



**Engenharia de Computação  
Inteligência Artificial  
04 - Estruturas e Estratégias de Busca**

# Introdução

## Algoritmos de busca

- não informados (depth-first, breadth-first):
  - usam somente definição do problema
- informados (Greedy best-first, A<sup>\*</sup>):
  - usam conhecimento sobre o domínio além do problema
  - conhecimento na forma de heurísticas

## Aplicações

- sistemas baseados em conhecimento
- sequenciamento de produção
- busca internet

## Desempenho e complexidade em algoritmos de busca

- Desempenho
    - **completeza**: garantia de encontrar uma solução, se existir
    - **optimalidade**: estratégia busca encontra solução ótima
    - **complexidade temporal**: tempo para achar uma solução
    - **complexidade espacial**: quantidade de memória para a busca
  - Complexidade
    - **fator de ramificação** ( $b$ ): número máximo de sucessores de um nó
    - **profundidade** ( $d$ ): profundidade do nó meta mais raso
    - **comprimento trajetória** ( $m$ ): maior entre todas as trajetórias

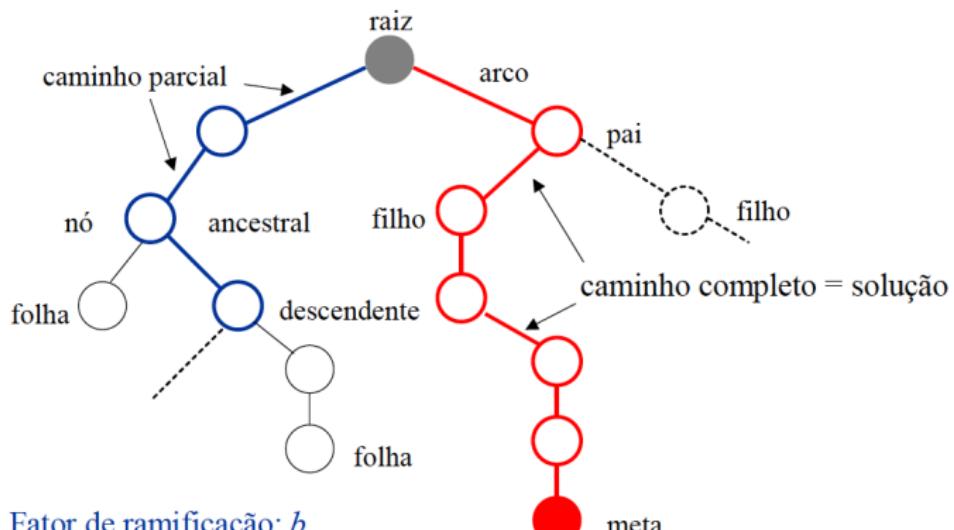
## Algoritmos de busca não informados

- Características
    - busca cega
    - utiliza somente informação contida no problema
    - definidos pela ordem em que os nós são expandidos
    - necessário eliminar ciclos (organizar soluções em uma árvore)
    - detectar estados redundantes
    - complexidade

## Algoritmos de busca não informados

- Componentes de um problema
    - 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 (remove) o elemento mais antigo da fila
    - **LIFO:** pops (remove) o elemento mais novo da fila
    - **PRIORITY:** pops (remove) elemento com maior prioridade

## Árvore de busca



Fator de ramificação:  $b$

Profundidade:  $d$

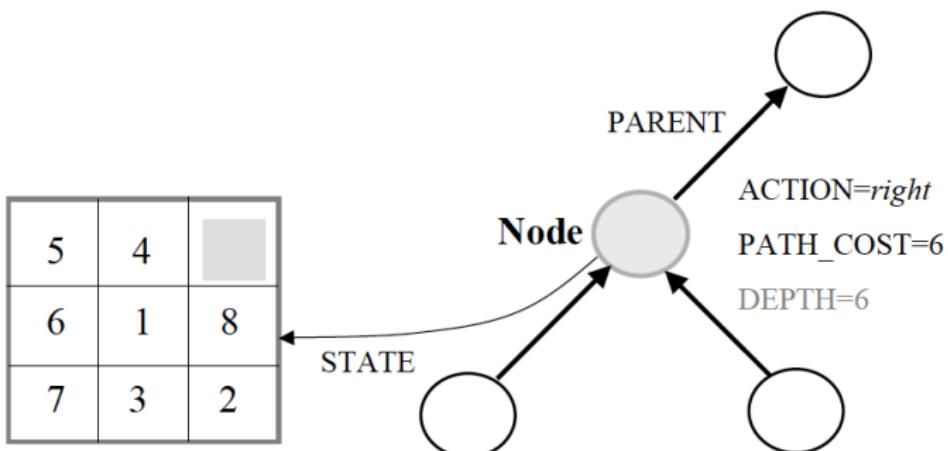
Total caminhos:  $b^d$

## Nós em algoritmos de busca

Estrutura de dados com os seguintes componentes

- **Estado:**  $n.STATE$  - estado correspondente ao nó  $n$
- **Nó pai:**  $n.PARENT$  - nó da árvore que gerou o nó  $n$
- **Ação:**  $n.ACTION$  - ação aplicada ao pai que gerou  $n$
- **Custo:**  $n.PATH-COST = g(n)$  custo do estado inicial até  $n$
- **Profundidade:**  $n.DEPTH$  - número de arcos no caminho da raiz até o nó  $n$
- **Custo do passo:**  $p.STEP-COST = c(s,a,n)$  - custo do passo para problema  $p$
- **Resultado:**  $p.RESULT = RESULT(s,a)$  - modelo de transição sucessor de  $p$

## Nó em algoritmos de busca

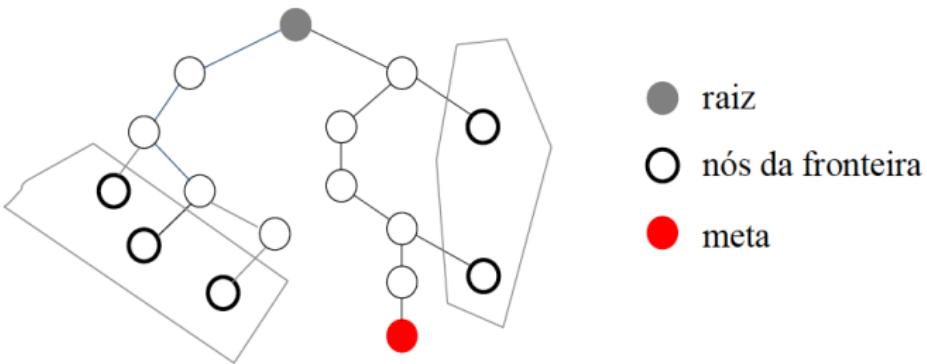


## Geração de filho de um nó

```
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)

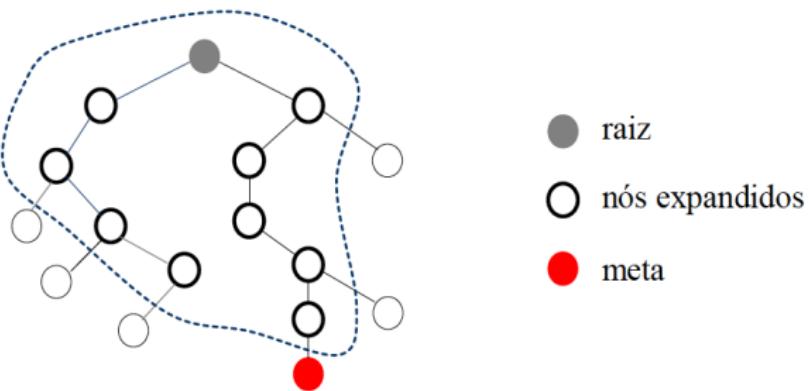


- **FIFO:** pops (remove) o elemento mais antigo da fila
- **LIFO:** pops (remove) o elemento mais novo da fila
- **PRIORITY:** pops (remove) elemento com maior prioridade

## Operações com filas

- $\text{EMPTY?}(queue)$ : retorna true somente se fila é vazia
- $\text{POP}(queue)$ : remove e retorna primeiro elemento da fila
- $\text{INSERT}(element, queue)$ : insere elemento e retorna fila resultante
- $\text{SOLUTION}(n)$ : retorna a sequência de ações de  $n$  até a raiz

## Conjuntos de nós expandidos (explored set)



Conjunto nós expandidos = hash table

Propósito: verificar estados repetidos

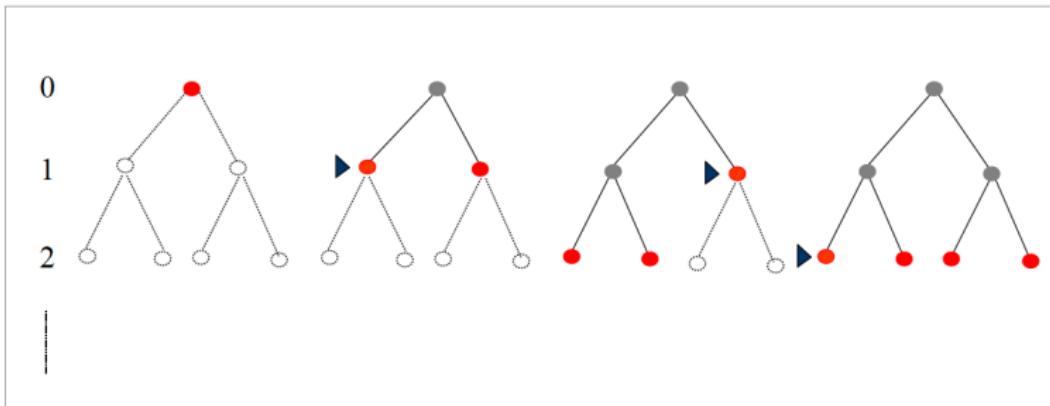
Igualdade de conjuntos:  $\{Bucharest, Vaslui\} = \{Valui, Bucharest\}$

## 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)



## Busca em largura (breadth-first search)

```
function BREADTH_FIRST_SEARCH (problem) returns a solution, or failure
    node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE; PATH-COST = 0
    if problem.GOAL-TEST (node.STATE) then return SOLUTION(node)
    frontier  $\leftarrow$  a FIFO queue with node as the only element
    explored  $\leftarrow$  an empty set

loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP (frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child  $\leftarrow$  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  $\leftarrow$  INSERT (child, frontier)
```

## Complexidade busca em largura

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

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

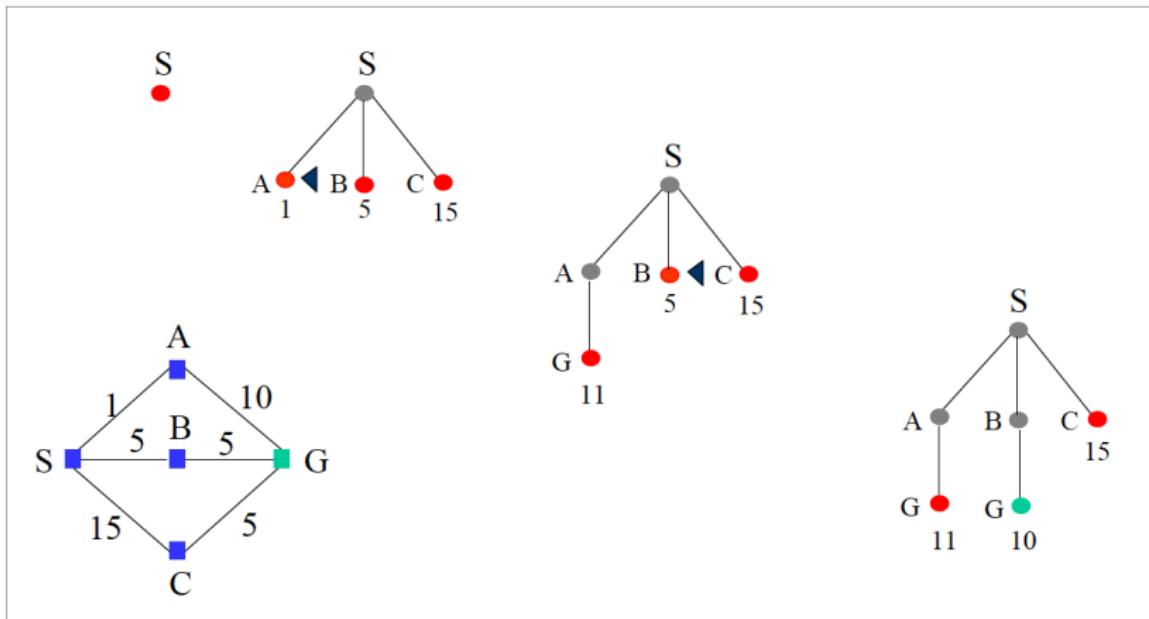
| Profundidade | Nós       | Tempo    | Memória             |
|--------------|-----------|----------|---------------------|
| 4            | 11.110    | 11 ms    | 10.6 MB             |
| 8            | $10^8$    | 2 m      | 103 GB              |
| 12           | $10^{12}$ | 13 dias  | 1 PB( $10^{15}$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

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

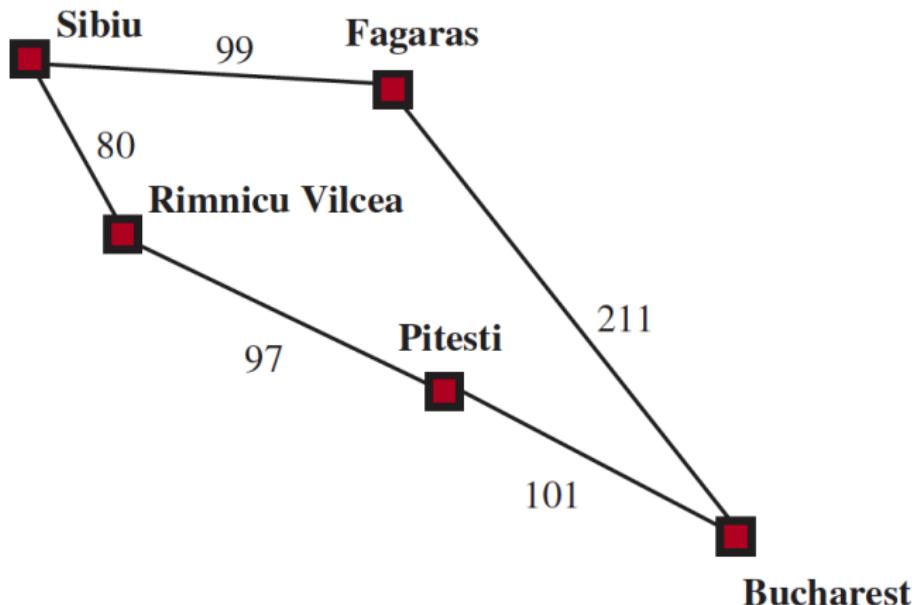
## Busca uniforme (uniform search)

- expande nó com menor  $g(n)$  (priority queue ordenada por g)
  - nó no caminho com menos 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*

## Busca uniforme



## Busca uniforme



## Busca uniforme

**function** UNIFORM\_COST\_SEARCH (*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0  
*frontier*  $\leftarrow$  a priority queue ordered by PATH-COST with *node* as the only element  
*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure  
    *node*  $\leftarrow$  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*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)  
        **if** *child*.STATE is not in *explored* or *frontier* **then**  
            *frontier*  $\leftarrow$  INSERT (*child*, *frontier*)  
        **else if** *child*.STATE is in *frontier* with higher PATH-COST **then**  
            replace that *frontier* node with *child*

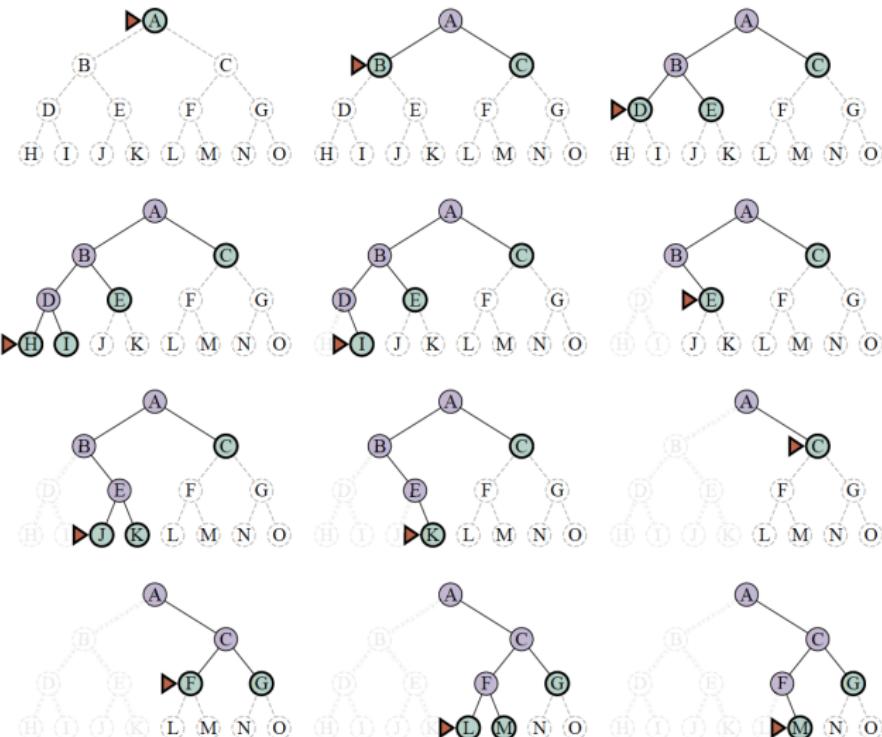
## Complexidade busca uniforme

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

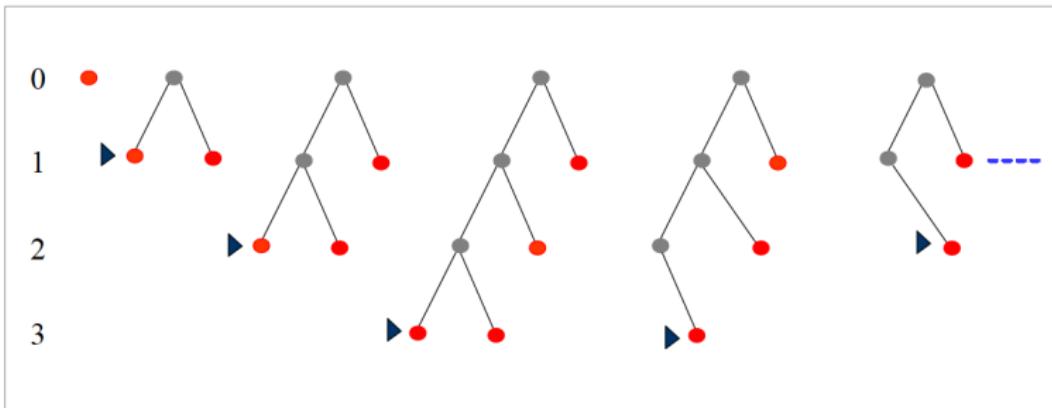
$$O(b^{1+\lfloor C^*/\varepsilon \rfloor}) \geq O(b^d)$$

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

## Busca em profundidade (depth-first search)



## Busca em profundidade (depth-first search)



Exemplo: assume nós com profundidade 3 sem sucessores

## Busca em profundidade (depth-first search)

**function** DEPTH-FIRST-SEARCH (*problem*) **returns** a solution or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE; PATH-COST = 0  
*frontier*  $\leftarrow$  a LIFO queue with *node* as the only element  
*explored*  $\leftarrow$  an empty set

**loop do**

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

*node*  $\leftarrow$  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*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

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

*frontier*  $\leftarrow$  INSERT (*child*, *frontier*)

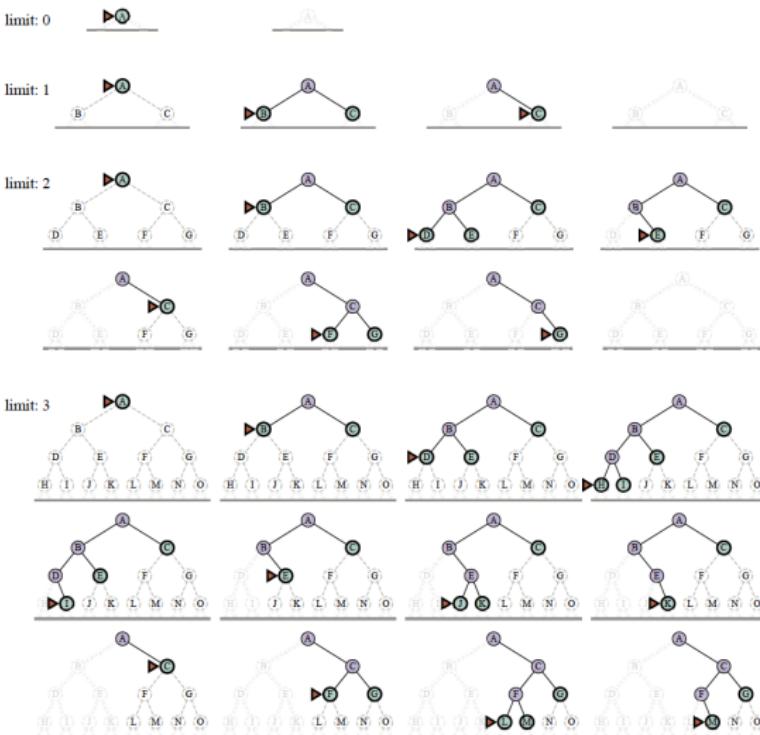
## Complexidade busca em profundidade

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

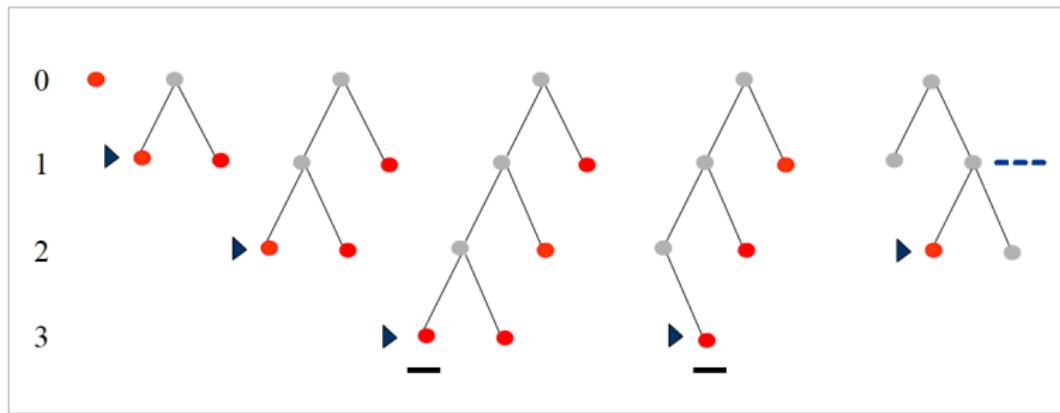
## Busca profundidade limitada (depth-limited search)

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

## Busca profundidade limitada (depth-limited search)



## Busca profundidade limitada (depth-limited search)



Exemplo: assume nós com profundidade 3 ( $\ell = 3$ )

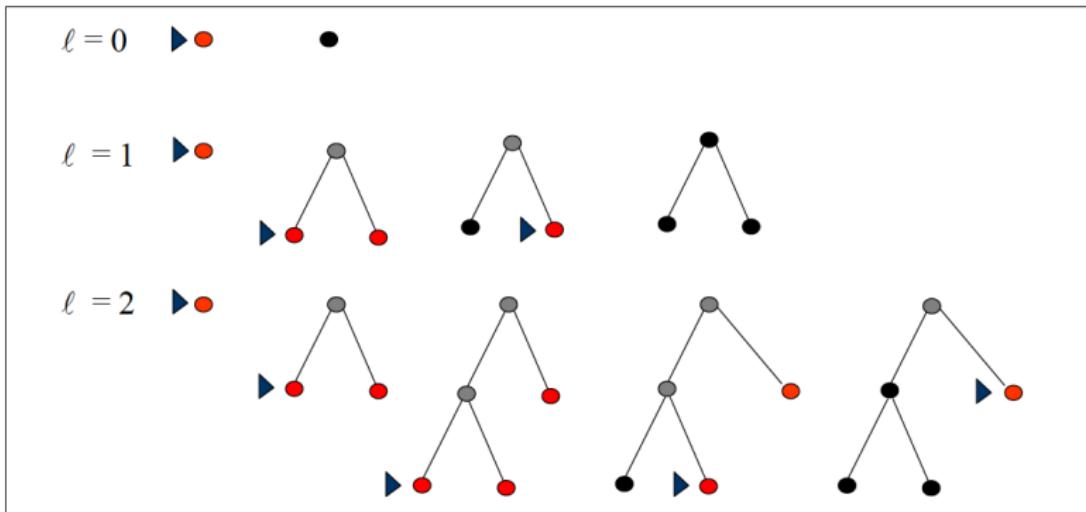
## Algoritmo de busca em profundidade limitada

```
function DEPTH _ LIMITED _ SEARCH (problem, limit) returns a solution, or failure/cutoff  
  
node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE; PATH-COST = 0  
frontier  $\leftarrow$  a LIFO queue with node as the only element  
explored  $\leftarrow$  an empty set  
  
loop do  
    if EMPTY?(frontier) then return failure  
    node  $\leftarrow$  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  $\leftarrow$  CHILD-NODE(problem, node, action)  
        if child.STATE is not in explored or frontier then  
            if DEPTH(node) = limit then return cutoff  
            frontier  $\leftarrow$  INSERT (child, frontier)
```

## Busca profundidade progressiva (interative-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:  $O(bd)$

## Busca profundidade progressiva (interative-deepening search)



## Busca profundidade progressiva (iterative-deepening search)

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff
  frontier  $\leftarrow$  a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result  $\leftarrow$  failure
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node)  $>$   $\ell$  then
      result  $\leftarrow$  cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

## Busca profundidade progressiva (interative-deepening search)

- $N(IDS) = (d)b + (d - 1)b^2 + \dots + b^d$
- $N(BFS) = b + b^2 + \dots + b^d$
- $b = 10, d = 5 \rightarrow N(IDS) = 123.450 \quad N(BFS) = 111.110$
- $N(IDS)$  não é muito maior que  $N(BFS)$ !
- IDS: método de escolha quando espaço busca é grande profundidade da solução não é conhecida a priori
- Abordagem híbrida: busca em largura e busca em profundidade progressiva

## 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

$\ell$  : limite da profundidade

$C^*$ : custo da solução ótima

$\varepsilon$  : menor custo de uma ação

## Algoritmos de busca informados

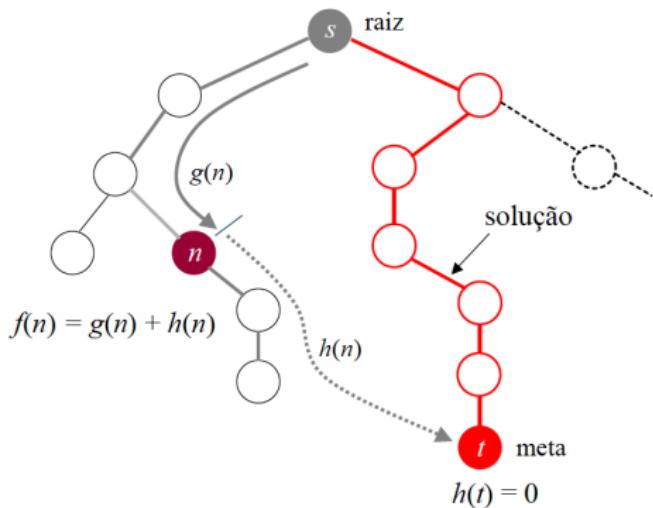
### Características

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

### Algoritmos do tipo best-first

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

## Árvore de busca



$n$ : 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

## Best First Search - melhor escolha

```
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
```

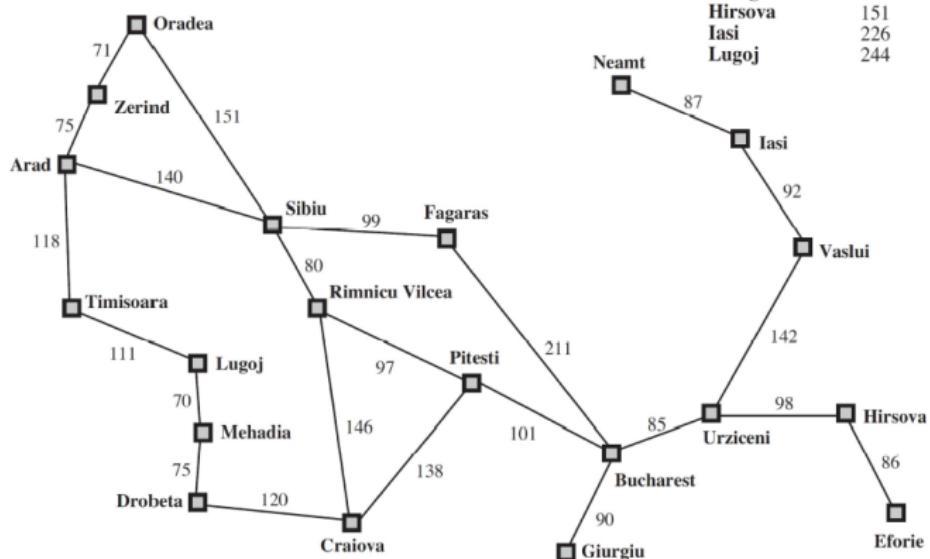
## Greedy best-first search - Busca gulosa

```
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
```

## Greedy best-first search

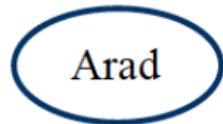
### Exemplo



|           |     |                |     |
|-----------|-----|----------------|-----|
| 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 |

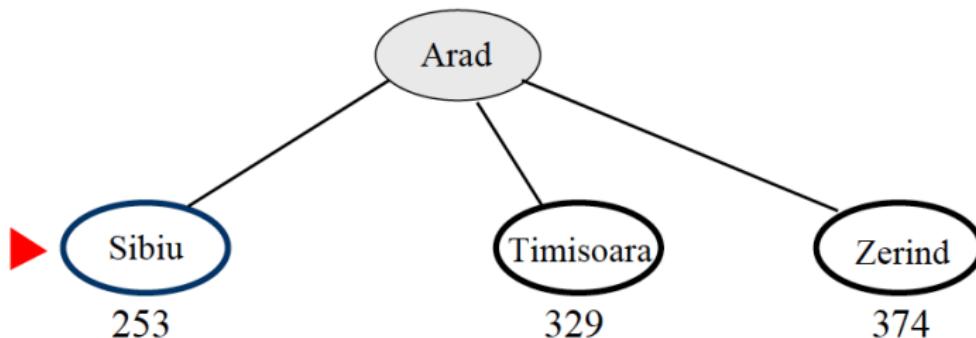
## Greedy best-first search

Estado inicial  $In(Arad)$

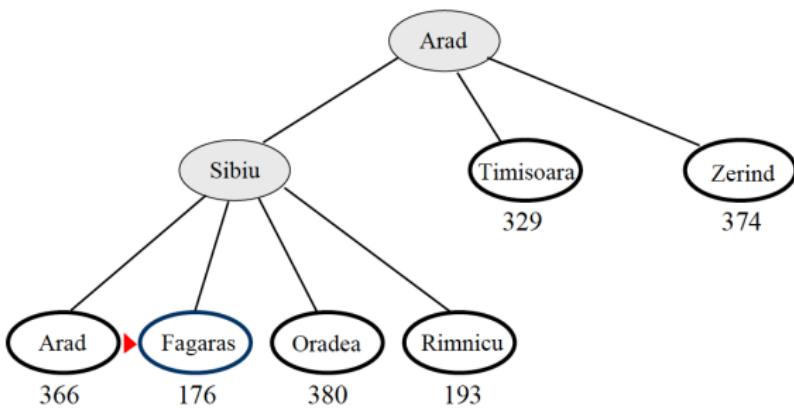


366

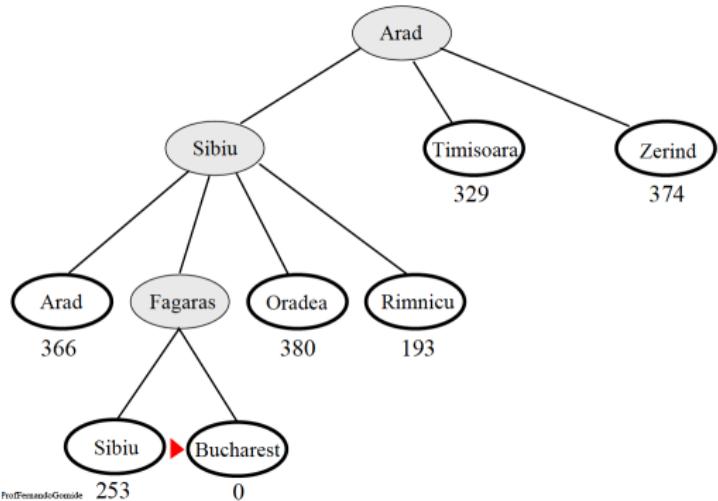
## Greedy best-first search



## Greedy best-first search



## Greedy best-first search



## Greedy best-first search

- baixo custo de busca
- não é ótimo
  - Arad-Sibiu-Fagaras-Bacharest=450
  - Arad-Sibiu-Rimnicu Vilcea-Pitesti-Bucharest=418
- versão árvore: incompleto (mesmo em espaço estado finito)
  - caminho de lasi 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
```

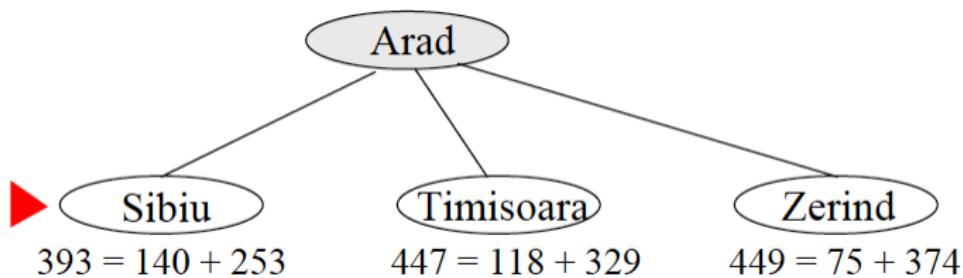
$$f(n) = g(n) + h(n)$$

## Busca A\*

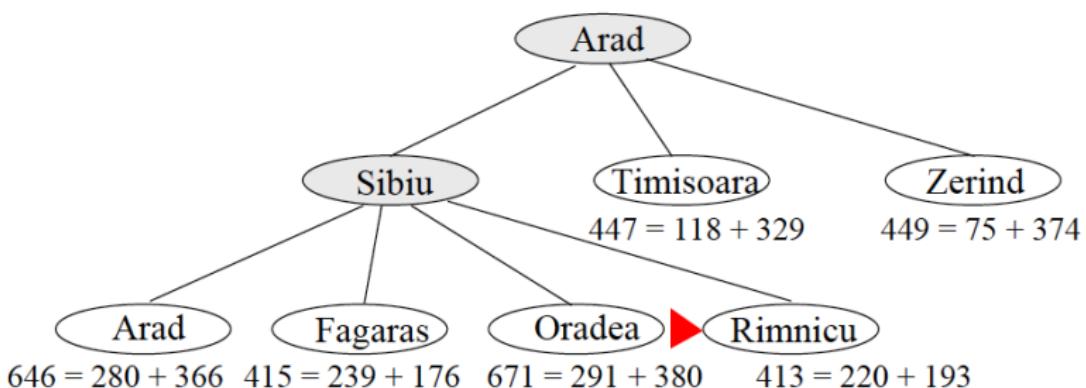
Estado inicial *In* (*Arad*)   Arad

$$366 = 0 + 366$$

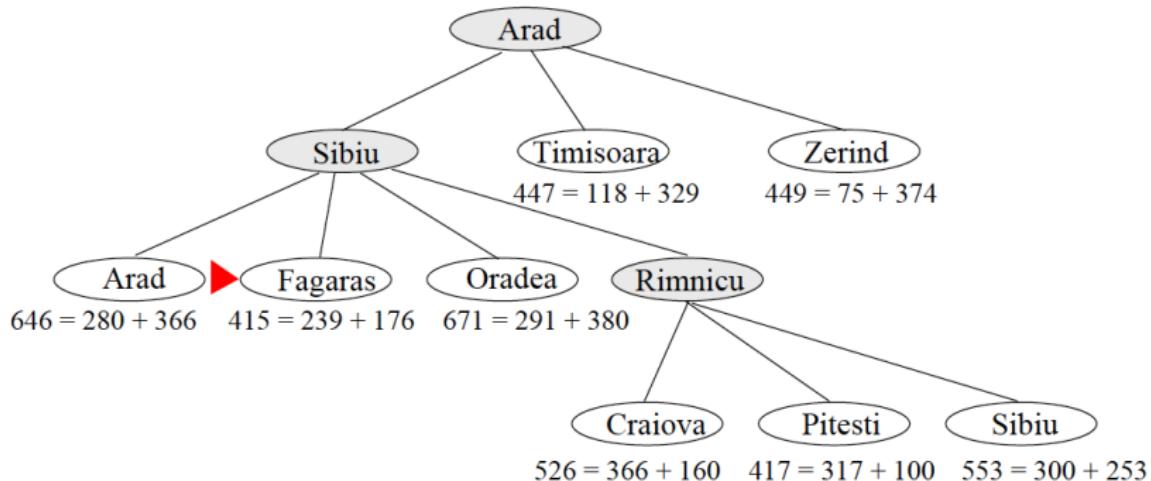
## Busca A\*



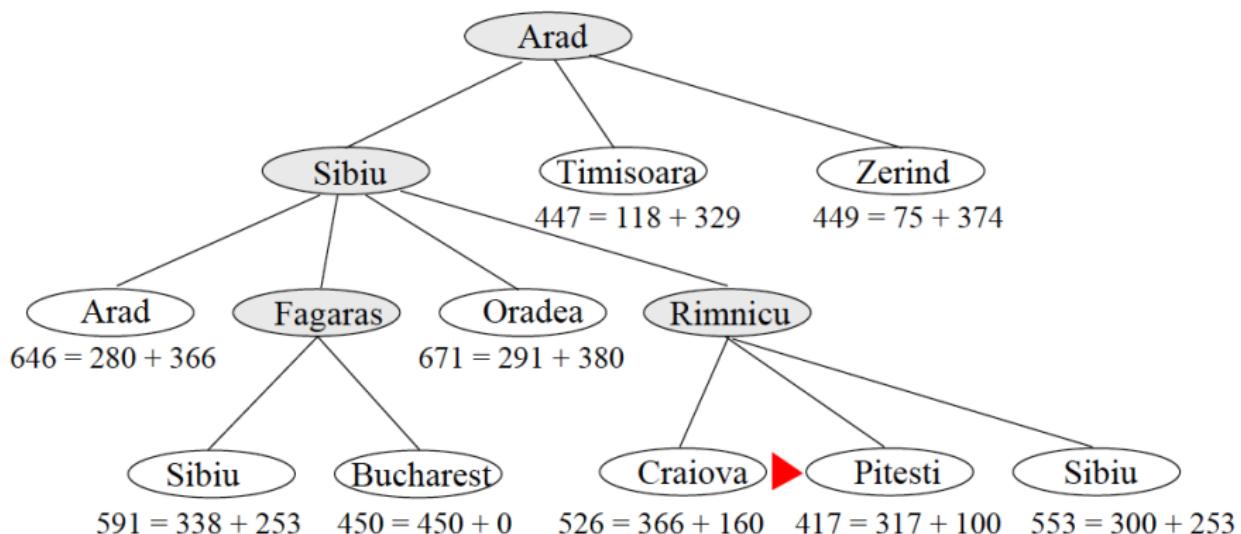
## Busca A\*



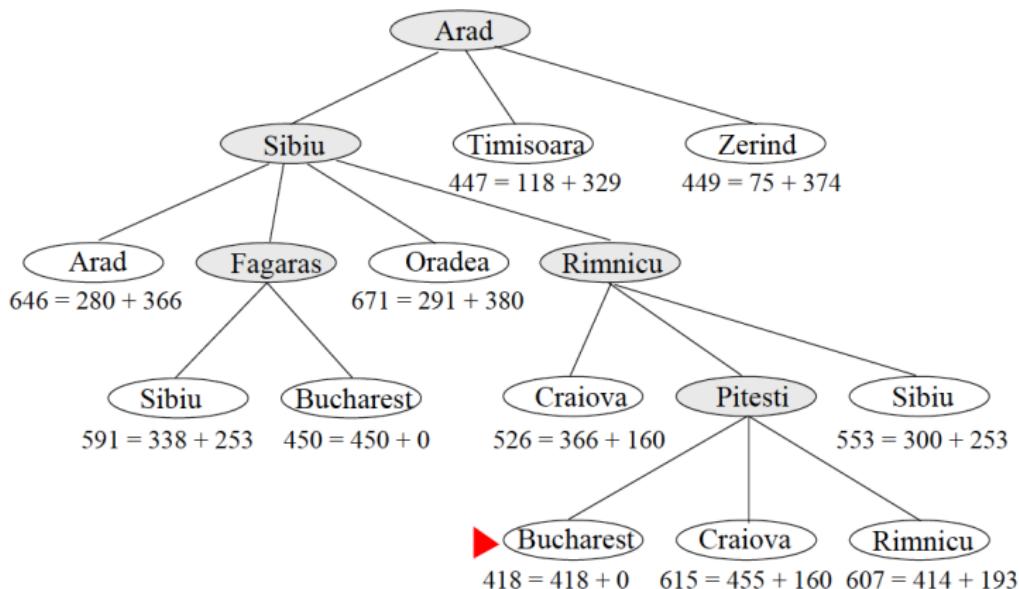
## Busca A\*



## Busca A\*



## Busca A\*



## Busca A\*

- 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

## 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

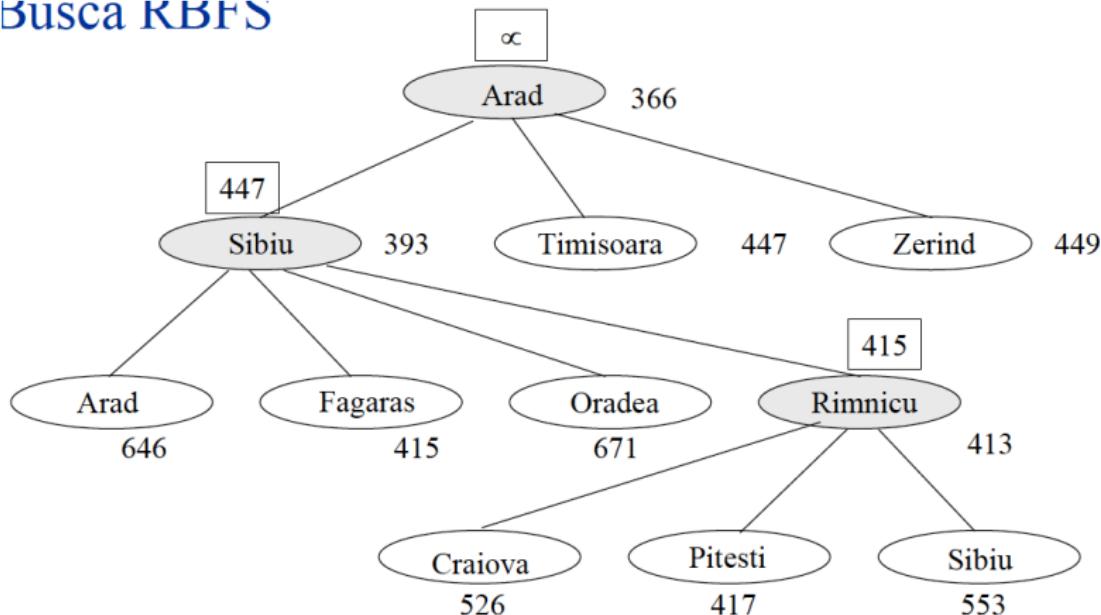
## Algoritmo de busca RBFS

```
function RECURSIVE_BEST_FIRST_SEARCH (problem) returns a solution, or failure
    return RBFS (problem, MAKE-NODE(problem.INITIAL-STATE,  $\infty$ ))

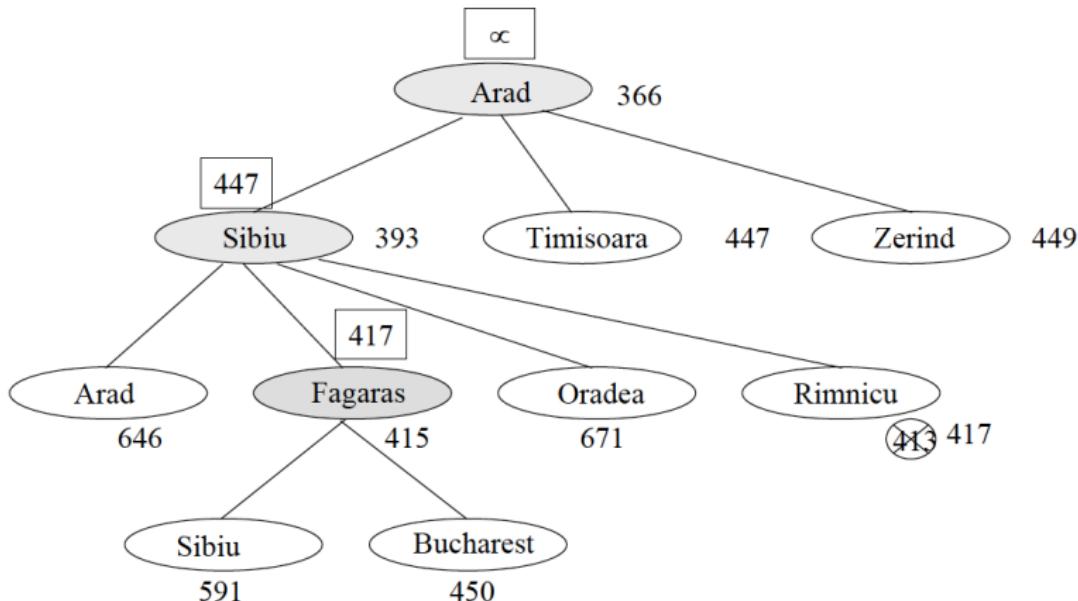
function RBFS (problem, node, f_limit) returns a solution, or failure and new f-cost limit
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors  $\leftarrow$  []
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure,  $\infty$ 
    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

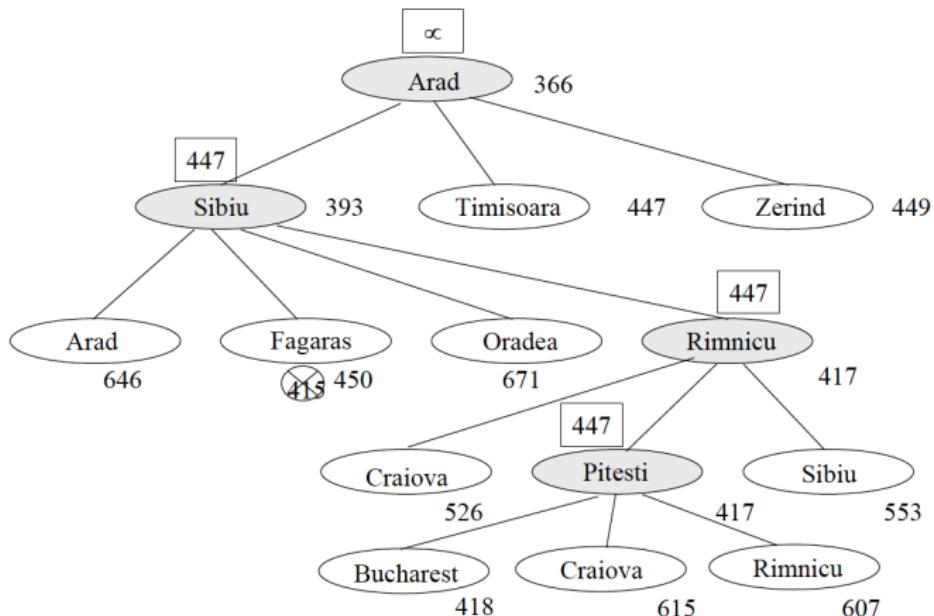
## Busca KBFS



## Busca RBFS



## Busca RBFS



## Funções heurísticas

|   |   |   |
|---|---|---|
| 7 | 2 | 4 |
| 5 | ● | 6 |
| 8 | 3 | 1 |

estado inicial

|   |   |   |
|---|---|---|
| ● | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

meta

- 8 peças  $\rightarrow 9!/2 = 181.440$  estados distintos atingíveis
- 15 peças  $\rightarrow 16!/2 =$  mais de 10 trilhões estados distintos atingíveis

## Funções heurísticas

|   |   |   |
|---|---|---|
| 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

## Comparação IDS x A\* com heurísticas $h_1$ e $h_2$

|     | Número médio de nós gerados |            |            |
|-----|-----------------------------|------------|------------|
| $d$ | 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  $d$

## Comparação BFS x A\* com heurísticas $h_1$ e $h_2$

| $d$ | Search Cost (nodes generated) |             |             | Effective Branching Factor |             |             |
|-----|-------------------------------|-------------|-------------|----------------------------|-------------|-------------|
|     | BFS                           | A*( $h_1$ ) | A*( $h_2$ ) | BFS                        | A*( $h_1$ ) | A*( $h_2$ ) |
| 6   | 128                           | 24          | 19          | 2.01                       | 1.42        | 1.34        |
| 8   | 368                           | 48          | 31          | 1.91                       | 1.40        | 1.30        |
| 10  | 1033                          | 116         | 48          | 1.85                       | 1.43        | 1.27        |
| 12  | 2672                          | 279         | 84          | 1.80                       | 1.45        | 1.28        |
| 14  | 6783                          | 678         | 174         | 1.77                       | 1.47        | 1.31        |
| 16  | 17270                         | 1683        | 364         | 1.74                       | 1.48        | 1.32        |
| 18  | 41558                         | 4102        | 751         | 1.72                       | 1.49        | 1.34        |
| 20  | 91493                         | 9905        | 1318        | 1.69                       | 1.50        | 1.34        |
| 22  | 175921                        | 22955       | 2548        | 1.66                       | 1.50        | 1.34        |
| 24  | 290082                        | 53039       | 5733        | 1.62                       | 1.50        | 1.36        |
| 26  | 395355                        | 110372      | 10080       | 1.58                       | 1.50        | 1.35        |
| 28  | 463234                        | 202565      | 22055       | 1.53                       | 1.49        | 1.36        |

## Geração de heurísticas - Relaxação

Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $X$  for adjacente a  $Y$  e  $Y$  estiver vazio.

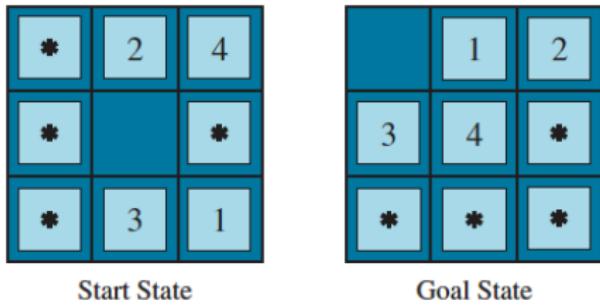
- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $X$  for adjacente a  $Y$
- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $Y$  estiver vazio.
- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$ .

## Geração de heurísticas - Relaxação

Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $X$  for adjacente a  $Y$  e  $Y$  estiver vazio.

- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $X$  for adjacente a  $Y \rightarrow h_2$  (distância de Manhattan)
- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y$  se  $Y$  estiver vazio.
- Uma peça pode se mover do quadrado  $X$  para o quadrado  $Y \rightarrow h_1$

## Geração de heurísticas - Pattern databases



## Geração de heurísticas - Pontos de referência

7 h 17 min (578 km)

via A1 and E81

Fastest route, the usual traffic:

Arad

Romania

> Get on A1 from Strada Andrei Saguna, Strada Dorobentii and Calea Bodogului/D1682F  
8 min (4.3 km)

> Get on DN1B/A/E673 in Județul Timiș from A1  
53 min (101 km)

> Get on A1 in Șoimug from DN1B/A/E673 and DN7/E88  
1 h 13 min (27.7 km)

> Follow A1, DN1/DN7/E88/E81 and A1 to DN1/DN7/E88/E81 in Județul Sibiu. Take the DN7/DN1 exit from A1  
1 h 13 min (21.1 km)

> Continue to Râmești Vâlcea

1 h 21 min (36.4 km)

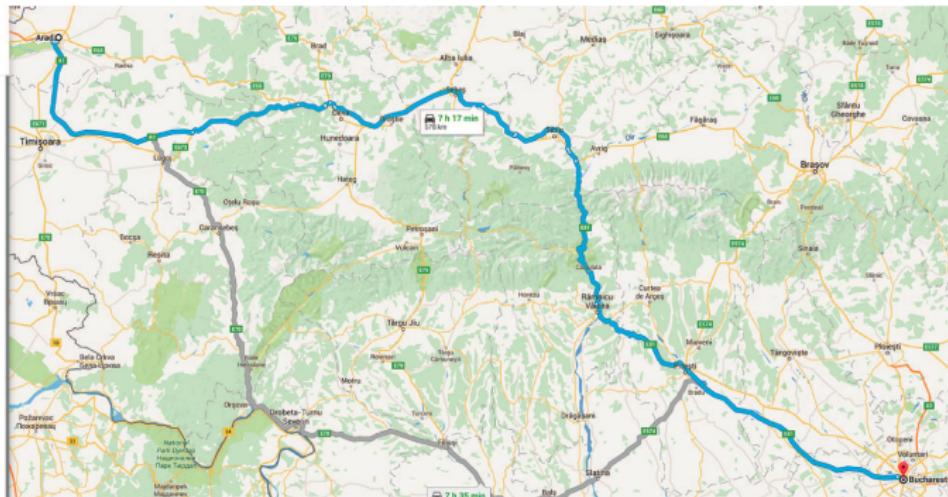
† Continue onto DN7/E81  
① Continue to follow E81

2 h 15 min (177 km)

> Take Spăleră Independenței to Bulevardul Unirii/E81  
9 min (3.0 km)

Bucharest

Romania



- Calcular previamente alguns custos de caminho ótimos.
- Espaço =  $O(|V|^2)$  e Tempo =  $O(|E|^3)$
- Escolha de pontos de referência

## Geração de heurísticas - Experiência e Aprendizagem

- Evitar explorar subárvores pouco promissoras.
- Minimizar o custo total da solução de problemas.
- Técnicas de aprendizado de máquina.