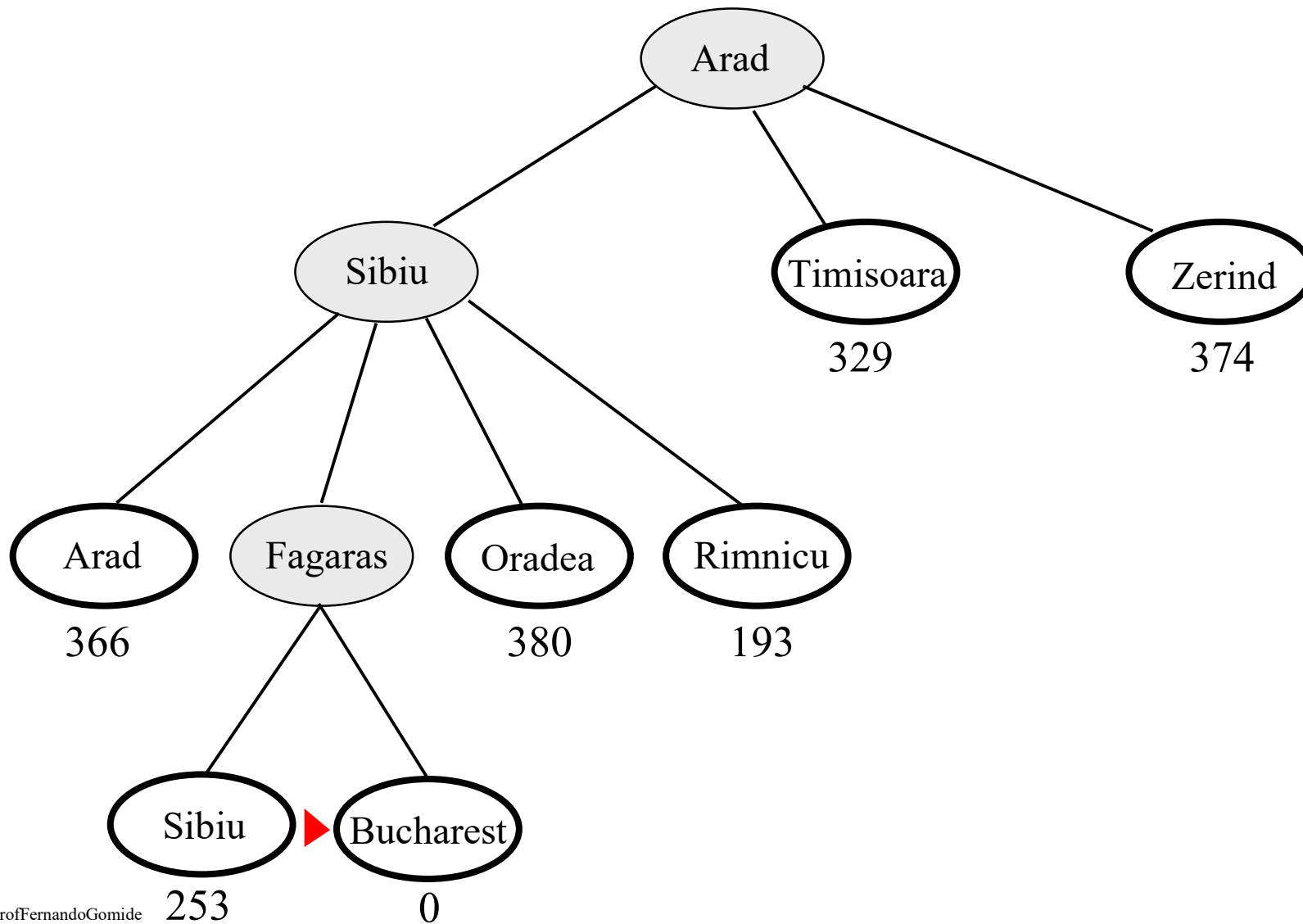
Estado inicial *In (Arad)*



366







Greedy best-first search

- baixo custo de busca
- não é ótimo
 - Arad-Sibiu-Fagaras-Bucharest = 450
 - Arad-Sibiu-Rimnicu Vilcea-Pitesti-Bucharest = 418
- versão árvore: incompleto (mesmo em espaço estado finito)
 - caminho de Iasi para Fagaras
- versão grafo: completo (em espaço estado finito)
- complexidade temporal/espacial: $O(b^m)$
- qualidade de $h(n)$ reduz complexidade

Busca A*

function A*_SEARCH (*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0

frontier ← a priority queue ordered by PATH-COST with *node* as the only element

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP (*frontier*) /* chooses the node with lowest $f(n)$ in *frontier* */

if *problem*.GOAL-TEST (*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

for each *action* in *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT (*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher PATH-COST **then**

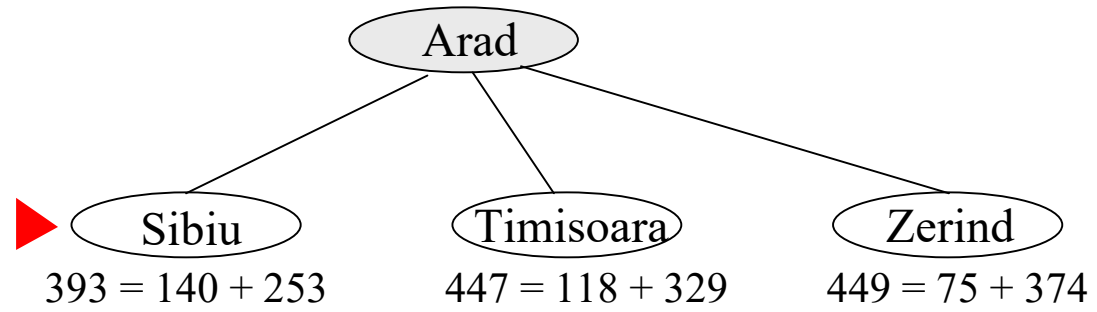
replace that *frontier* node with *child*

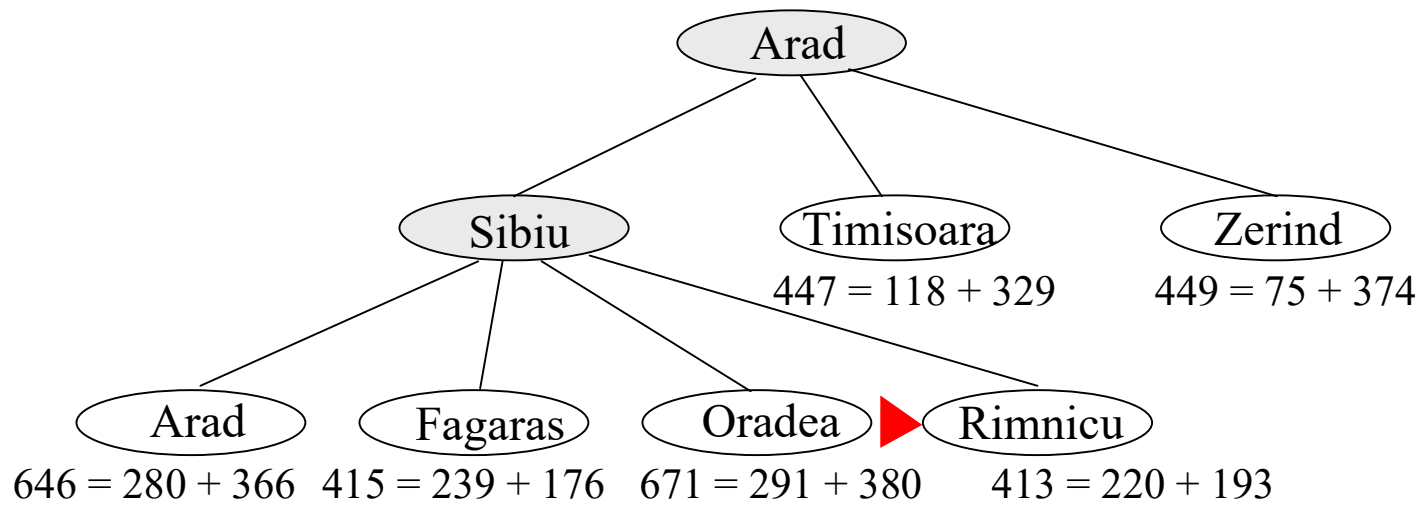
Busca A*

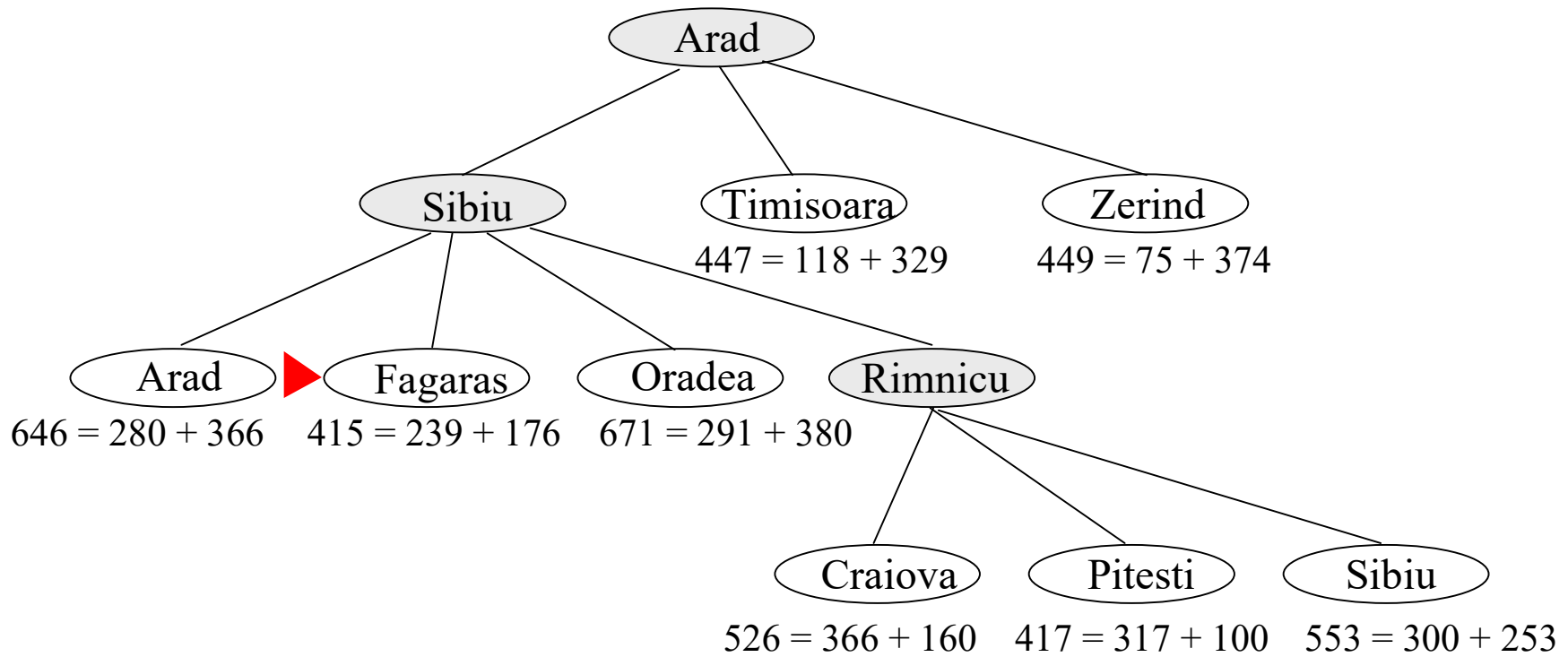
Estado inicial *In (Arad)*

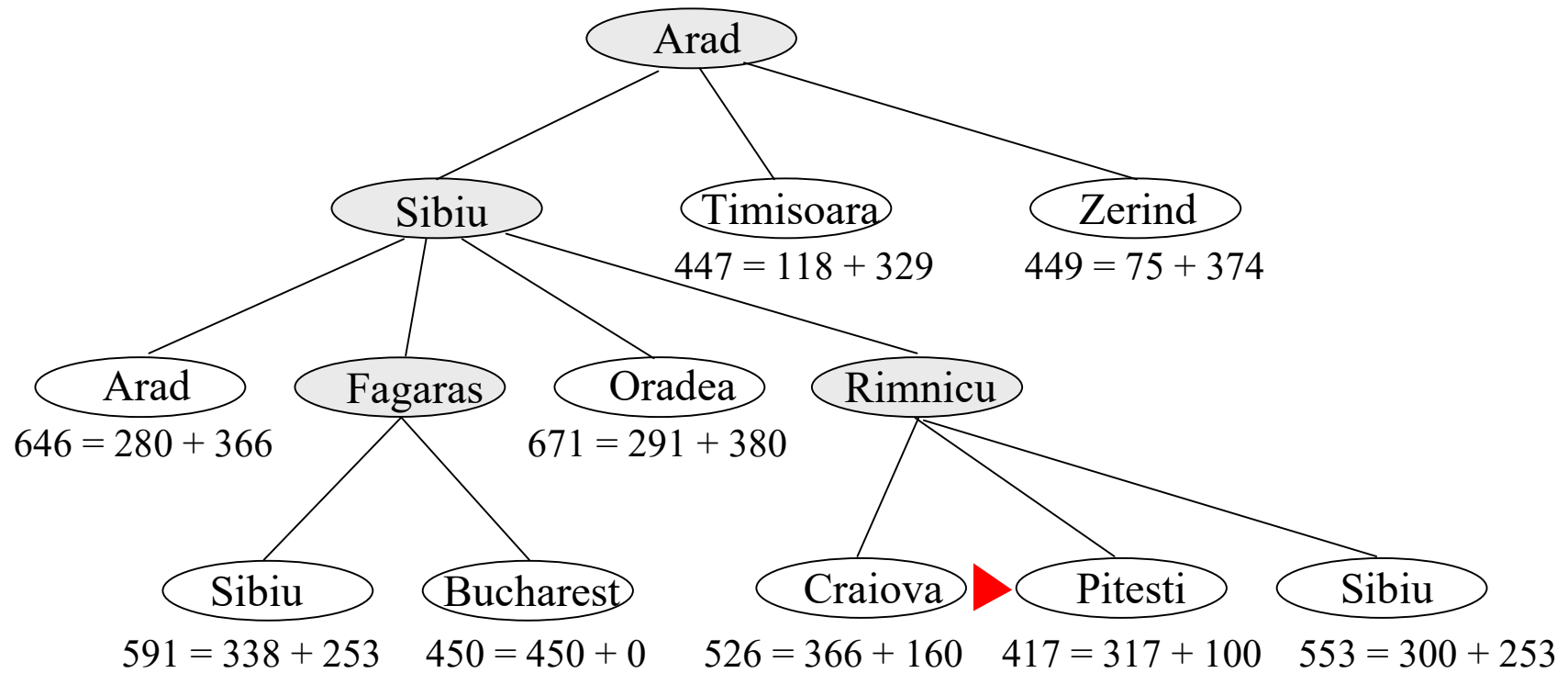


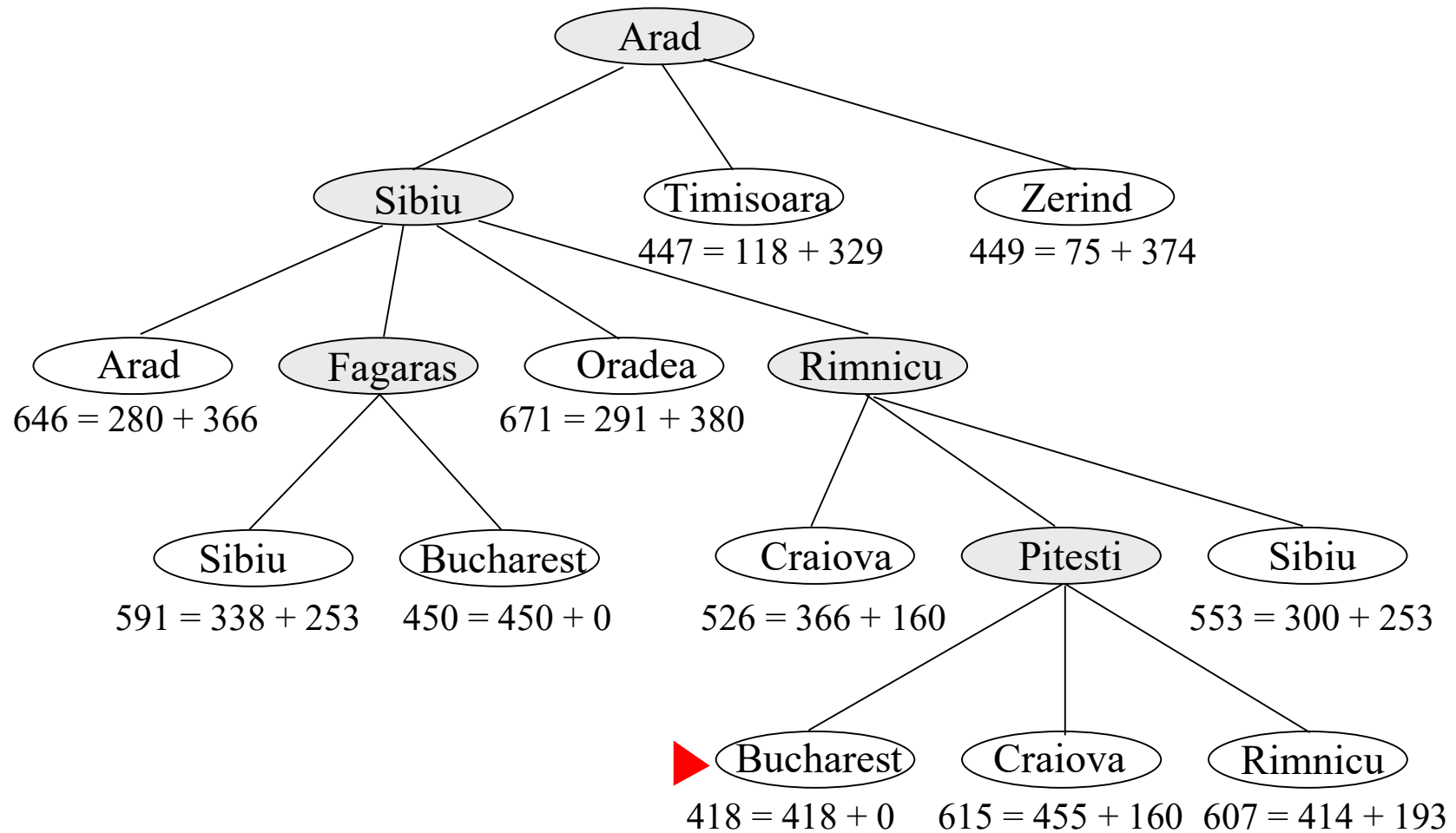
$$366 = 0 + 366$$











Propriedades do algoritmo A*

- $f(n) = g(n) + h(n)$
- $h(n)$ otimista (nunca superestima o custo de atingir a meta) $\rightarrow h$ admissível
 - exemplo: distância em linha reta (h_{SLD}) no exemplo da Romênia
- $h(n) \leq c(n, a n') + h(n') \rightarrow h$ consistente (monotônica)
- A* com TREE_SEARCH é ótimo se $h(n)$ é admissível
- A* com GRAPH_SEARCH é ótimo se $h(n)$ é consistente

Teorema: A* com TREE_SEARCH e h admissível é ótimo.

Prova:

Supor meta G_2 foi gerada e está na fila

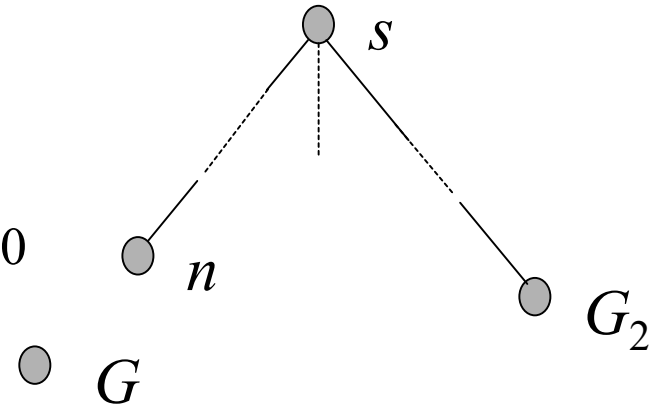
Seja n um nó não expandido no caminho ótimo

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \text{ pois } h(G_2) = 0$$

$$f(n) = g(n) + h(n) \leq C^*$$

$$f(n) \leq C^* \leq f(G_2)$$

G_2 nunca será expandido, logo A* tem que retornar solução ótima



Lema: Se $h(n)$ é consistente, então os valores de $f(n)$ ao longo de qualquer caminho não decrescem (A^* expande nós em ordem crescente dos valores de f)

Prova:

$$h(n) \leq c(n, a, n') + h(n') \quad \text{consistência}$$

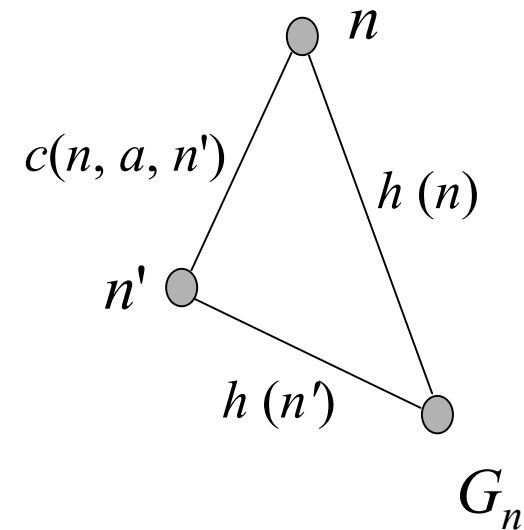
$$n' \text{ sucessor de } n \Rightarrow g(n') = g(n) + c(n, a, n')$$

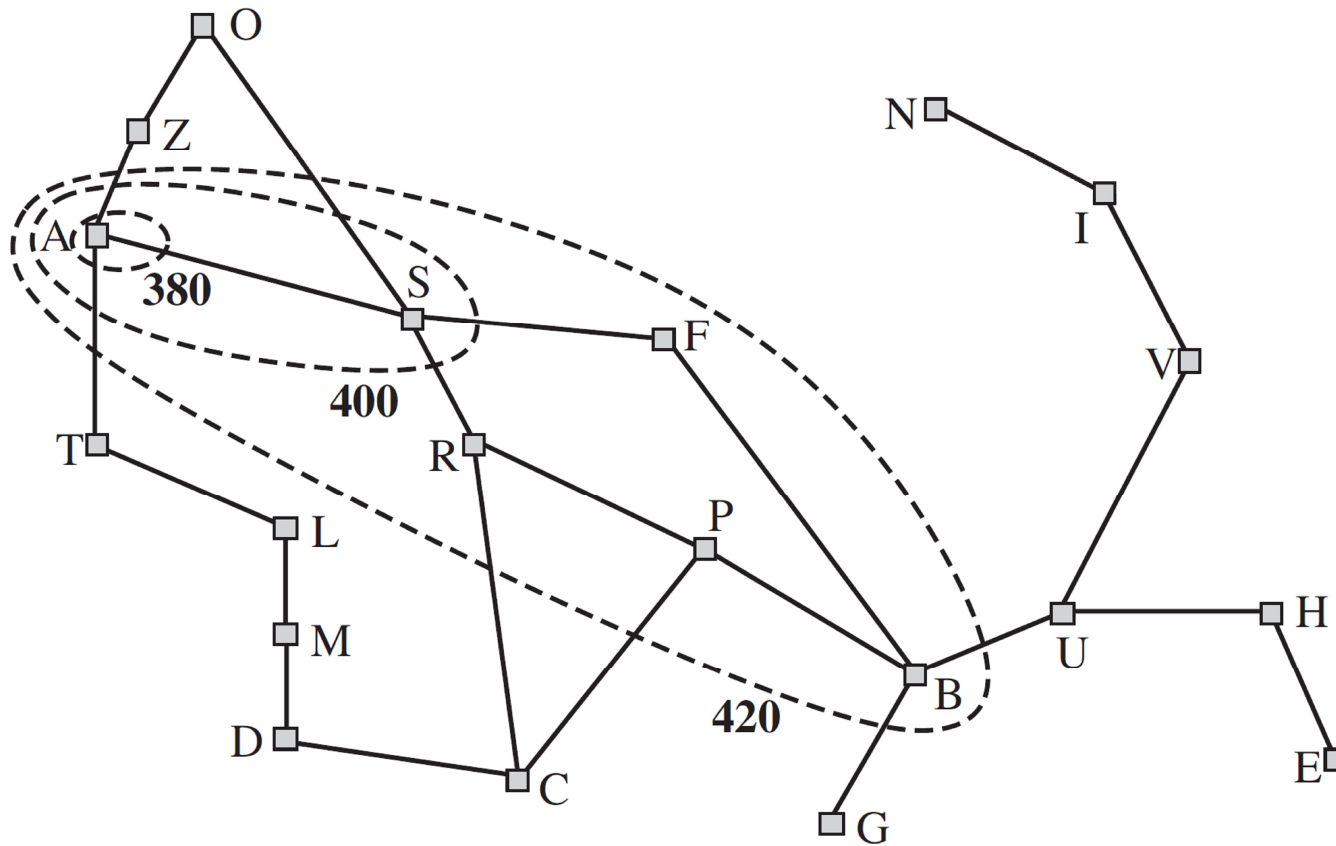
$$f(n') = g(n') + h(n')$$

$$= g(n) + c(n, a, n') + h(n')$$

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

isto é, $f(n)$ é não decrescente ao longo de qualquer caminho.





contorno i contém todos nós com $f = f_i$, $f_i < f_{i+1}$

Teorema: A* com GRAPH_SEARCH e h consistente é ótimo.

Prova:

1- h consistente $\Rightarrow f(n)$ ao longo de qualquer caminho é não decrescente (Lema)

2- sempre que A* seleciona um nó n para expansão, o caminho ótimo da raiz para o nó n já foi encontrado

Se este não fosse o caso, existiria um nó n' na fronteira (propriedade da separação) no caminho ótimo da raiz para n tal que $f(n') < f(n)$ (valor de f não diminui ao longo de qualquer caminho) e n' seria selecionado primeiro.

3- os itens 1 e 2 significam que a sequência de nós expandidos pelo A* usando GRAPH_SEARCH está em ordem não decrescente de f . Então o primeiro nó selecionado para expansão tem que ser a solução ótima pois $h(\text{meta}) = 0$ e todos nós seguintes certamente terão custo maior.

Resumo

- A* é completo, ótimo e eficiente
- Complexidade é exponencial
- Memória é o maior problema
- Para torná-lo mais eficiente: escolha apropriada da heurística
 - abstração do problema
 - relaxação
 - experimentos estatísticos
 - aprendizagem de parâmetros de funções

IDA* *Iterative deepening* A*

- ideia: aumentar limite progressivamente
- limite: custo $f\text{-cost}(g + h)$ e não a profundidade
- valor de corte: $newcutoff = \min \{f\text{-cost dos nós com } f\text{-cost} > oldcutoff\}$
- prático para custos passos unitários
- sofre dos mesmos problemas da busca uniforme

Busca heurística com limite de memória: RBFS

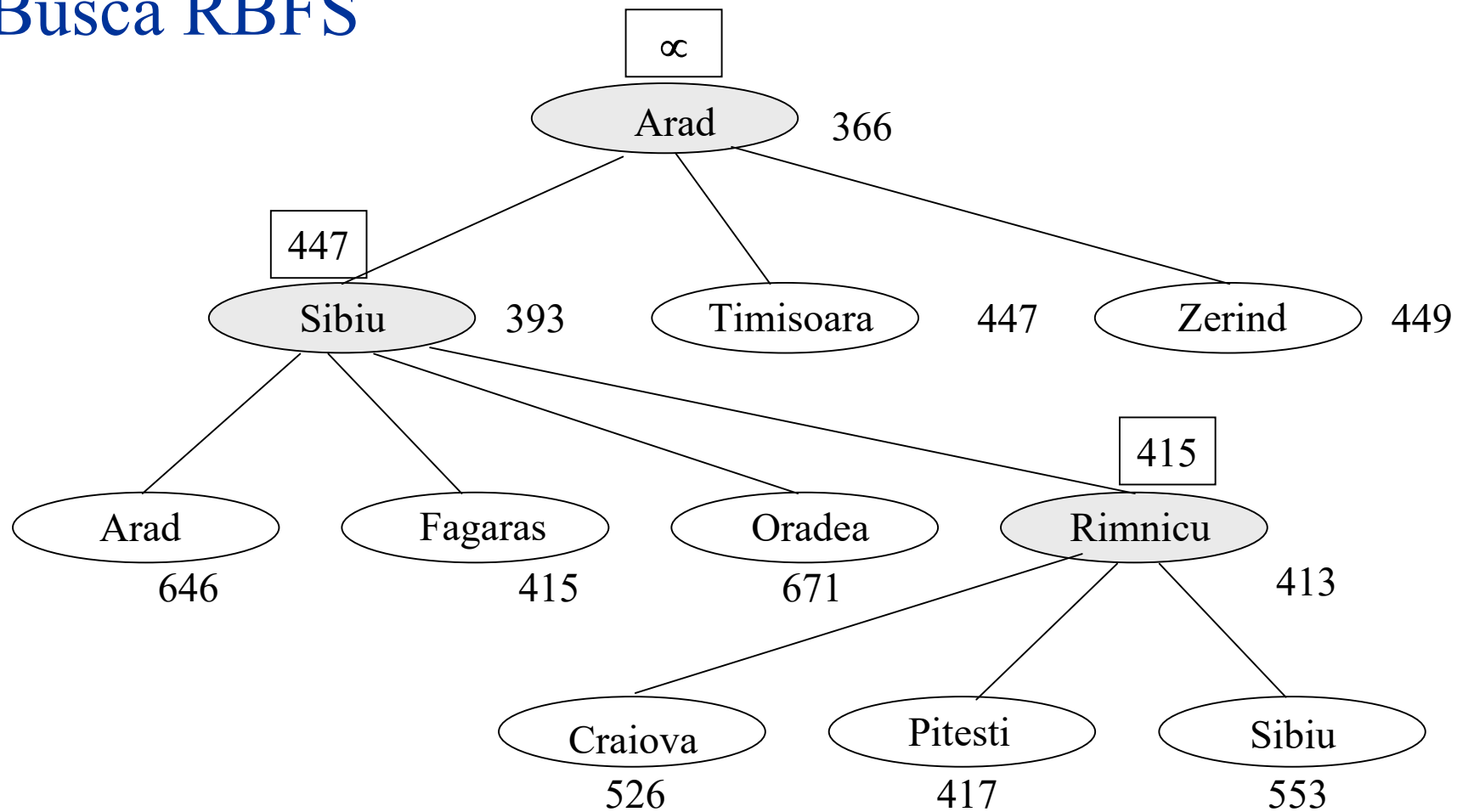
- ideia: mimetizar *best-first*, mas com espaço linear
- limite: f_limit para rastrear f -value da melhor alternativa dos ancestrais
- valor de corte: $newcutoff = \min \{f\text{-cost dos nós com } f\text{-cost} > oldcutoff\}$
- se nó corrente excede limite, algoritmo volta para caminho alternativo
- atualiza f -value de cada nó no caminho com um valor: **backed-up value**
- **backed-up value**: melhor f -value dos filhos do nó

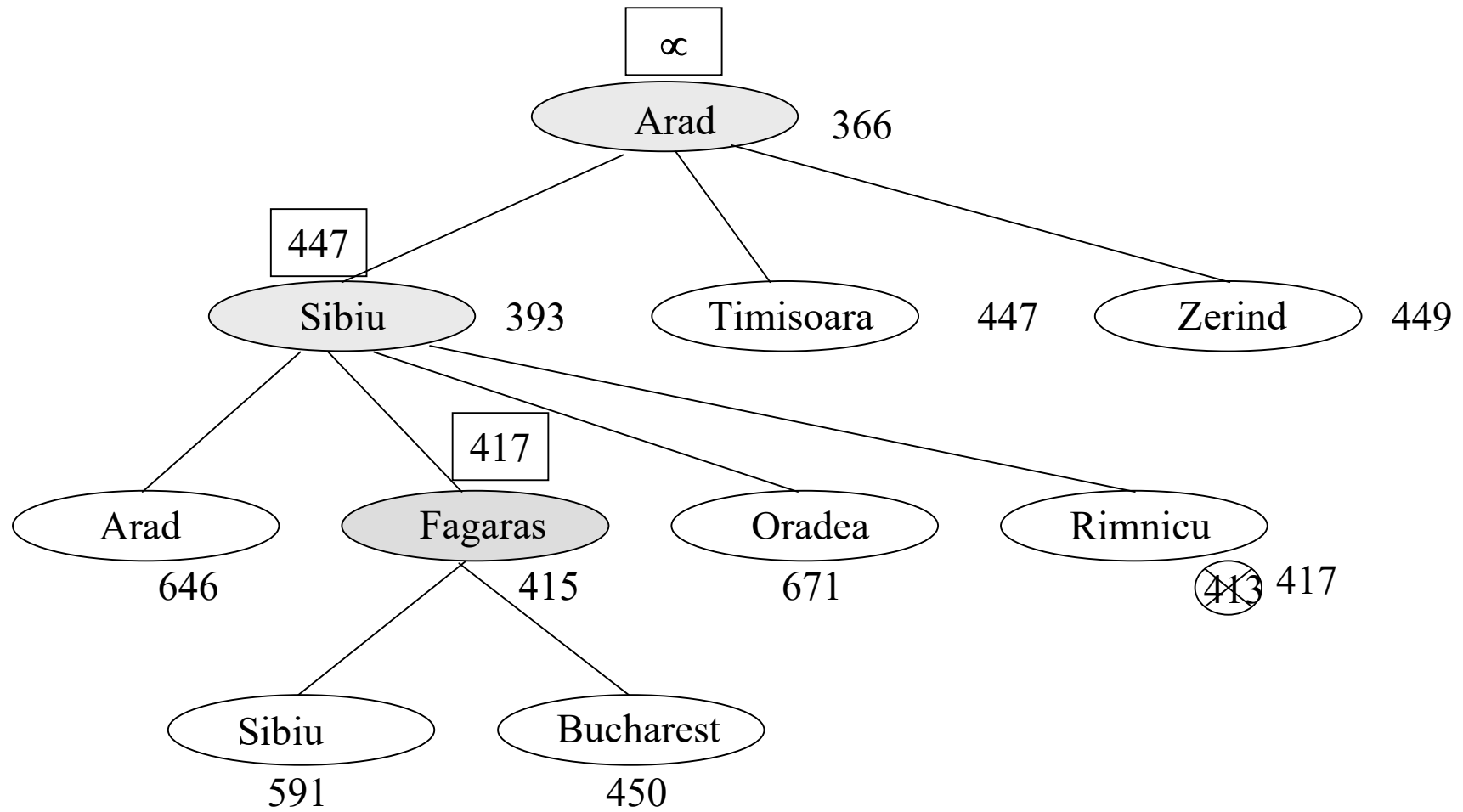
Algoritmo de busca RBFS

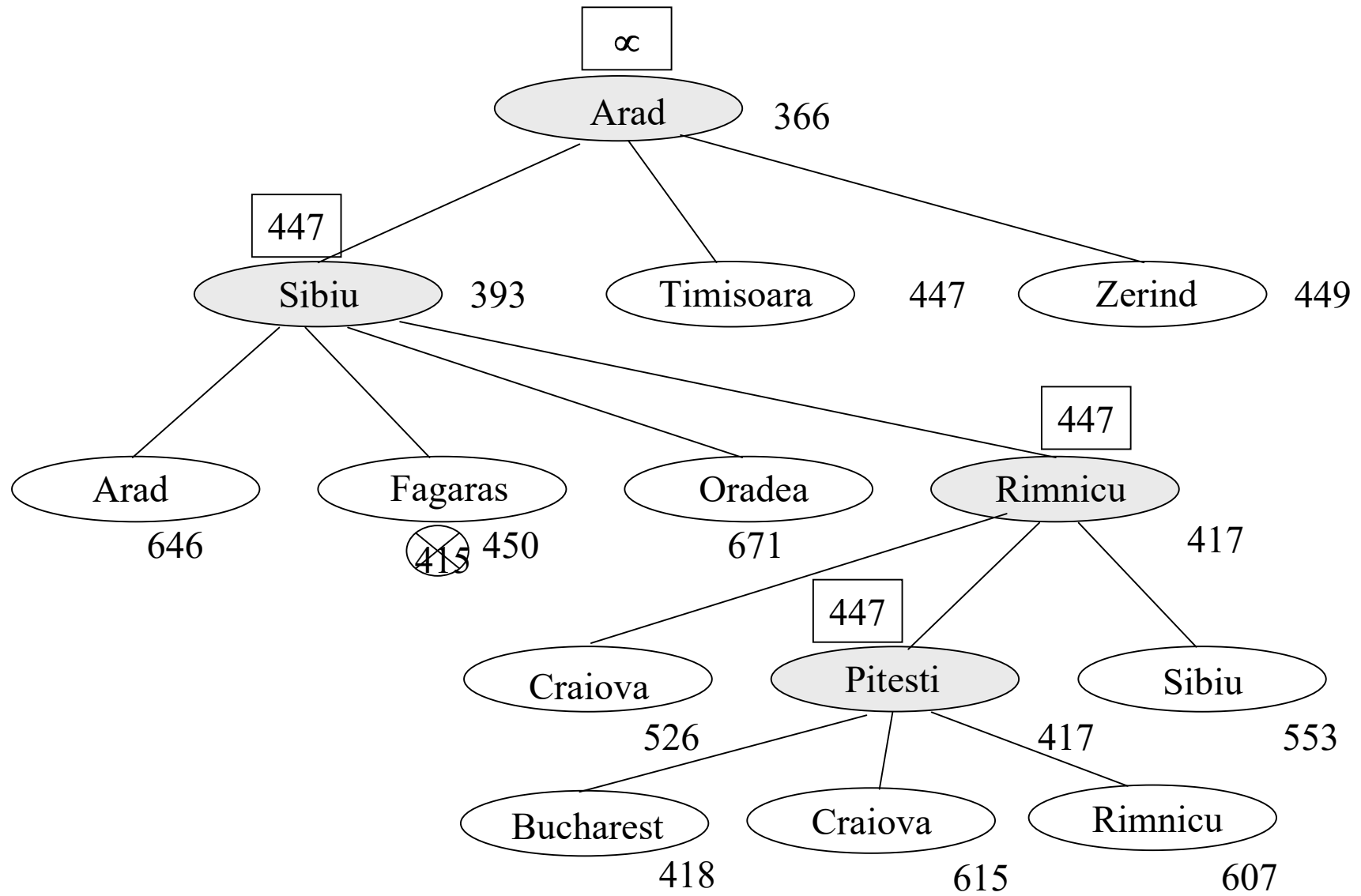
function RECURSIVE_BEST_FIRST_SEARCH (*problem*) **returns** a solution, or failure
return RBFS (*problem*, MAKE-NODE(*problem*.INITIAL-STATE, ∞)

function RBFS (*problem*, *node*, *f_limit*) **returns** a solution, or failure and new *f*-cost limit
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
successors \leftarrow []
for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
add CHILD-NODE(*problem*, *node*, *action*) into *successors*
if *successors* is empty **then return** failure, ∞
for each *s* **in** *successors* **do** /* update *f* with values from previous search, if any */
s.f \leftarrow max (*s.g* + *s.h*, *node.f*)
loop do
best \leftarrow the lowest *f*-value node in *successors*
if *best.f* > *f_limit* **then return** failure, *best.f*
alternative \leftarrow the second-lowest *f*-value among *successors*
result, *best.f* \leftarrow RBFS (*problem*, *best*, min (*f_limit*, *alternative*))
if *result* \neq failure **then return** *result*

Busca RBFS







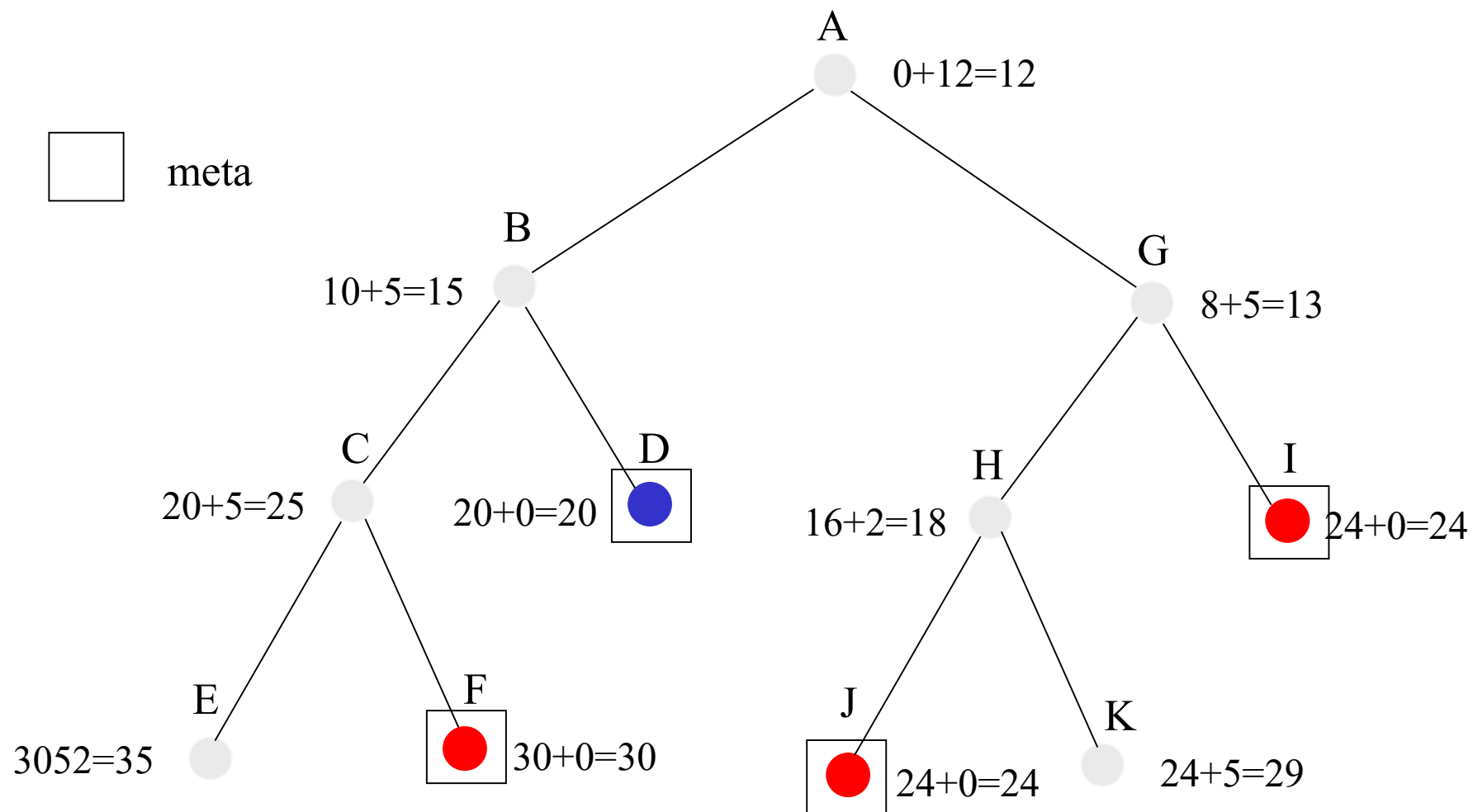
SMA* *Simplified memory-bounded A**

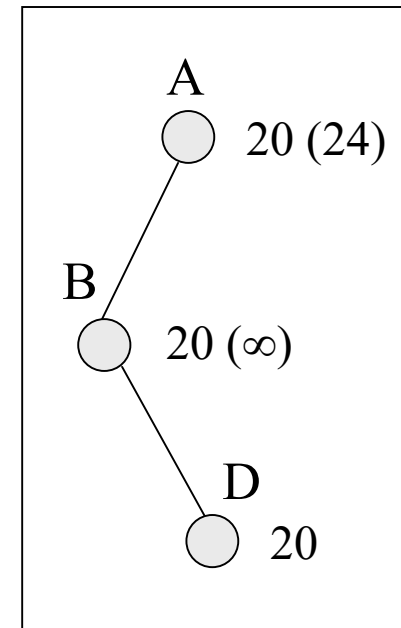
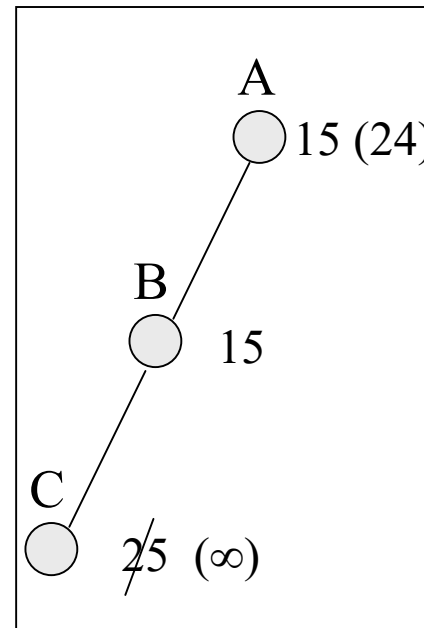
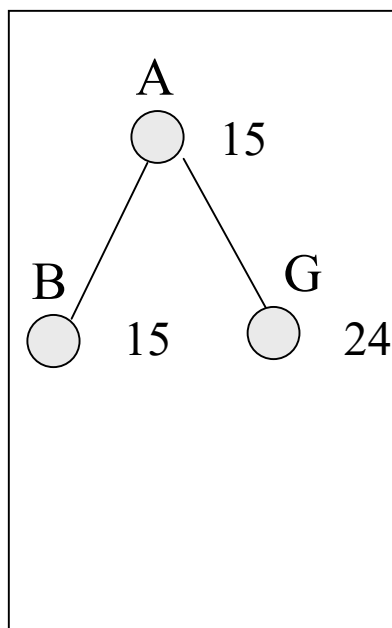
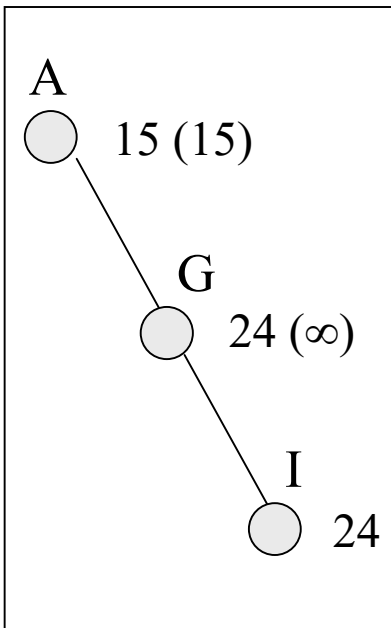
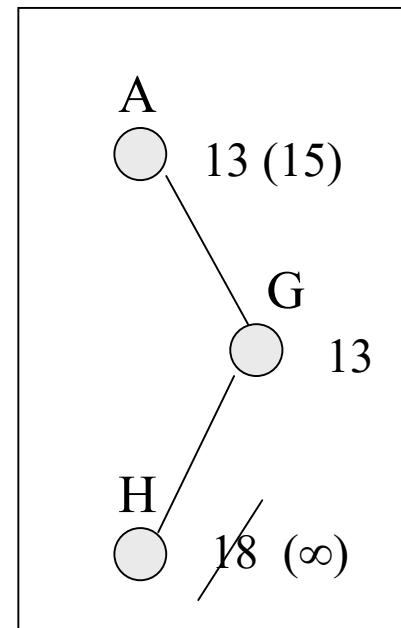
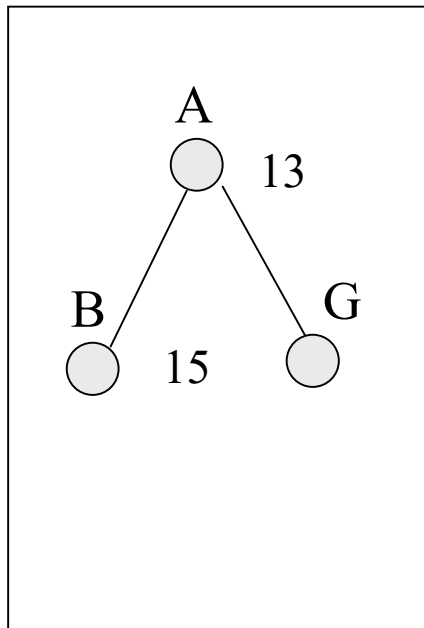
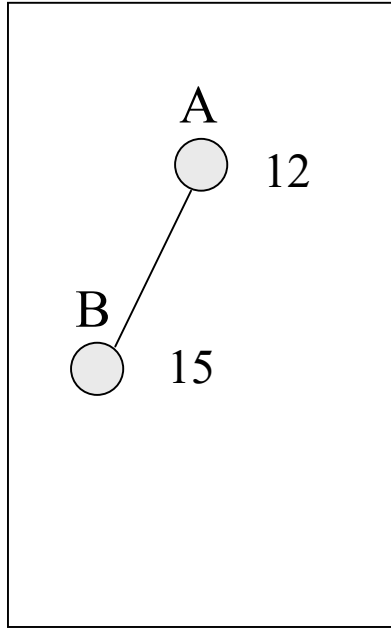
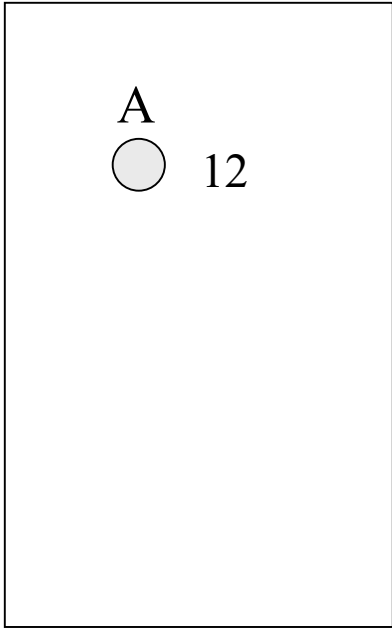
- utiliza a memória que estiver disponível
- evita repetição de expansão sempre que a memória permitir
- completo se a memória é suficiente para armazenar a solução mais raza
- ótimo se memória é suficiente para armazenar solução ótima
 - senão, fornece a melhor solução que pode ser encontrada

■ Propriedades do SMA*

- quando a memória é suficiente para toda árvore de busca ele é eficiente
- robusto para encontrar solução ótima quando
 - espaço de estado é um grafo
 - custos dos passos não são uniformes
 - geração de nós é caro, comparado com *overhead* para manter
 - *frontier*
 - *explored*

SMA*: exemplo





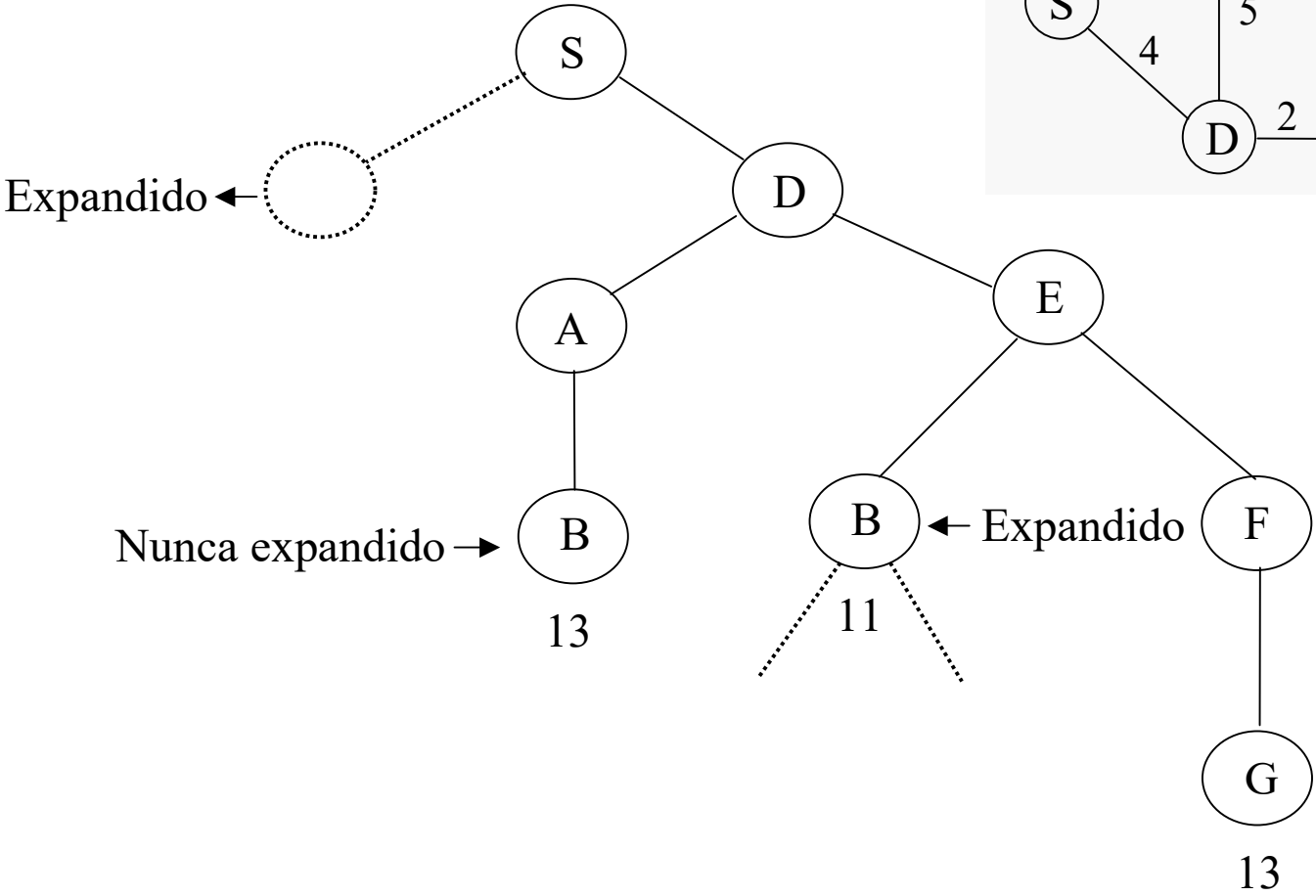
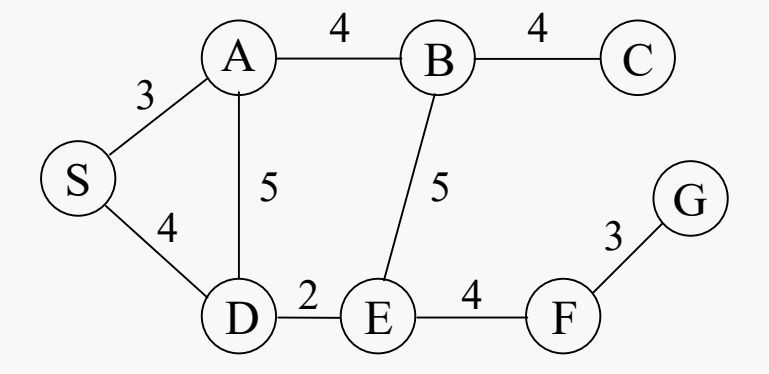
Comentário sobre o algoritmo A*

Winston (Artificial Intelligence, 3rd Edition, Addison Wesley, 1993) descreve o algoritmo A* de outra maneira, resumida abaixo:

$A^* = \textit{Branch-and-bound} + \textit{Princípio otimalidade Bellman}$

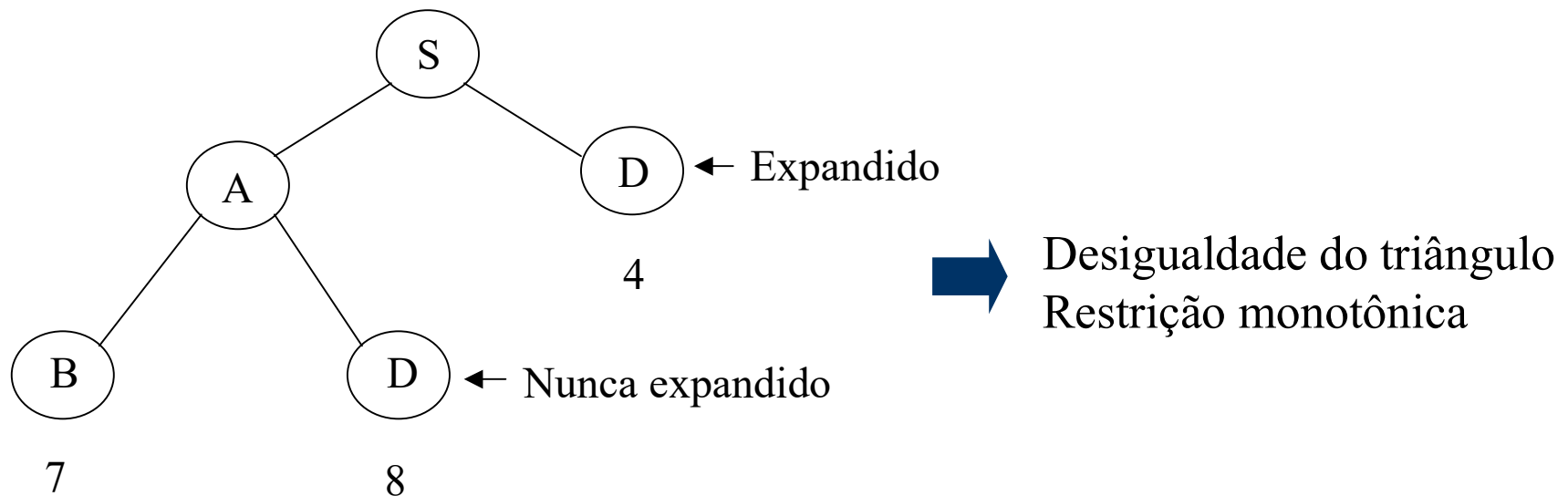
Esta maneira nos ajuda a explicitar conceitos importantes que estão, de certa forma, implícitos no algoritmo como descrito por Russell&Norvig.

Branch-and-bound



Princípio de otimalidade de Bellman

O melhor caminho entre um nó inicial e uma meta que passa por um nó particular intermediário, é o melhor caminho do nó inicial até este, seguido pelo melhor caminho deste nó até a meta. Não é necessário considerar nenhum outro caminho que passa por este nó particular.



Esta é a ideia da programação dinâmica

A^* = *Branch-and-bound* + Princípio otimalidade

Construir uma fila com um caminho de comprimento zero contendo a raiz;

Repetir até que o primeiro caminho da fila contenha a meta ou a fila vazia;

remover o primeiro caminho da fila; criar novos caminhos estendendo o primeiro caminho até todos os nós vizinhos do nó terminal;

rejeitar todos novos caminhos com ciclos;


adicionar os caminhos restantes, se algum, na fila;

se um ou mais caminhos atingem o mesmo nó, eliminar todos eles, exceto aquele com o menor valor;


ordenar todos os caminhos de acordo com $f = g + h$, colocando os caminhos com *menor valor* à frente da fila;

Se a meta for encontrada, sucesso; caso contrário falha.

Funções heurísticas

7	2	4
5		6
8	3	1

estado inicial

	1	2
3	4	5
6	7	8

meta

profundidade média: 22 (nº passos)


$b \approx 3$

busca exaustiva com árvore: $3^{22} \approx 3.1 \times 10^{10}$ estados


$9!/2 = 181.440$ estados distintos atingíveis

redução de 170.000 se usar busca grafo

15-puzzle: 1013

7	2	4
5		6
8	3	1

estado inicial

	1	2
3	4	5
6	7	8

meta

h_1 : números fora do lugar correto $h_1 = 8$

h_2 : distância de Manhattan (soma distância horizontal e vertical)

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 2 = 18$$

profundidade da solução: 26

h_1, h_2 : admissíveis

■ Qualidade de uma heurística

- caracterizada pelo fator efetivo de ramificação b^*

A^* gera N nós, profundidade d

b^* fator de ramificação que uma árvore uniforme com profundidade d contém $N + 1$ nós

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

- ideal $b^* \approx 1$
- $h_2(n) \geq h_1(n)$, h_2 domina h_1
- dominância \Rightarrow maior eficiência

Comparação IDS \times A* com heurísticas h_1 e h_2

d	Número médio de nós gerados		
	IDS	A*(h_1)	A*(h_2)
6	680	20	18
12	3.644.035	227	73
24	—	39.135	1.641

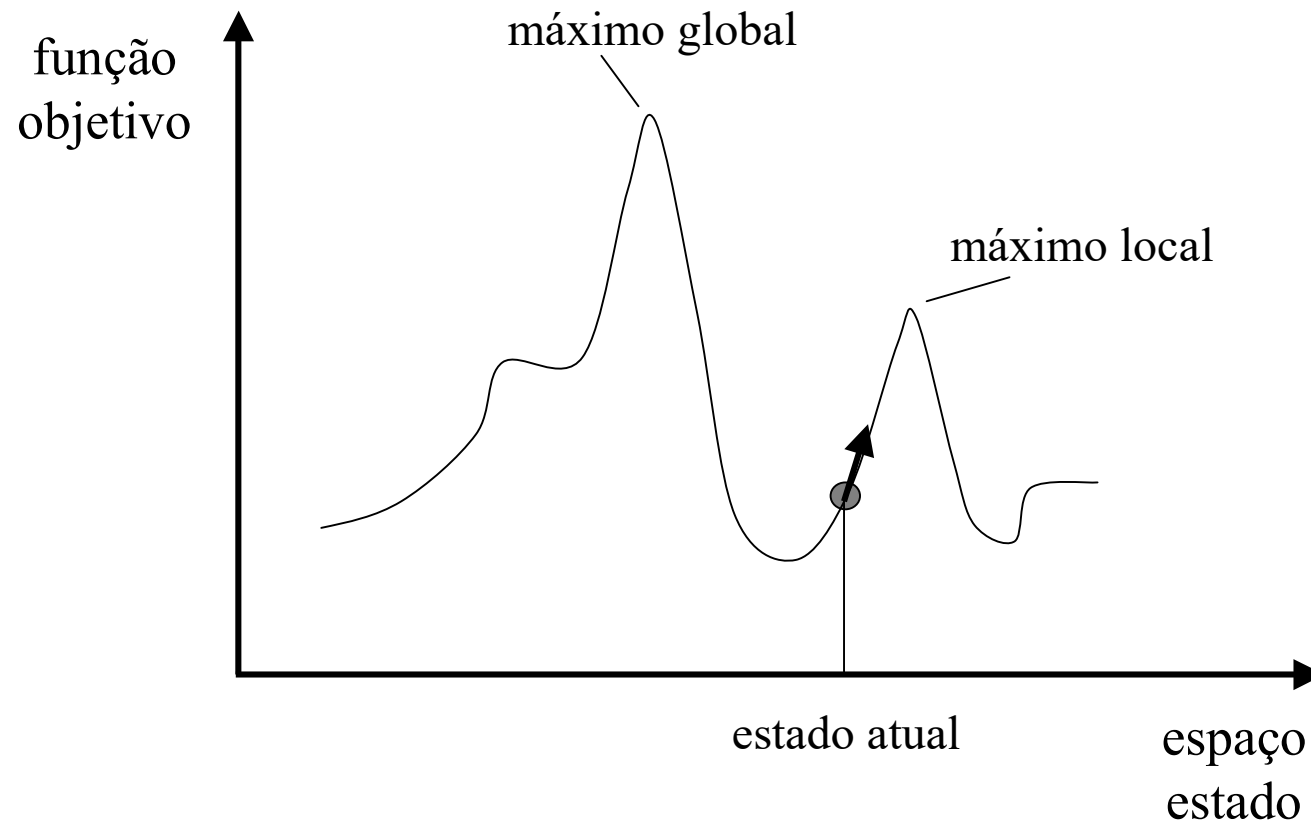
A* versão TREE_SEARCH, média de 100 instâncias para cada d

■ Geração de heurísticas

- relaxação
- *pattern databases*
- experiência
- aprendizagem

Busca local

- estado é o que interessa, não caminho
- busca inicia com um nó
- move para vizinhos do nó
- necessitam de pouca memória
- operam em espaços contínuos e discretos
- completo: se encontrar uma meta (se existir)
- ótimo: se encontrar um ótimo global



Algoritmo do gradiente (*hill-climbing*)

function HILL_CLIMBING (*problem*) **returns** a state that is local maximum

current ← MAKE-NODE (*problem*.INITIAL-STATE)

loop do

neighbor ← a highest-valued successor of *current*

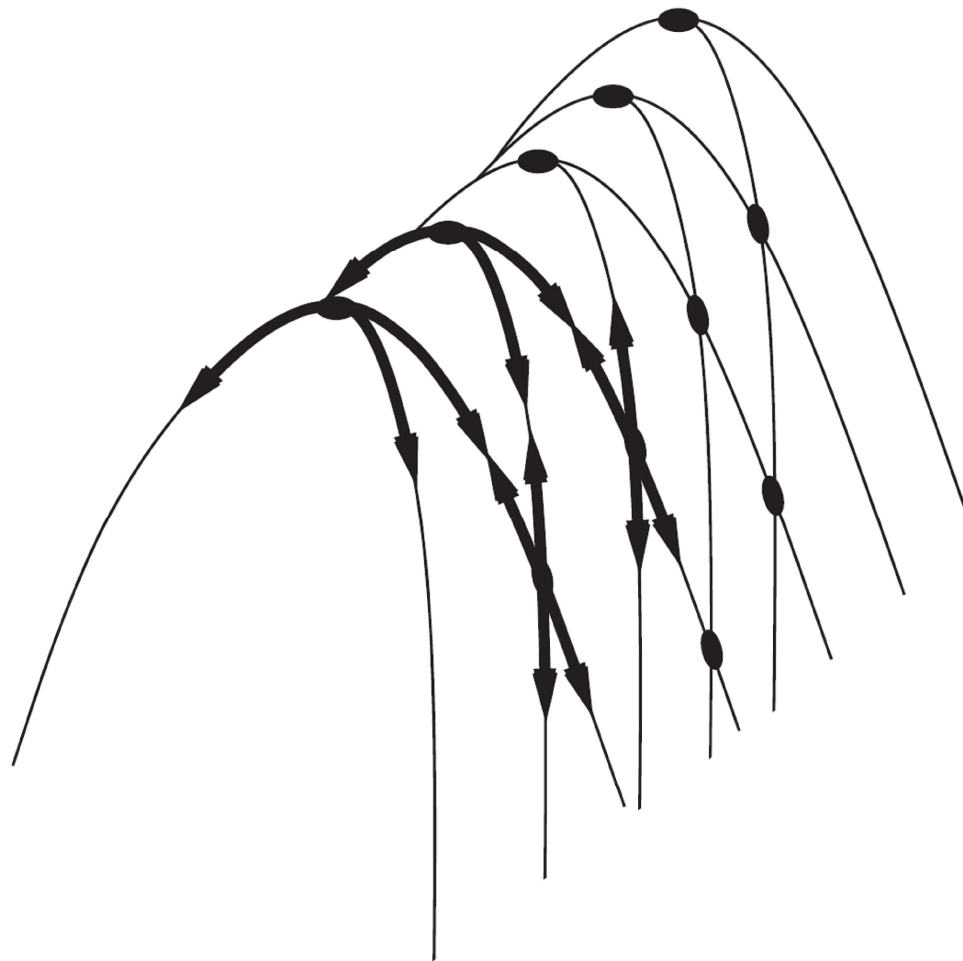
if *neighbor*.VALUE < *current*.VALUE **then return** *current*.STATE

current ← *neighbor*









■ Características

- não mantém uma árvore de busca
- estrutura dados nó: estado e valor função objetivo
- *complete state formulation*
- move para vizinhos imediatos do nó
- busca local gulosa
- problemas: ótimos locais, *plateaux*, *ridges*
- incompleto (ótimos locais)
- reinicializações aleatórias até encontrar meta:
 - torna-se completo com probabilidade 1

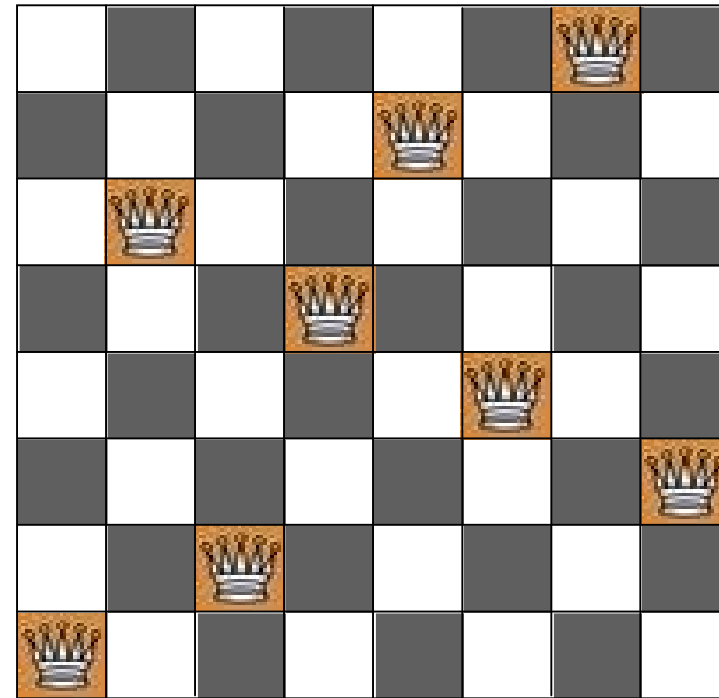
Ótimos locais



Exemplo: problema das 8 rainhas

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14		13	16	13	16
	14	17	15		14	16	16
17		16	18	15		15	
18	14		15	15	14		16
14	14	13	17	12	14	12	18

$h = 17$



$h = 1$ (mínimo local)

h = número de pares de rainhas que se atacam (direta e indiretamente)

■ 8 rainhas com busca local

- estado inicial: gerado aleatoriamente
- 86% pára (falha) depois de 4 passos (média)
- 14% acha solução depois de 3 passos (média)
- espaço estado: $8^8 \approx 17$ milhões estados!
- busca em *plateaux*: limite no número de iterações
 - 94 % resolvidos com 21 passos (média)
 - 6% de falhas com 64 passos em média
- reinicializações: problema com 3 milhões de rainhas em 3 min.!

Algoritmo *simulated annealing*

function SIMULATED_ANNEALING (*problem*, *schedule*) **returns** a solution state

inputs: *problem*, a problem
schedule, a mapping from time to “temperature”

current \leftarrow MAKE_NODE (*problem*.INITIAL-STATE)
for *t* \leftarrow 1 **to** ∞ **do**
 T \leftarrow *schedule* [*t*]
 if *T* = 0 **then return** *current*
 next \leftarrow a randomly selected successor of *current*
 $\Delta E \leftarrow$ *next*.VALUE – *current*.VALUE
 if $\Delta E > 0$ **then** *current* \leftarrow *next*
 else *current* \leftarrow *next* only with probability $\exp(\Delta E/T)$

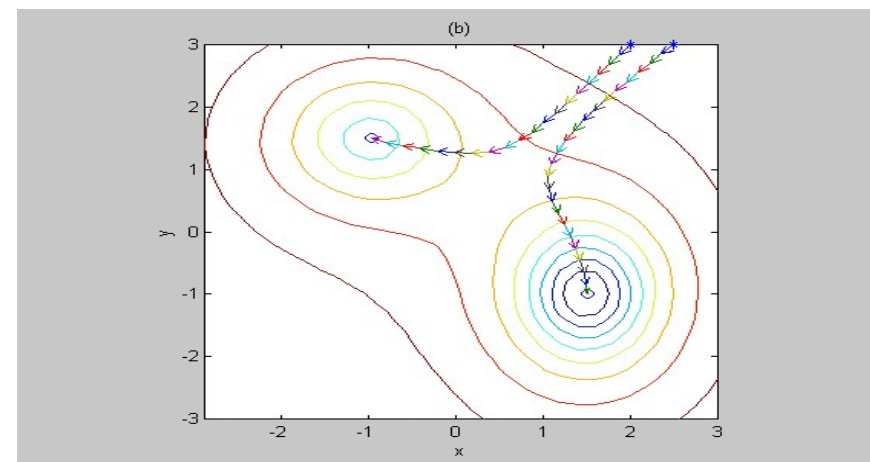
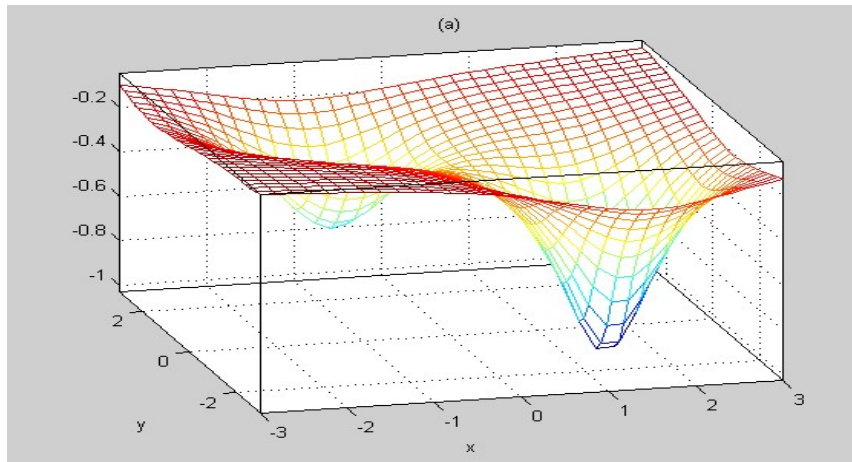
■ Características

- probabilidade diminui exponencialmente se a qualidade piora
- probabilidade diminui quando a temperatura diminui
- *schedule* diminui probabilidade suavemente
- ótimo global com probabilidade $\rightarrow 1$
- aplicações:
 - VLSI
 - planejamento/programação de operações

Local beam search

- mantém k nós (estados) ao invés de um único
- inicializado com k nós, gerados aleatoriamente
- gera todos os sucessores dos k nós
- se encontra meta: pára
- senão escolhe os k melhores e continua
- difere do *hill-climbing* com reinicializações
- problema: diversidade das k soluções
 - aliviado escolhendo k nós aleatoriamente

Busca local em espaços contínuos



$$\begin{aligned} & \max (\min) f(\mathbf{x}) \\ & \text{s.a.} \quad \mathbf{x} \in D \subseteq \mathbb{R}^n \end{aligned}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}) \quad \text{gradiente}$$

$$\mathbf{x} \leftarrow \mathbf{x} - H_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}) \quad \text{Newton}$$

$$D = \mathbb{R}^n$$

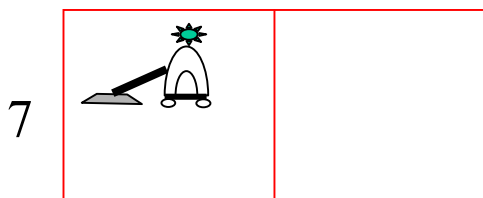
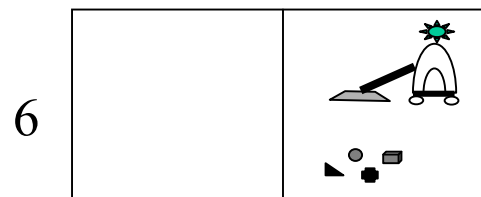
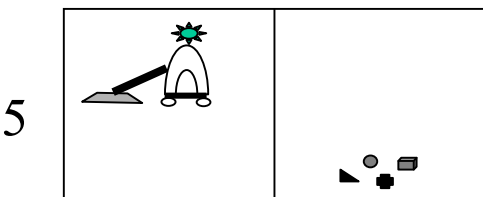
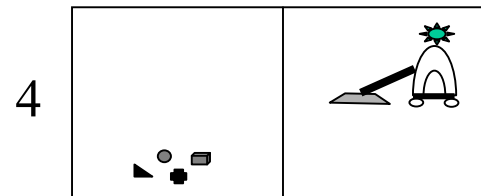
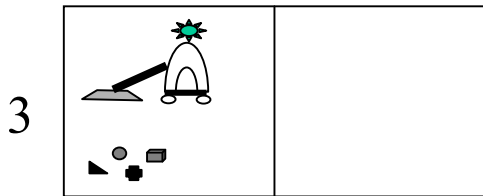
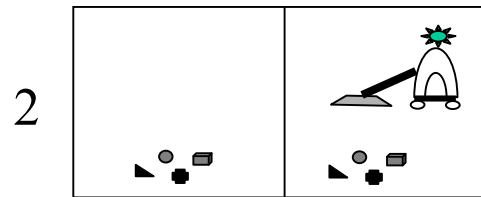
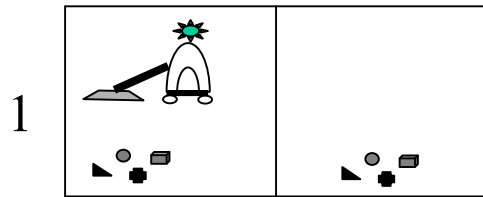
Busca com ações não determinísticas

- Ambiente
 - parcialmente observável
 - não determinístico
- Importância dos *percepts*:
 - ajuda a focalizar a busca
 - resultados das ações
- *Percepts* futuros são desconhecidos
- Solução do problema: estratégia (plano de contingência)

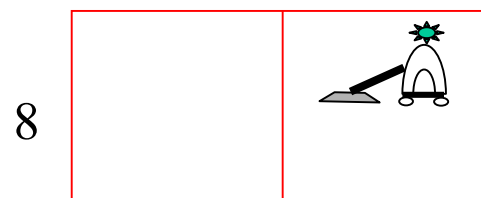
Exemplo 1: agente errático

- Ação *Suck*
 - posição com sujeira: limpa posição e eventualmente a adjacente
 - posição limpa: ação eventualmente deposita sujeira
- Modelo de transição
 - função RESULTS (ao invés de RESULT)
 - retorna um conjunto de estados
 - exemplo: $\{1\} \text{ Suck} \rightarrow \{5, 7\}$
- Solução
 - plano de contingência (estratégia)
 - [*Suck*, **if** *State* = 5 **then** [*Right*, *Suck*] **else** []]

Espaço de estados do problema



meta



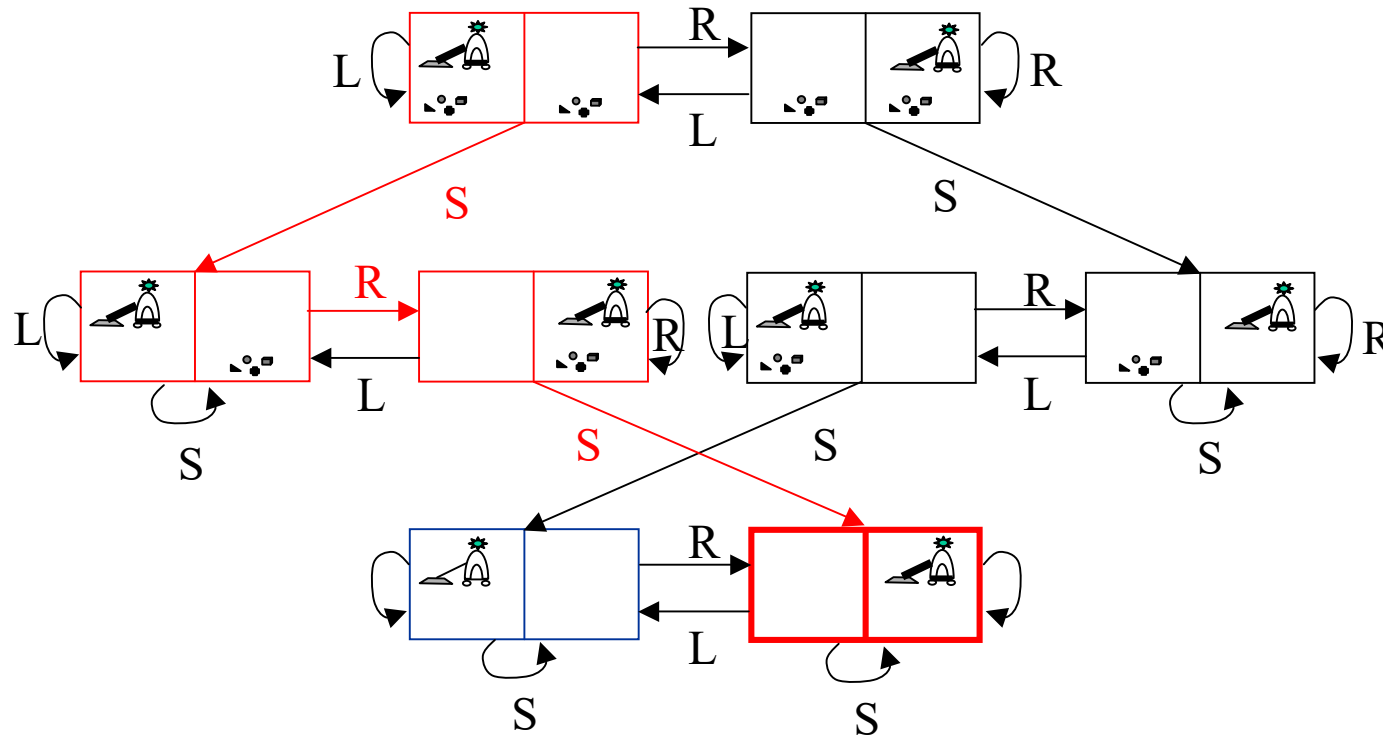
meta

- Se

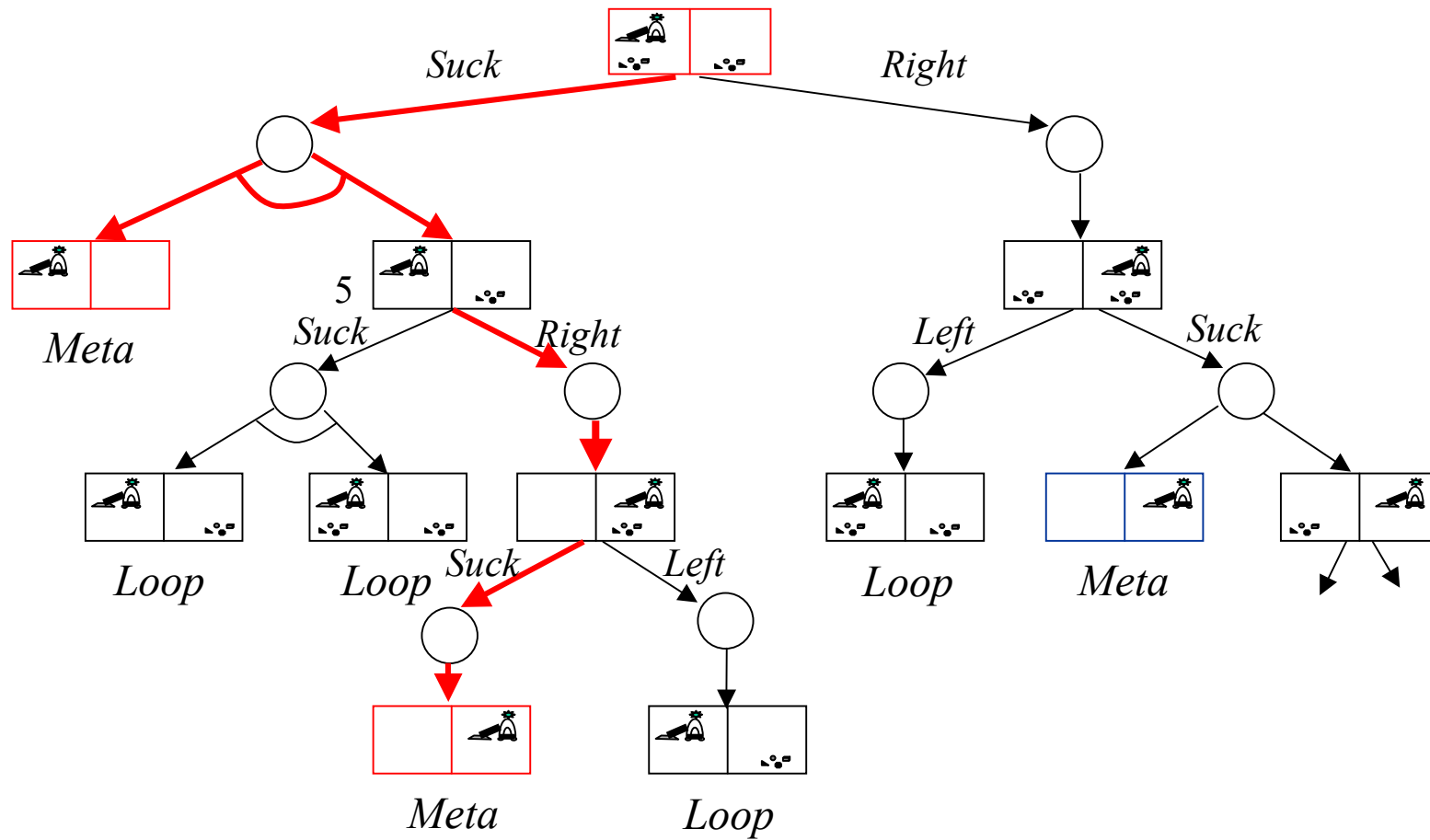
- ambiente: observável, determinístico, completamente conhecido
- agente conhece o estado onde está
- efeito das ações são conhecidos

- Então

- solução: sequência de ações
- *percepts* são irrelevantes



Árvores de busca And-Or



- Solução em um grafo AND-OR: sub-árvore que
 - tem uma meta em cada folha
 - especifica uma ação em cada nó OR
 - inclui todos descendentes em cada nó AND
 - solução: é um plano do tipo

[*Suck*, **if** *State* = 5 **then** [*Right*, *Suck*] **else**]

Algoritmo de busca em grafos And-Or

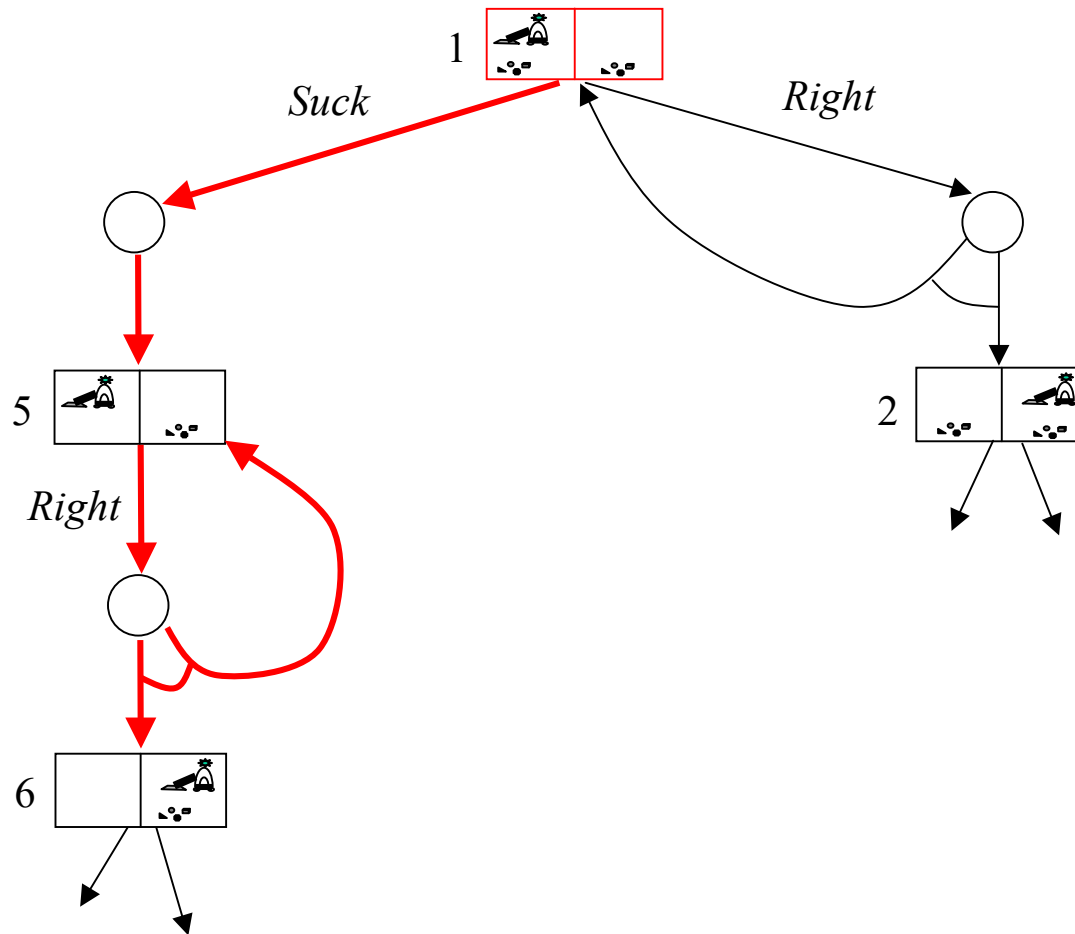
function AND_OR_GRAPH_SEARCH (*problem*) **returns** a conditional plan, or failure
OR_SEARCH (*problem*.INITIAL-STATE, *problem*, [])

function OR_SEARCH (*state*, *problem*, *path*) **returns** a conditional plan, or failure
if *problem*.GOAL-TEST(*state*) **then return** the empty plan
if *state* is on *path* **then return failure**
for each *action* **in** *problem*.ACTIONS(*state*) **do**
 plan ← AND_SEARCH(RESULTS(*state*, *action*), *problem*, [*state*|*path*])
 if *plan* ≠ *failure* **then return** [*action*|*plan*]
return failure

function AND_SEARCH (*states*, *problem*, *path*) **returns** a conditional plan, or failure
for each s_i **in** *states* **do**
 *plan*_{*i*} ← OR_SEARCH(s_i , *problem*, *path*)
 if *plan*_{*i*} = *failure* **then return failure**
return [**if** s_1 **then** *plan*₁ **else if** s_2 **then** *plan*₂ **else....if** s_{n-1} **then** *plan*_{*n-1*} **else** *plan*_{*n*}]

Exemplo 2: agente com falha no movimento

- Ação R (*Right*) e L (*Left*) do agente
 - falha eventualmente
 - agente permanece no mesmo local
 - exemplo: $\{1\} \textit{Right} \rightarrow \{1, 2\}$
- Solução cíclica
 - AND_OR_GRAPH_SEARCH falha
 - exemplo solução: aplicar *Right* até funcionar
 - como? rotulando parte do plano



$[Suck, S_1: Right, \mathbf{if\ State = 5\ then\ S_1\ else\ Suck}]$

while $State = 5$ **do** $Right$

Busca com observações parciais

- Conceito importante

- estado crença (*belief state*)
- conjunto de estados físicos possíveis
(dados: sequência de ações e *percepts*)

Exemplo 1: busca sem observações

Agente sem sensores

- *percepts* não fornecem nenhuma informação

Hipótese: agente conhece geografia do seu mundo

- mas não conhece sua posição
- não conhece a distribuição de sujeira

Estado inicial: um elemento de $\{1, 2, 3, 4, 5, 6, 7, 8\}$

- *Right* $\rightarrow \{2, 4, 6, 8\}$
- [*Righ*, *Suck*] $\rightarrow \{4, 8\}$
- [*Righ*, *Suck*, *Left*, *Suck*] sempre atinge $\{7\}$ (meta)
para qualquer estado inicial

- Espaço de estados crença totalmente observável
- *Percepts* observados depois das ações são previsíveis
 - são sempre vazios!
- Não há contingências
- Formulação problemas busca com estados crença ?

■ Formulação do problema

Problema subjacente P

– $ACTIONS_P$, $RESULT_P$, $GOAL-TEST_P$, $STEP-COST_P$

1. espaço de estados crença

– conjunto de todos estados possíveis de P

– se P tem N estados, então 2^N estados possíveis

– nem todos estados são atingíveis

2. estado inicial

– conjunto de todos estados de P

3. Ações

- dado um estado crença $b = \{s_1, s_2\}$
- em geral $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$
- se todas ações são aplicáveis

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s)$$

- senão

$$\text{ACTIONS}(b) = \bigcap_{s \in b} \text{ACTIONS}_P(s)$$

conjunto das ações aplicáveis em todos os estados

4. Modelo de transição

- se ações são determinísticas

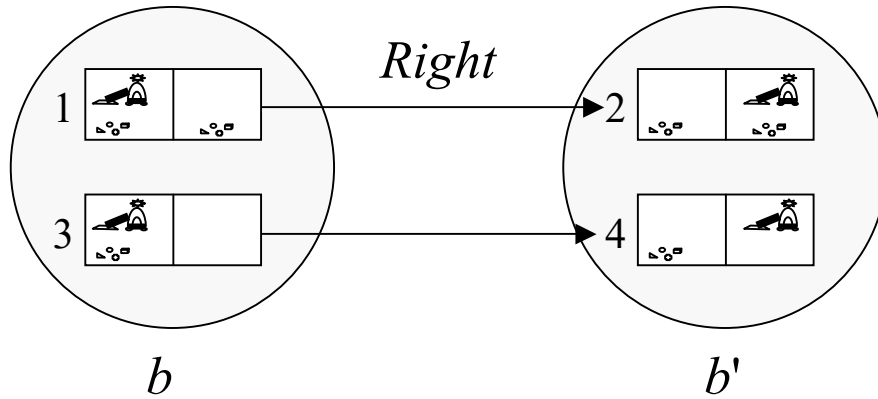
$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

- se ações são não determinísticas

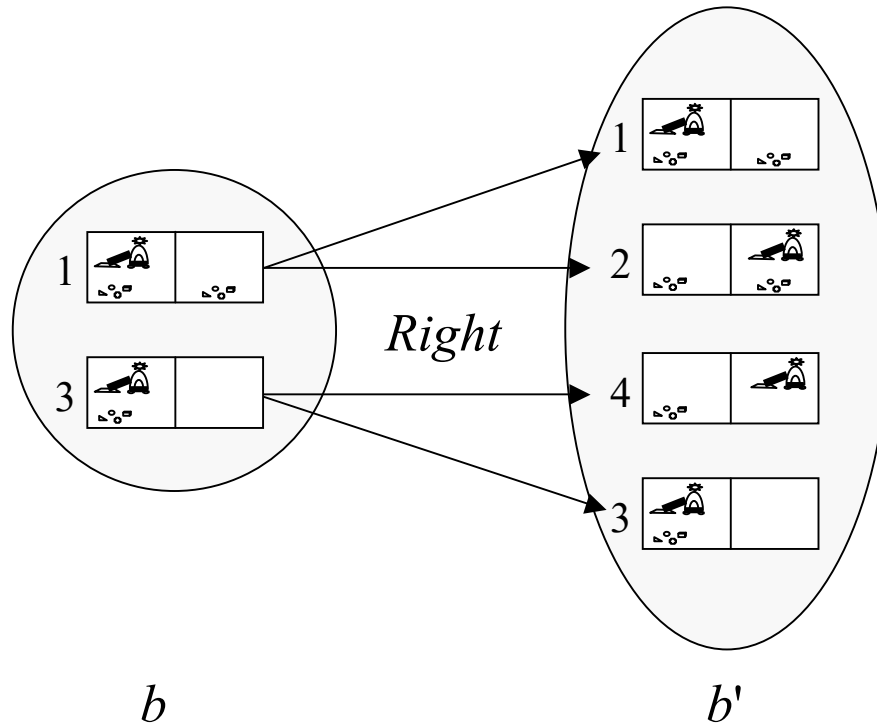
$$\begin{aligned} b' = \text{RESULT}(b, a) &= \{s' : s' \in \text{RESULTS}_P(s, a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}(s, a) \end{aligned}$$

- previsão: $b' = \text{PREDICT}_P(b, a)$

Previsão: $b' = \text{PREDICT}_P(b, a)$



Ação determinística



Ação não determinística

5. Teste de meta

- agente: necessita de um plano que funcione, com certeza
- estado crença satisfaz meta se todos seus elementos satisfazem $GOAL-TEST_P$

6. Custo de um caminho

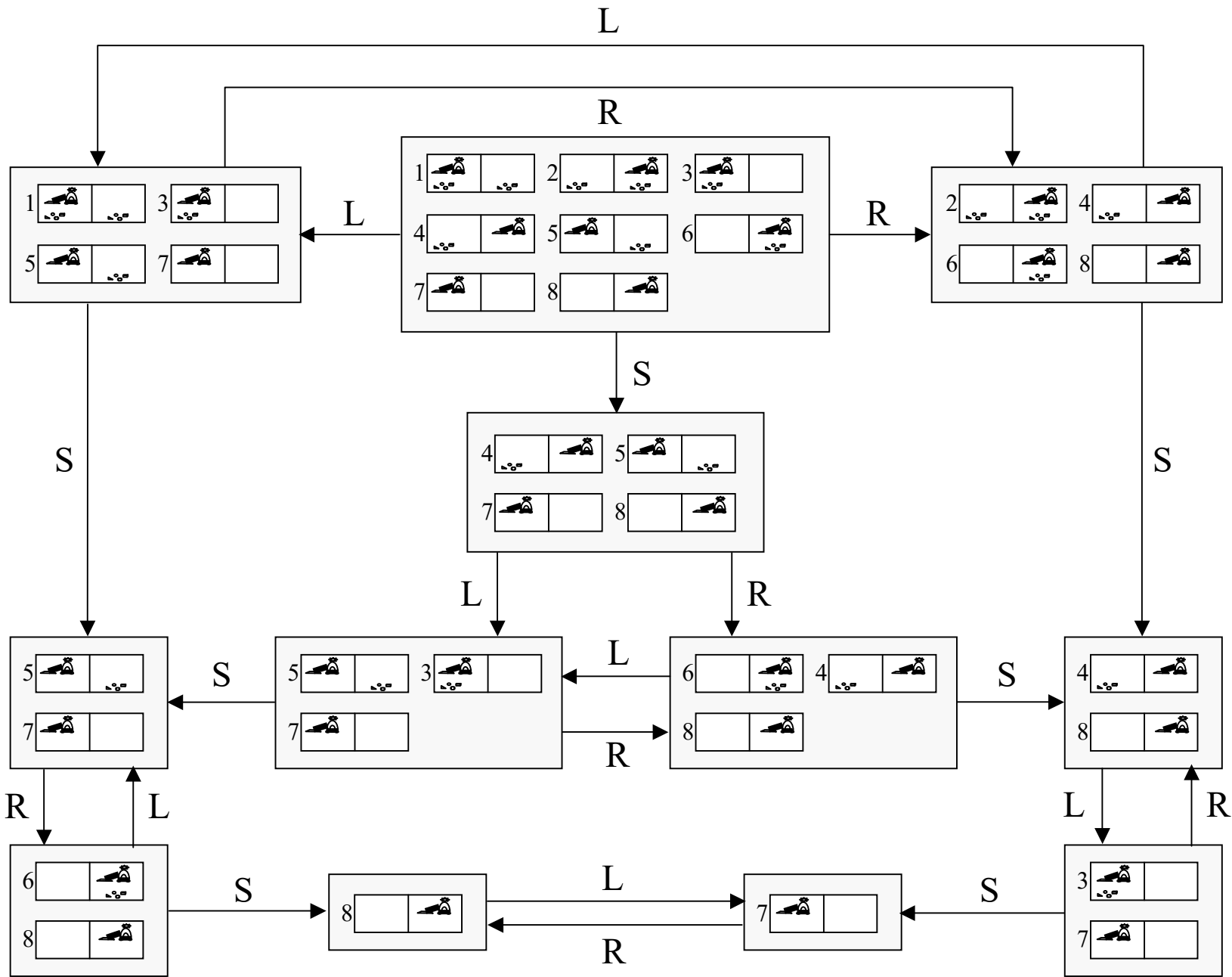
- ações podem ter diferentes custos
- aqui assumimos custos são os mesmos

- 12 estados crença atingíveis entre os $2^8 = 256$ possíveis
- algoritmos de busca informados/não informados são aplicáveis
- se uma sequência de ações é uma solução para um estado crença b então ela é também é solução para qualquer subconjunto de b

exemplo: [*Suck, Left, Suck*] \rightarrow {5, 7}

[*Left*] \rightarrow {1, 3, 5, 7} superconjunto de {5, 7}

podemos descartar {1, 3, 5, 7} se {5, 7} foi gerado



- inconveniente: dimensão dos estados crença
exemplo: aspirador 10×10 contém $100 \times 2^{100} \sim 10^{32}$ estados físicos
- solução: busca incremental
 - obter solução para cada estado físico, um de cada vez
solução deve funcionar para todos os estados
 - vantagem: detecta falha rapidamente
 - desvantagem: difícil de usar quando não existe uma solução

Exemplo 2: busca com observações parciais

- Agente com sensores
 - ambiente gera *percepts*
 - sensores do agente fornece informação locais
 - posição
 - sujeita na posição (não em outras posições)
- Definição do problema requer função $\text{PERCEPT}(s)$
 - determinístico: retorna o *percept* recebido em um estado s
exemplo: $\text{PERCEPT}(\{1\}) = [A, \text{Dirty}]$
 - não determinístico: PERCEPTS retorna conjunto de *percepts*
exemplo: observação (parcial) $[A, \text{Dirty}] \rightarrow$ estado crença inicial associado $\{1, 3\}$

■ Formulação do problema

ACTIONS, GOAL-TEST_P, STEP-COST_P

– idêntico ao caso do agente sem sensores

Modelo de transição

– transições: estado crença → ação → estado crença

– ocorrem em três estágios

1. predição
2. observação da predição
3. atualização

1. Predição

$$\hat{b} = \text{PREDICT}(b, a)$$

2. Predição da observação

- determina o conjunto de observações o que poderia ser observado no estado crença previsto

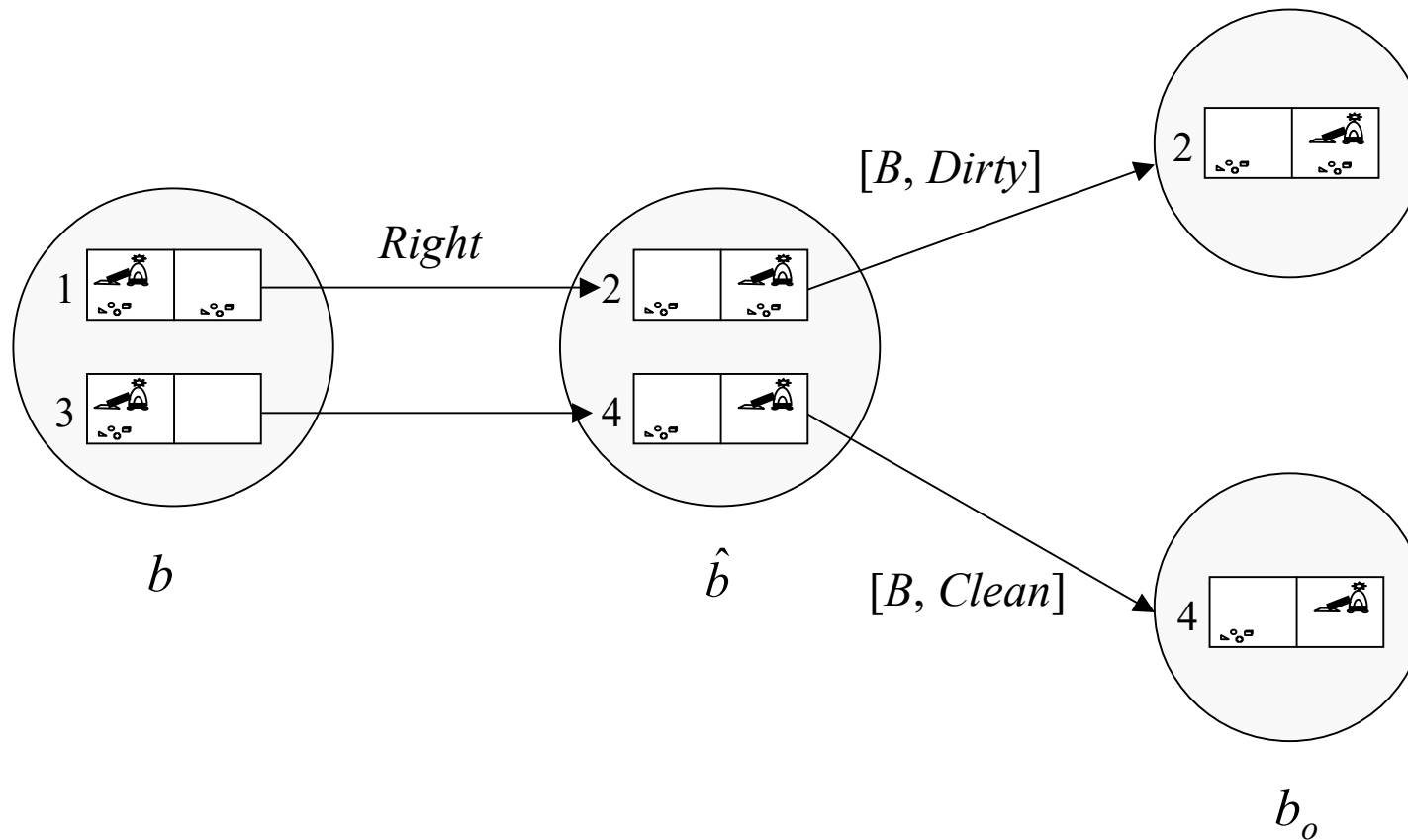
$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}$$

3. Atualização

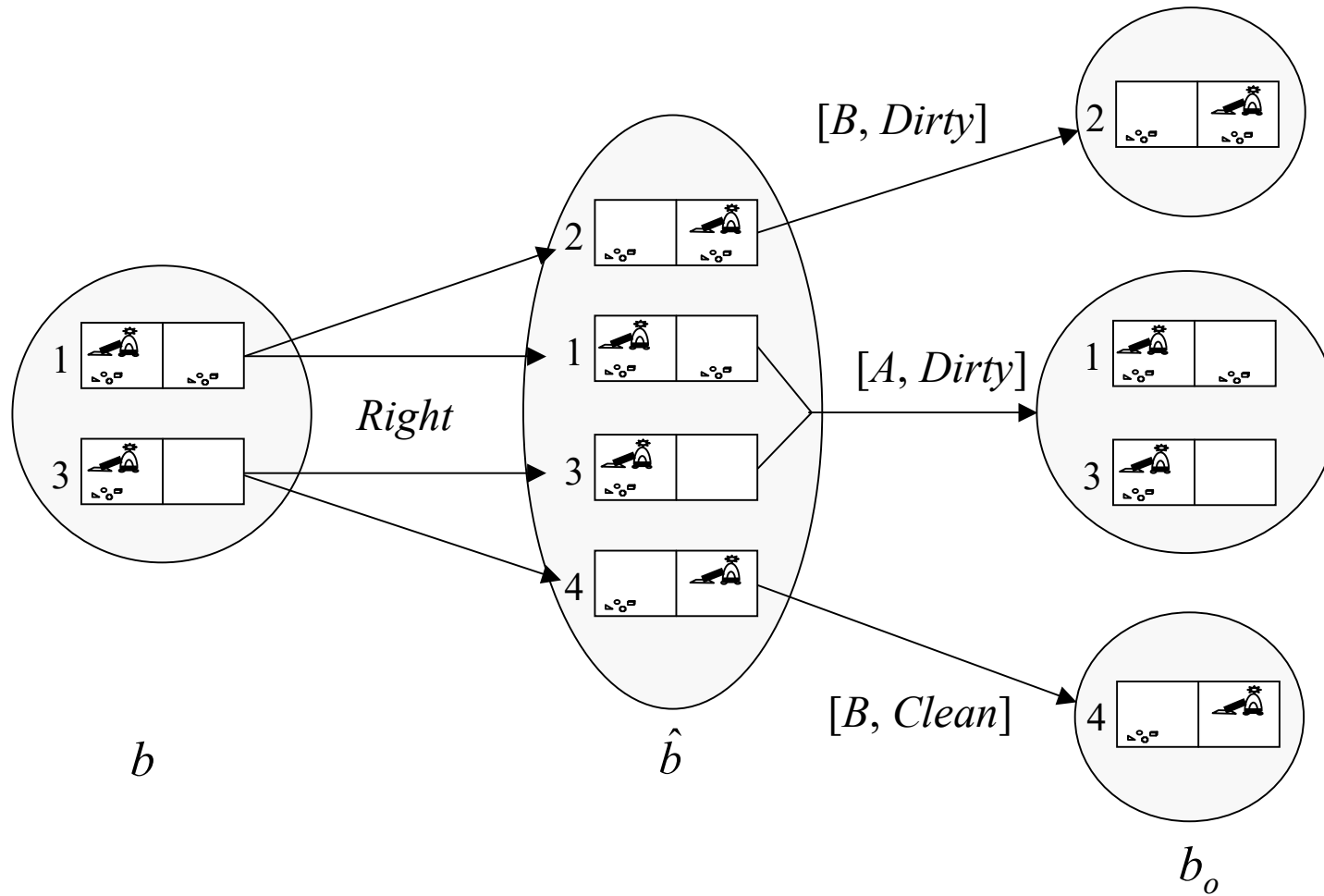
- determina, para cada *percept* possível, estado crença resultante

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}$$

Transição de estado com ação determinística



Transição de estado com ação não determinística



– observações ajudam a diminuir incerteza

$$|b_o| \leq |\hat{b}|$$

– ação/observação determinística

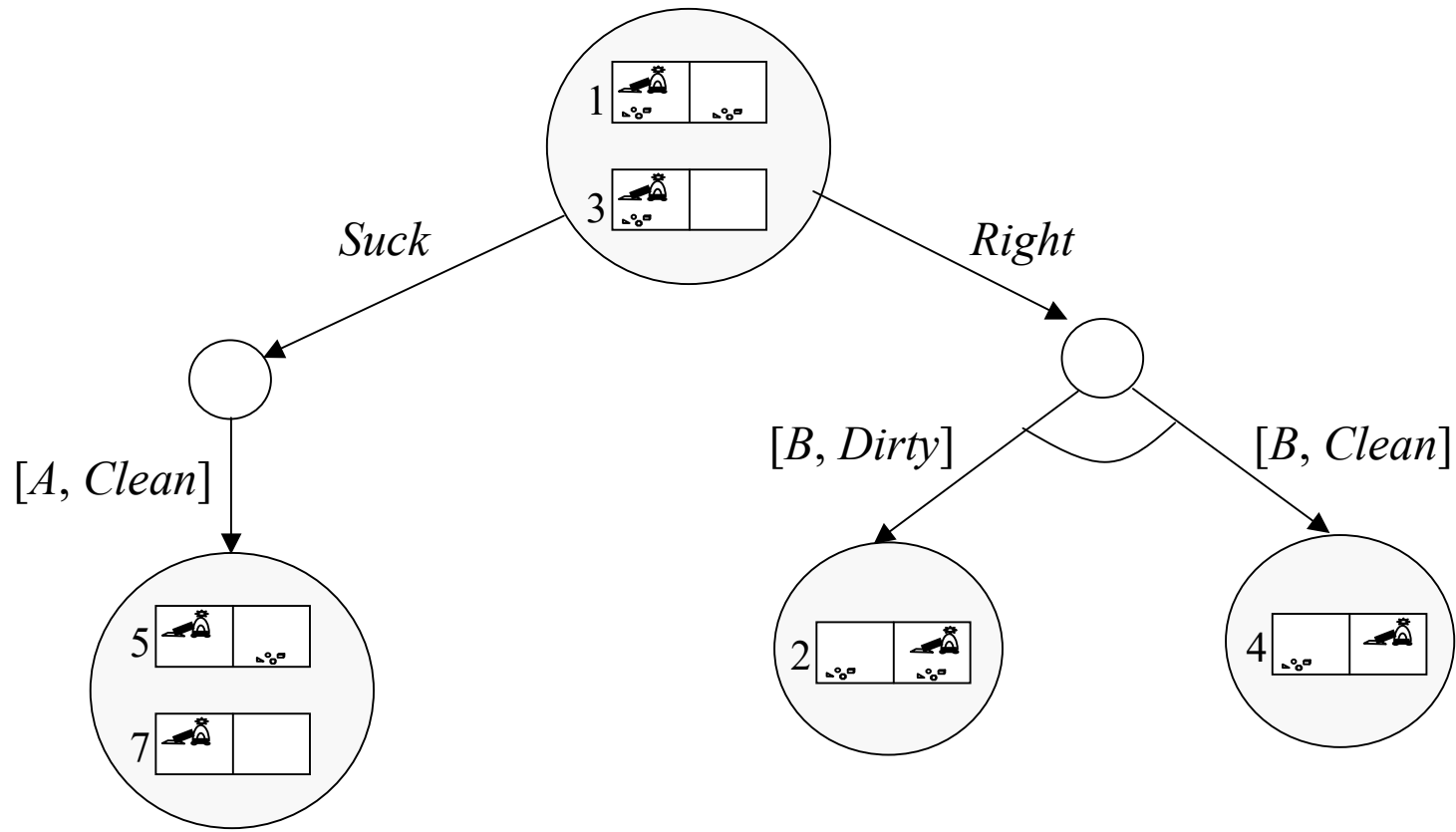
- estados crença para *percepts* possíveis diferentes são disjuntos
- estados crença formam uma partição do estado crença previsto

– agregando os três estágios:

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}$$

■ Solução do problema

- formular problema: *problem*
- executar AND_OR_GRAPH_SEARCH (*problem*)
 - busca considera estados crença
 - algoritmo verifica estados já gerados
 - busca pode ser incremental
- solução é um plano de contingência



[Suck, Right, if BState = {6} then Suck else[]]

Agente em ambientes parcialmente observáveis

- Projeto do agente

- similar ao SIMPLE_PROBLEM_SOLVING_AGENT (*percept*)

- formula o problema
 - chama algoritmo busca
 - executa ações

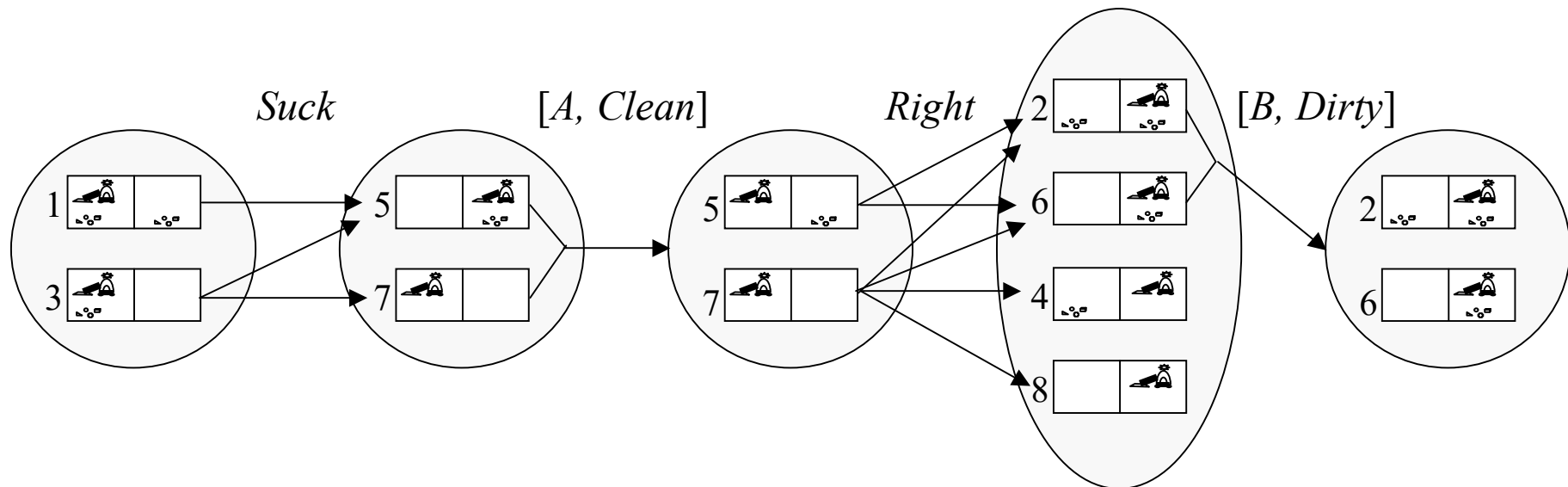
- Em ambientes parcialmente observáveis

- solução é um plano condicional
 - agente tem que manter o estado crença
 - estado crença é mais fácil de ser estimado

- se primeiro passo é **if-then-else**: verificar **if** e executar **then** ou **else**
- atualiza estado crença ao executar ações e receber *percepts*
- atualização estado crença mais simples porque
 - agente não estima/calcula o *percept*
 - agente usa o *percept* *o* fornecido pelo ambiente

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o)$$

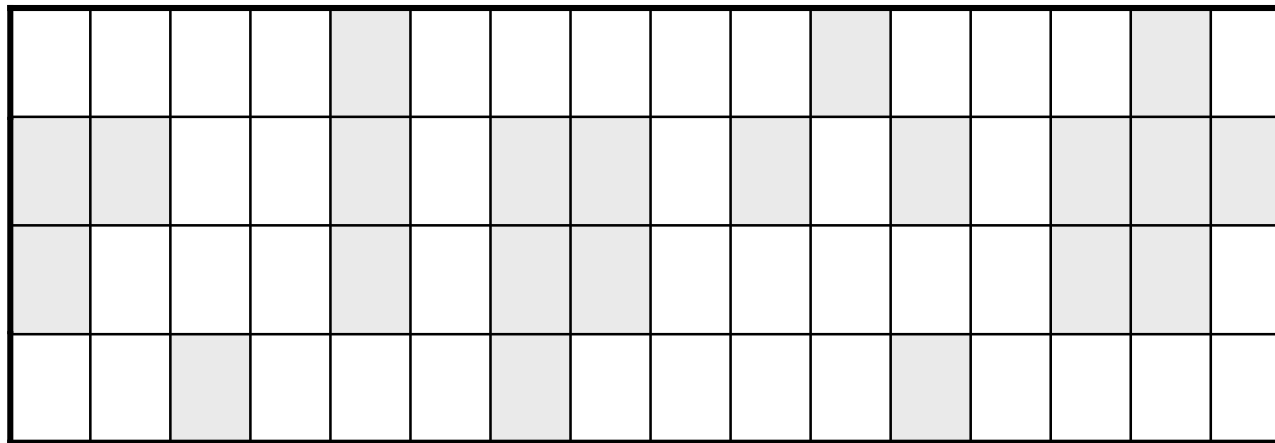
Ciclo predição-atualização do estado crença



$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o)$$

kindergarten world com sensor local: qualquer local pode ficar sujo, em qualquer instante, a menos que o agente esteja aspirando o ambiente

Exemplo: localização posição



robô



obstáculos



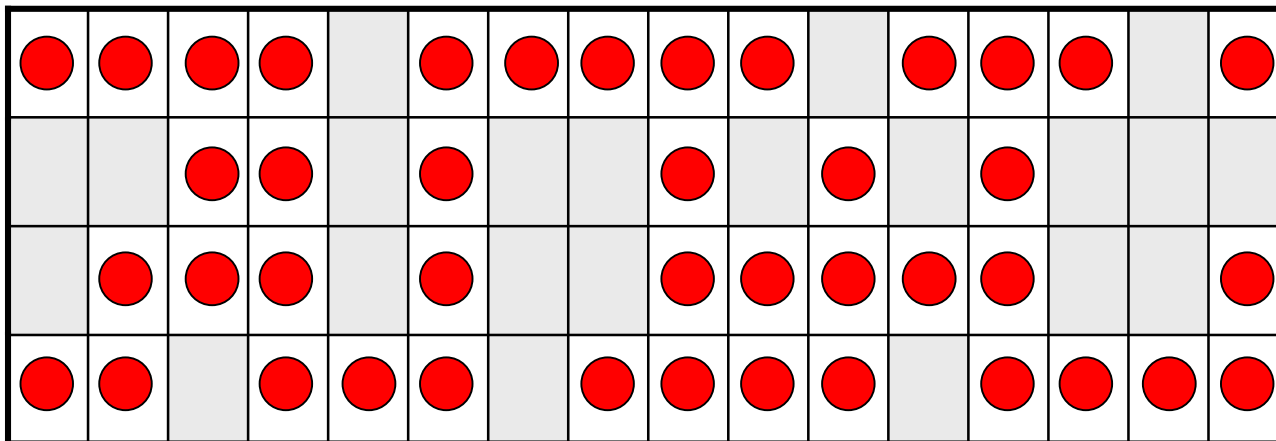
sensores: 4 sonares

detectam obstáculos

direção obstáculo: *N*, *S*, *W* ou *E*

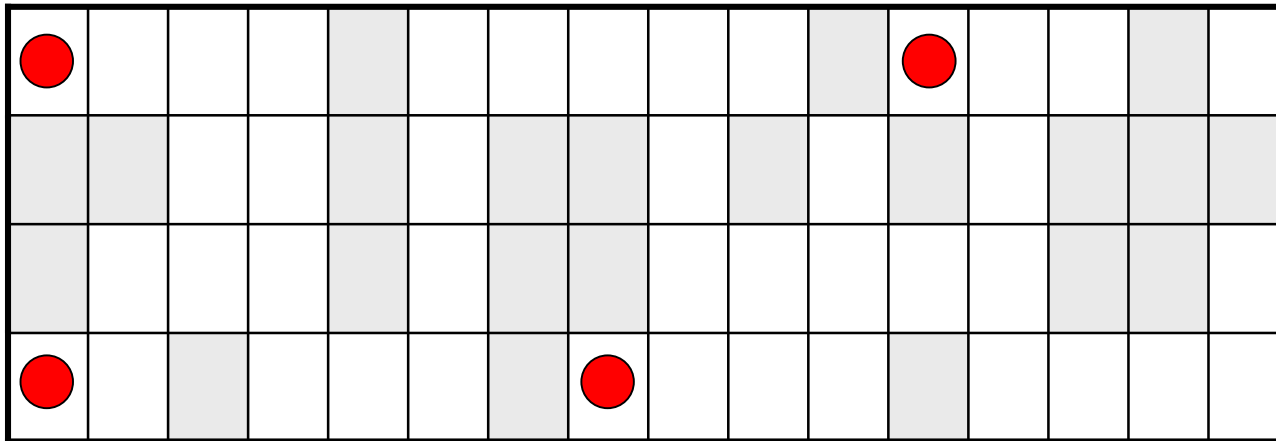
ação: *Move* (posição adjacente aleatória)

Posição atual do robô ?



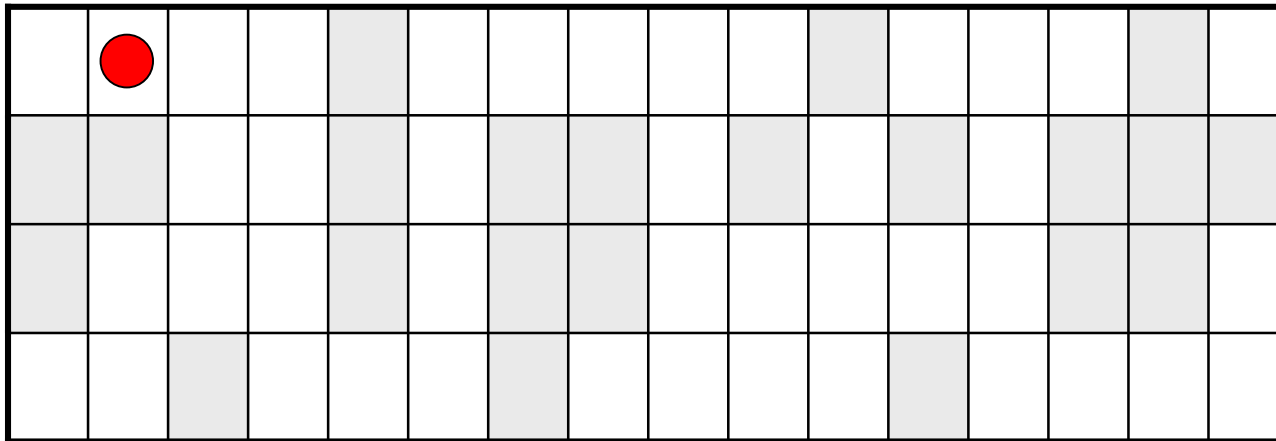
estado inicial: b

$o = NSW$



$b_o = \text{UPDATE}(b, o)$

$$o = NS$$



$$b_a = \text{PREDICT}(b_o, \text{Move})$$

$$b_n = \text{UPDATE}(b_a, o)$$

$$b_n = \text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, \text{NSW}), \text{Move}), \text{NS})$$

Agentes com busca *online*

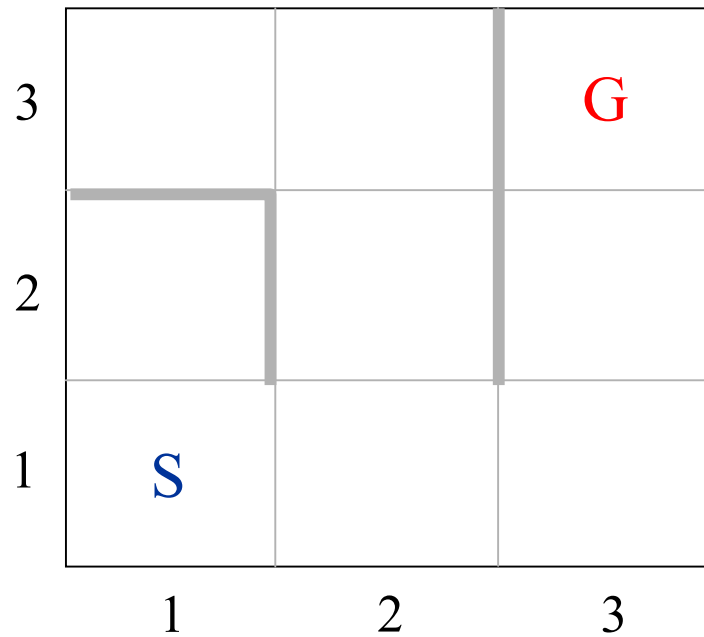
Agente online

- executa ciclos: ação – observação – ação – ...
- importante em ambientes:
 - dinâmicos, não determinísticos
- necessário em ambientes desconhecidos
 - não conhece estados
 - não sabe que ações executar
 - ações como experimentos de aprendizagem

■ Conhecimento do agente

- $ACTIONS(s)$: retorna lista das ações permitidas no estado s
- $c(s, a, s')$: custo de um passo; só pode ser calculado quando agente sabe que está em s'
- $GOAL-TEST(s)$
- $RESULT(s, a)$: determinado somente quando agente está em s e executa ação a
- Função heurística $h(n)$
- Objetivo: atingir meta com menor custo, explorar ambiente

Exemplo



S estado inicial

G meta

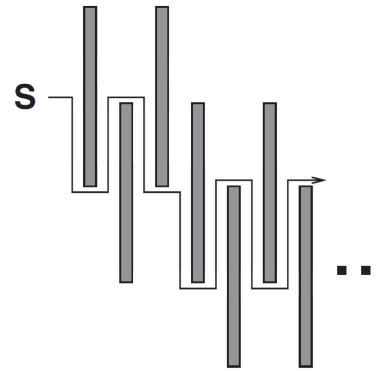
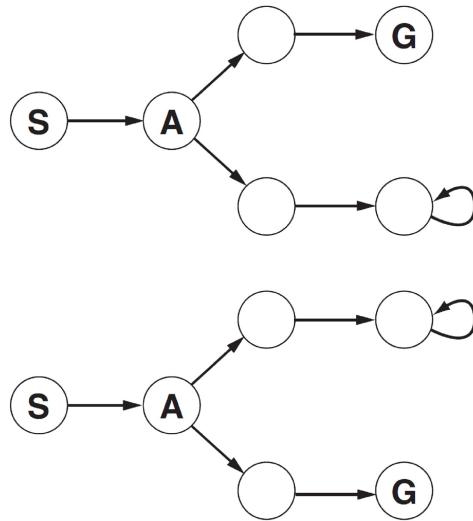
Ações: *UP*, *Down*

$h(n)$: distância Manhattan

Agente não sabe que da posição (1, 1) e ação *UP* → nova posição (1, 2)

■ Avaliação do agente

- razão competitiva
- pode ser infinita se ações são irreversíveis
- *dead-end*
- argumento adversário



Busca em profundidade *Online*

function ON_LINE_DFS_AGENT (s') **returns** an action

inputs: s' , a percept that identifies the current state

persistent: *result*, table indexed by state and action, initially empty

untried, table that lists, for each state, actions not yet tried

unbacktracked, table that lists, for each state, backtracks not yet tried

if GOAL-TEST(s') **then return** *stop*

if s' is a new state (not in *untried*) **then** *untried* [s'] \leftarrow ACTIONS(s')

if s is not null **then**

result [s,a] \leftarrow s'

add s to the front of *unbacktracked* [s']

if *untried* [s'] is empty **then**

if *unbacktracked* [s'] is empty **then return** *stop*

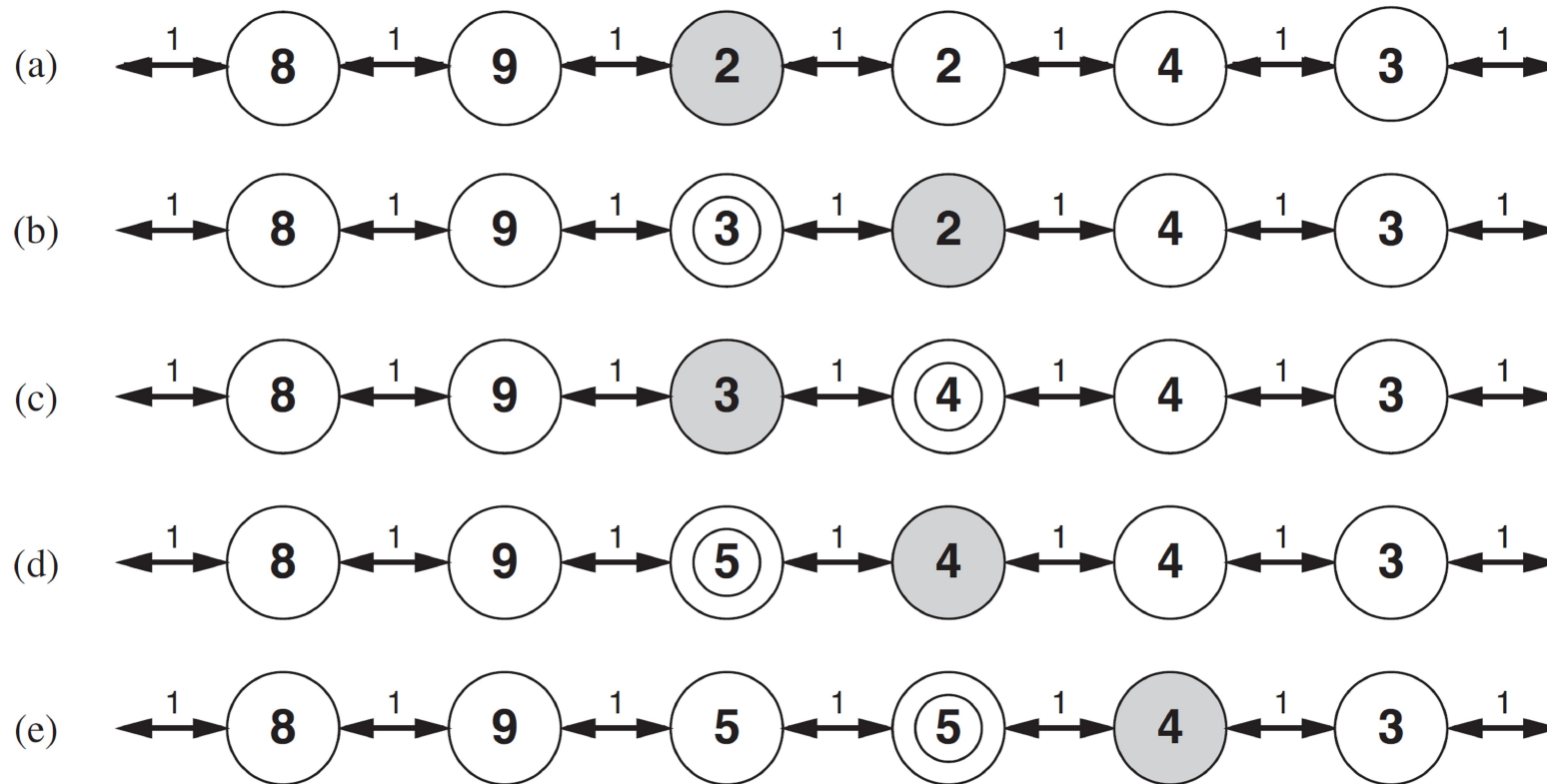
else $a \leftarrow$ an action b such that *result* [s',b] = POP(*unbacktracked* [s'])

else $a \leftarrow$ POP(*untried* [s'])

$s \leftarrow s'$

return a

Busca *online* local e aprendizagem



function LRTA*_AGENT (s') **returns** an action

inputs: s' , a percept that identifies the current state

static: $result$, table indexed by state and action, initially empty

H , a table of costs estimates indexed by state, initially empty

s , a , the previous state and action, initially null

if GOAL-TEST (s') **then return** $stop$

if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$

if s is not null

$result[s,a] \leftarrow s'$

$H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA}^*_\text{COST}(s,b,result[s,b],H)$

$b \in \text{ACTIONS}(s)$

$a \leftarrow$ action b in $\text{ACTIONS}(s')$ that minimizes $\text{LRTA}^*_\text{COST}(s',b,result[s',b],H)$

$s \leftarrow s'$

return a

function LRTA*_COST (s, a, s', H) **returns** a cost estimate

if s' is undefined **then return** $h(s)$

else return $c(s, a, s') + H[s']$

Referências

Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison Wesley, Massachusetts, USA, 1984.

Russell, S., Norvig, P. *Artificial Intelligence-A Modern Approach*, Prentice Hall, New Jersey, USA, 2010.

Luger, G. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Addison Wesley, Massachusetts, USA, 2009.

Nilsson, N. *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, USA, 1998.

Winston, P. *Artificial Intelligence*, 3rd Edition, Addison Wesley, USA, 1993.

Observação

Este material refere-se às notas de aula do curso EA 072 Inteligência Artificial em Aplicações Industriais da Faculdade de Engenharia Elétrica e de Computação da Unicamp. Não substitui o livro texto, as referências recomendadas e nem as aulas expositivas. Este material não pode ser reproduzido sem autorização prévia dos autores. Quando autorizado, seu uso é exclusivo para atividades de ensino e pesquisa em instituições sem fins lucrativos.