CEFET-MG

Engenharia de Computação
Inteligência Artificial
04 - Estruturas e Estratégias de Busca
Parte I - Busca Não Informada e Informada

Organização da aula

- Retomada: formulação de problemas, árvore de busca e fronteira.
- Critérios para comparar estratégias: completude, otimalidade, tempo e memória.
- Busca não informada: largura, custo uniforme, profundidade, profundidade limitada e aprofundamento iterativo.
- Busca informada: função heurística, busca gulosa e A*.
- Qualidade de heurísticas e extensões com limitação de memória.

Fio condutor

Dada uma fronteira com vários caminhos possíveis, qual nó deve ser expandido primeiro?

Retomada da Aula 3

Retomando

Na aula anterior, formulamos problemas de busca por meio de estados, ações, transições, teste de meta e custo.

- O **espaço de estados** representa as configurações possíveis do problema.
- A **árvore de busca** representa os caminhos explorados pelo algoritmo.
- A **fronteira** contém nós gerados e ainda não expandidos.
- A estratégia de busca define a ordem em que esses nós serão selecionados.

Pergunta

Se a formulação do problema é a mesma, por que duas estratégias de busca podem ter desempenhos tão diferentes?

A fronteira define a estratégia

- Em um algoritmo geral de busca, o mecanismo de expansão pode permanecer praticamente o mesmo.
- O que muda é o critério usado para escolher o próximo nó da fronteira.

Estratégia	**Critério principal**
Busca em largura	menor profundidade
Custo uniforme	menor $g(n)$
Busca em profundidade	nó mais recentemente gerado
Busca gulosa	menor $h(n)$
A*	menor $g(n) + h(n)$

Observação

Estudar busca é, em grande parte, estudar como diferentes formas de ordenar a fronteira alteram custo, memória e qualidade da solução.

Como comparar algoritmos de busca?

- **Completude:** o algoritmo garante encontrar uma solução quando alguma solução existe?
- **Otimalidade:** a solução encontrada tem o menor custo segundo a função definida?
- **Complexidade temporal:** quantos nós precisam ser gerados ou expandidos?
- **Complexidade espacial:** quantos nós precisam ser mantidos em memória?

Pergunta

Um algoritmo que encontra uma solução rapidamente é necessariamente melhor?

Parâmetros usados na análise

- b: **fator de ramificação** — número máximo de sucessores de um nó.
- d: profundidade da solução mais rasa.
- m: profundidade máxima do espaço de busca, quando existe.
- C^*: custo da solução ótima.
- ε: menor custo positivo de um passo, quando necessário para a análise.

Observação

Pequenos aumentos em b ou d podem provocar crescimento exponencial no número de nós explorados.

Busca não informada e busca informada

- **Busca não informada:**
  - usa a formulação do problema, sem estimativa adicional de proximidade da meta;
  - exemplos: largura, custo uniforme, profundidade e aprofundamento iterativo.

- **Busca informada:**
  - usa conhecimento adicional sobre o domínio;
  - esse conhecimento é normalmente expresso por uma função heurística $h(n)$;
  - exemplos: busca gulosa e A*.

Pergunta

Que tipo de conhecimento poderia ajudar um algoritmo a priorizar um caminho antes de explorá-lo completamente?

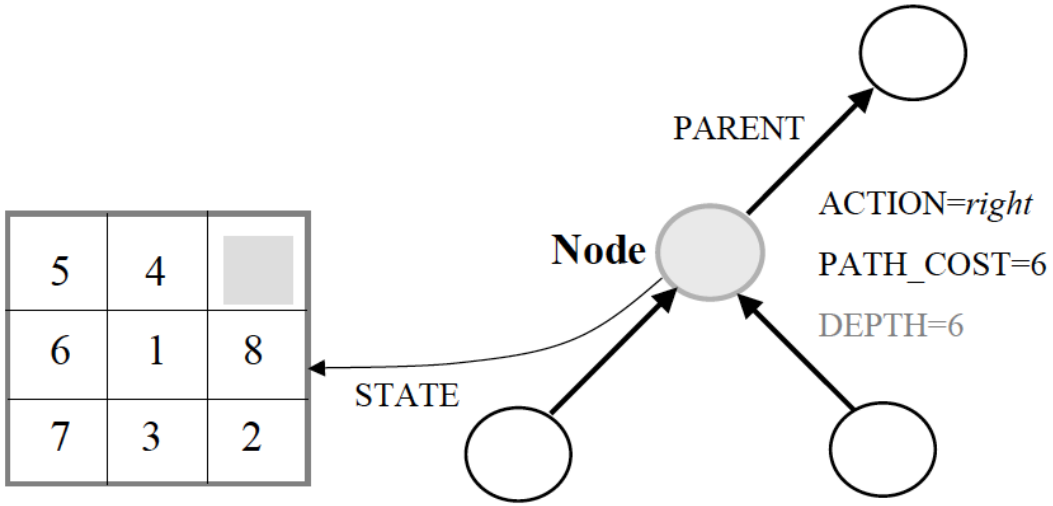
Algoritmo geral de busca em grafo

```
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)
```

Observação

A fronteira e o conjunto explorado evitam que a busca trate repetidamente os mesmos estados sem necessidade.

Nós, caminhos e custo



- Um nó armazena o estado, o nó pai, a ação usada e o custo do caminho.
- $g(n)$ representa o custo acumulado do estado inicial até o nó n.
- Nós diferentes podem representar o mesmo estado alcançado por caminhos distintos.

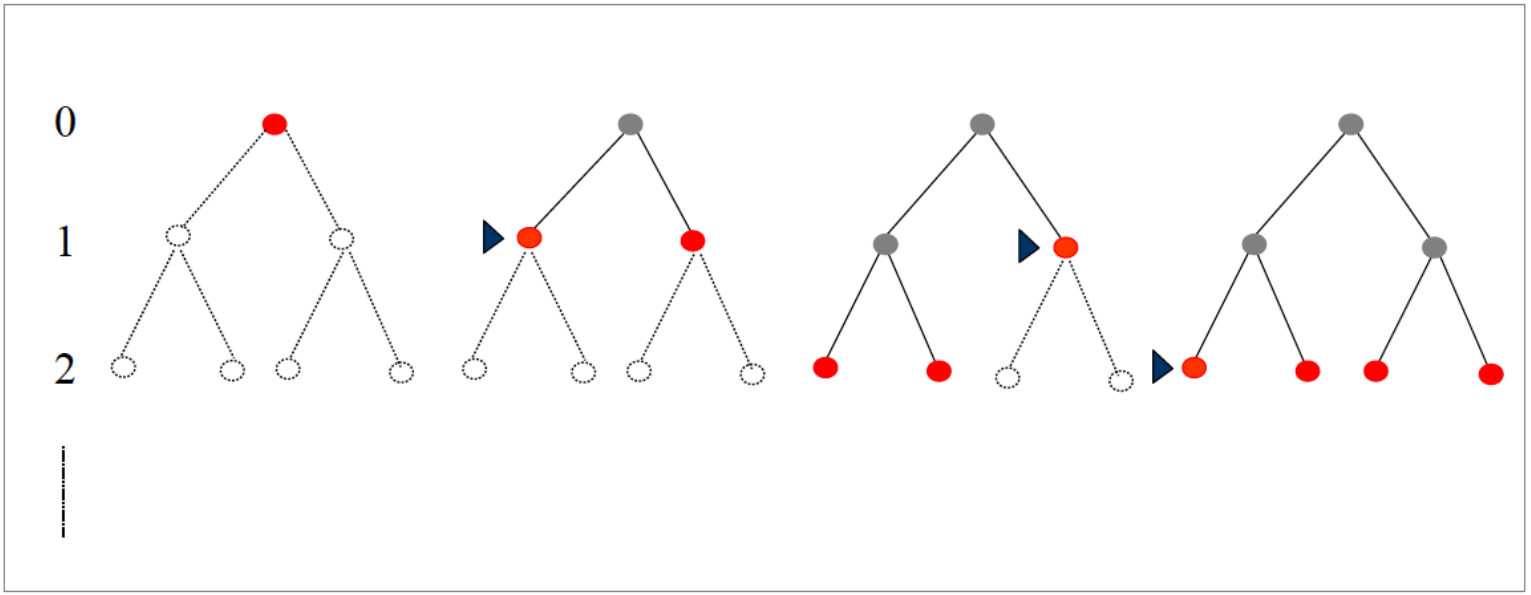
Busca em largura — Breadth-First Search

- Expande primeiro os nós de menor profundidade.
- A fronteira é organizada como uma fila **FIFO**.
- Todos os nós de um nível são considerados antes de avançar para o próximo nível.
- É adequada quando esperamos uma solução rasa e os custos de passo são iguais.

Pergunta

Se cada ação tiver um custo diferente, encontrar a solução mais rasa ainda significa encontrar a solução mais barata?

Busca em largura: ordem de expansão



Observação

A busca cresce por camadas. O custo de memória aumenta porque grande parte da fronteira precisa ser preservada.

Algoritmo de busca em largura

```
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)
```

Busca em largura: propriedades

- **Completa:** sim, se b for finito.
- **Ótima:** sim quando os custos dos passos são iguais; de forma mais geral, quando o custo não diminui com a profundidade.
- **Tempo:** $O(b^d)$.
- **Memória:** $O(b^d)$.

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})

Discussão: o custo da busca em largura

Discussão

Considere $b = 10$ e uma solução na profundidade $d = 16$. Mesmo que gerar cada nó seja muito rápido, a quantidade de nós na fronteira pode tornar o problema inviável em memória.

- A completude da estratégia não significa que ela seja viável na prática.
- Tempo e memória são parte da decisão sobre qual estratégia usar.

Pergunta

Que estratégia poderia reduzir o uso de memória se aceitarmos explorar caminhos mais profundos primeiro?

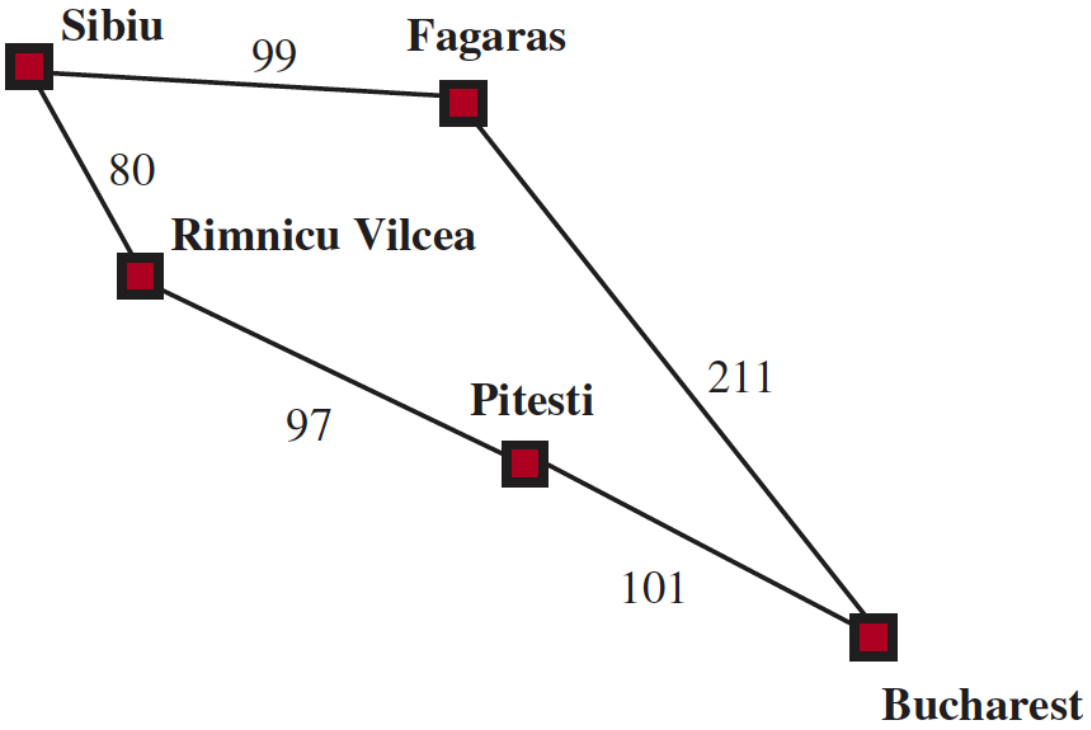
Busca de custo uniforme — Uniform-Cost Search

- Expande o nó da fronteira com menor custo acumulado $g(n)$.
- A fronteira é uma fila de prioridade ordenada por $g(n)$.
- O teste de meta é aplicado quando o nó é removido da fronteira para expansão.
- Um mesmo estado pode ser alcançado por caminhos com custos diferentes; o caminho de menor custo deve prevalecer.

Observação

Quando todos os custos de passo são iguais, a busca de custo uniforme se comporta de forma próxima à busca em largura.

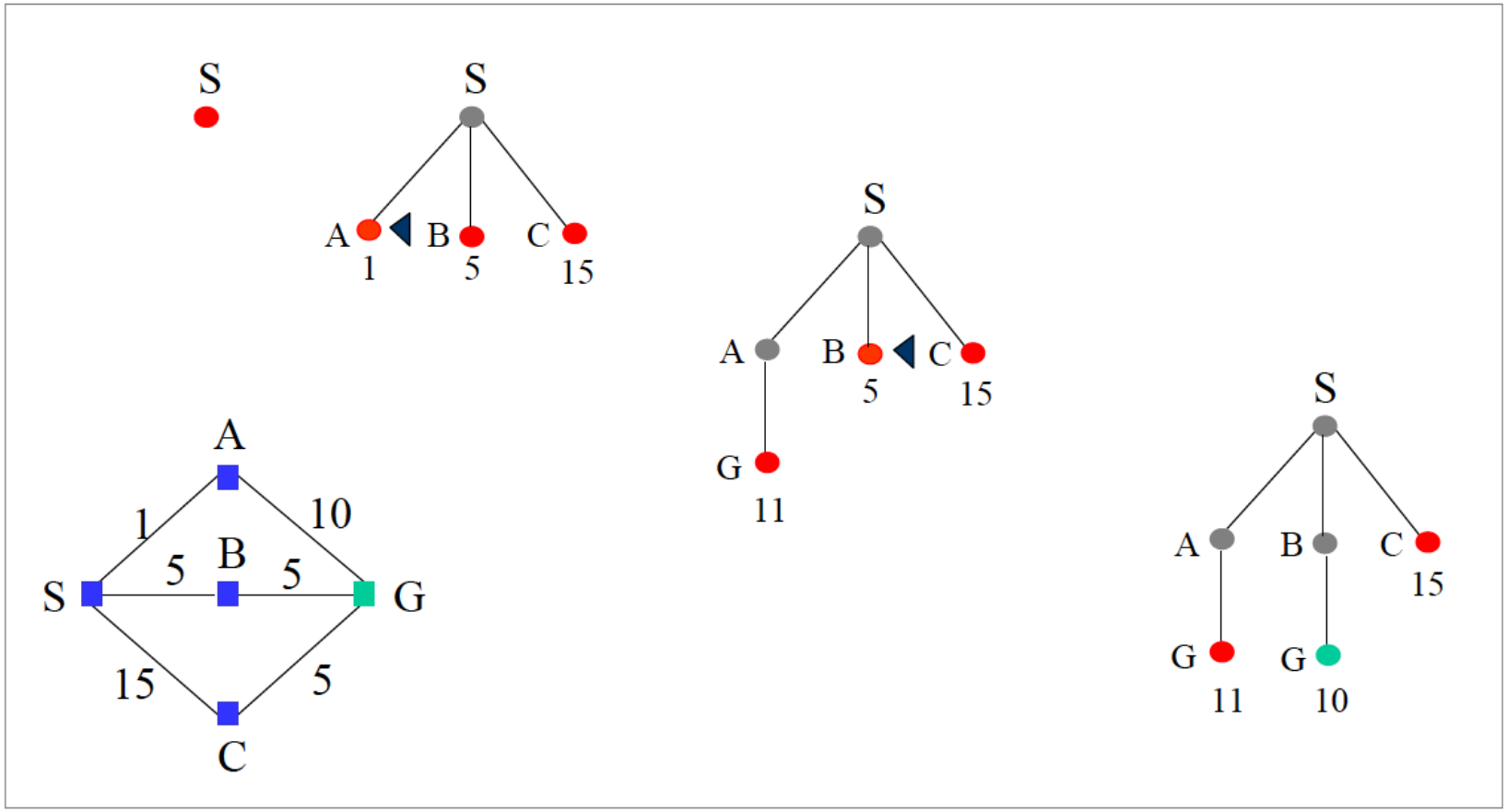
Busca de custo uniforme: exemplo



Pergunta

Por que Fagaras não deve ser escolhido apenas por parecer geograficamente mais próximo de Bucareste?

Busca de custo uniforme: expansão



Algoritmo de busca de custo 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
```

Busca de custo uniforme: propriedades

- **Completa:** sim, se cada passo tiver custo pelo menos $\varepsilon > 0$.
- **Ótima:** sim, pois os caminhos são expandidos em ordem crescente de custo.
- **Tempo e memória:**
$$O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right).$$
- Pode expandir muitos nós com custo menor do que o custo da solução ótima, mesmo que estejam em direções pouco promissoras.

Largura ou custo uniforme?

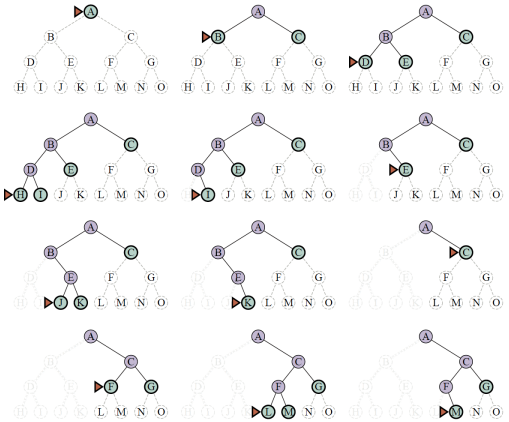
	Busca em largura	**Custo uniforme**
Prioridade	profundidade	$g(n)$
Estrutura	FIFO	prioridade
Ótima	custos iguais	custos positivos
Principal risco	memória	tempo e memória

Pergunta

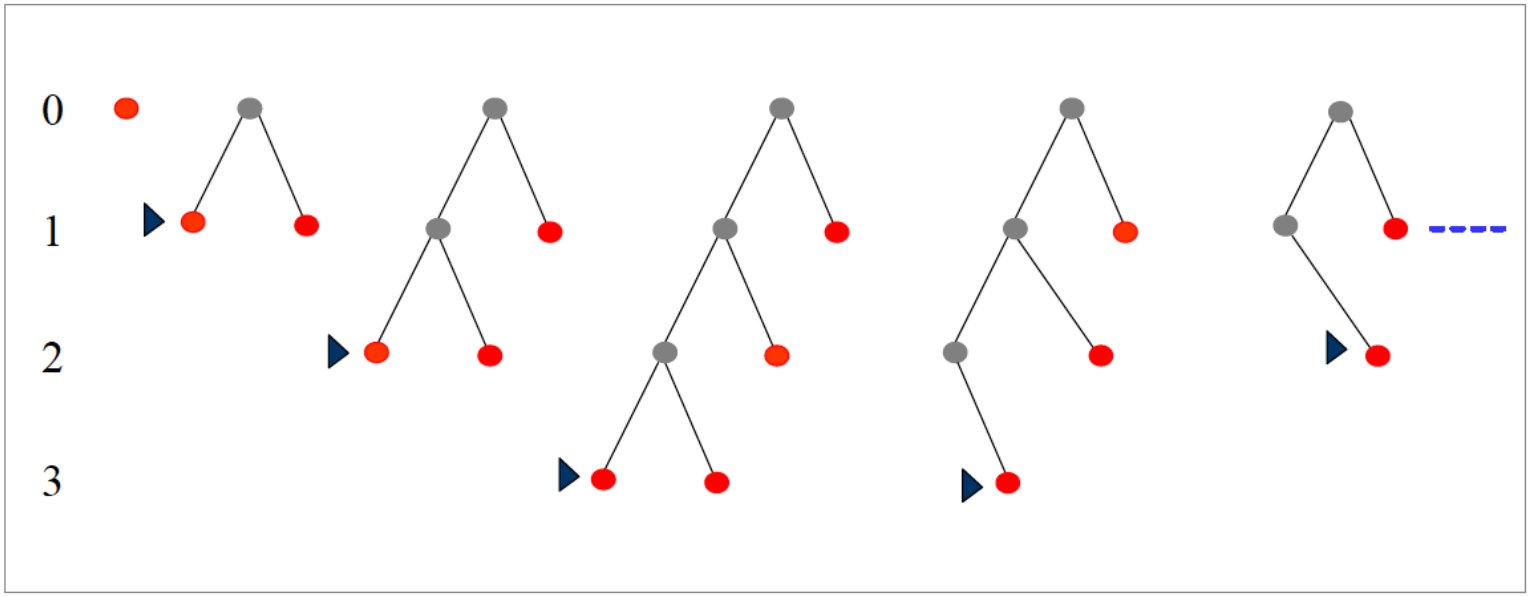
Em um mapa rodoviário, qual estratégia você escolheria se cada estrada tivesse um tempo de viagem diferente?

Busca em profundidade — Depth-First Search

- Expande primeiro o nó mais profundo disponível.
- A fronteira é organizada como uma pilha **LIFO**.
- Mantém poucos caminhos alternativos simultaneamente.
- Pode avançar por um ramo muito longo antes de explorar alternativas rasas.



Busca em profundidade: ordem de expansão



Observação

A vantagem de memória vem acompanhada do risco de seguir longamente um ramo ruim.

Algoritmo de busca em profundidade

```
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)
```

Busca em profundidade: propriedades

- **Completa:** não em busca em árvore com profundidade ilimitada; em grafo finito, pode ser completa com controle de repetição.
- **Ótima:** não.
- **Tempo:** $O(b^m)$ em busca em árvore.
- **Memória:** $O(bm)$ em busca em árvore.

Observação

DFS troca garantias de qualidade por um uso de memória muito menor que BFS em espaços profundos.

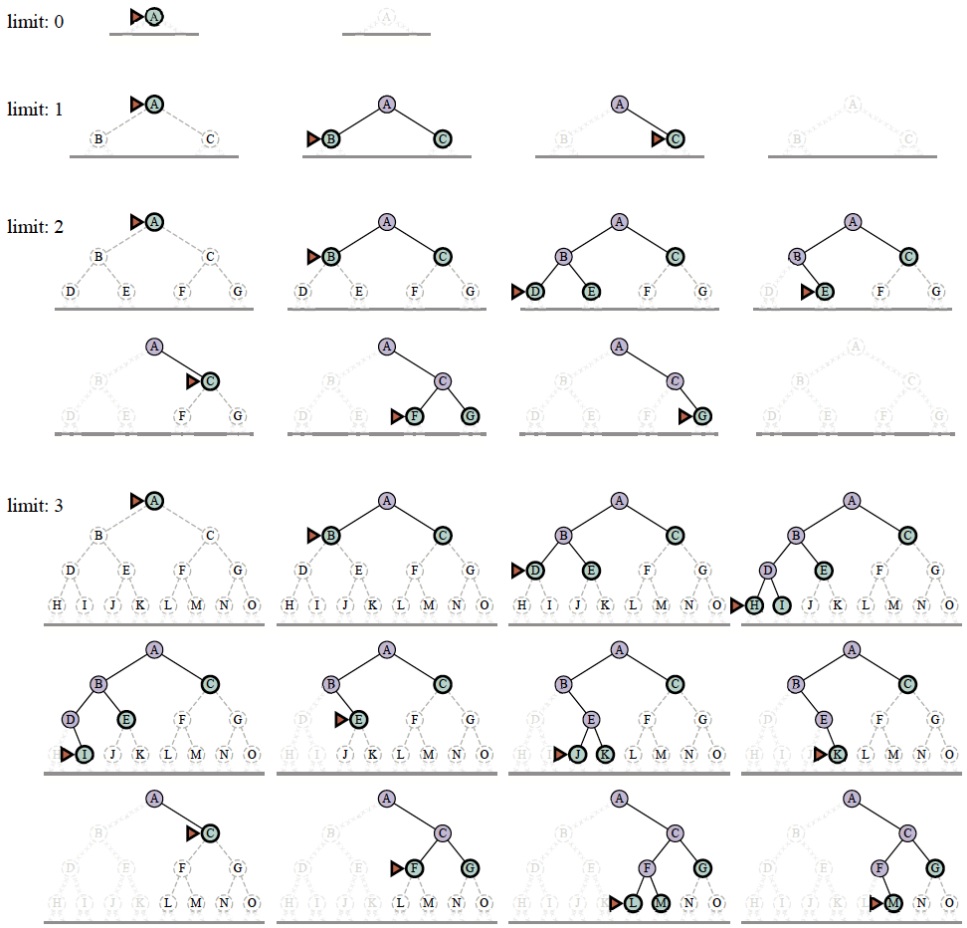
Busca em profundidade limitada

- Introduz um limite de profundidade ℓ para evitar caminhos infinitos ou excessivamente longos.
- Se $\ell < d$, uma solução existente pode ficar fora do limite.
- Se $\ell > d$, a estratégia continua sem garantia de otimalidade.
- **Tempo:** $O(b^\ell)$.
- **Memória:** $O(b\ell)$.

Pergunta

Como escolher ℓ quando não sabemos a profundidade da solução?

Profundidade limitada: visualização



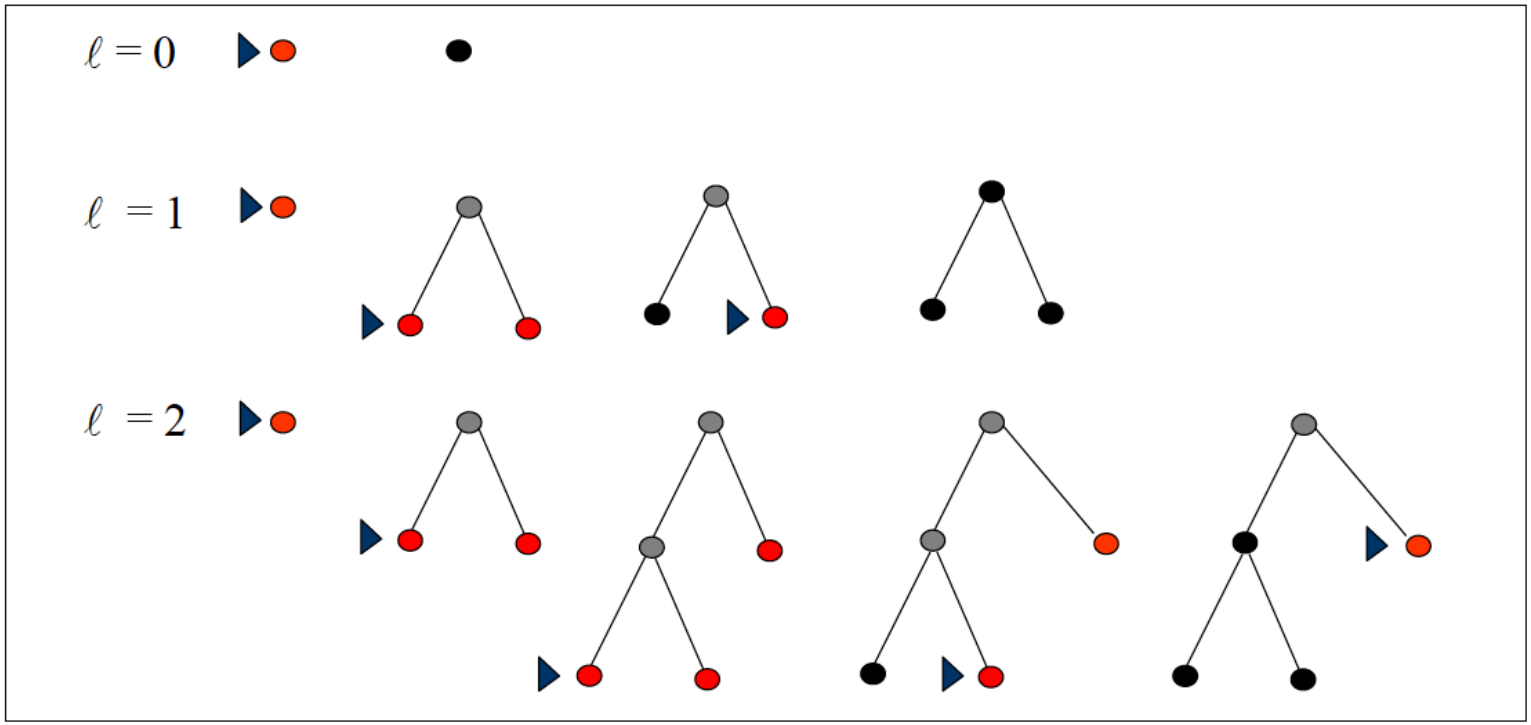
Algoritmo de profundidade limitada

```
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)
```

Aprofundamento iterativo — Iterative Deepening

- Executa buscas em profundidade limitada com limites crescentes: $0, 1, 2, \ldots$.
- Combina a baixa memória de DFS com a busca sistemática por profundidade de BFS.
- **Completa:** sim, se b for finito.
- **Ótima:** sim para custos uniformes de passo.
- **Memória:** $O(bd)$.

Aprofundamento iterativo: limites sucessivos



Algoritmo de aprofundamento iterativo

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

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

Aprofundamento iterativo: o retrabalho é caro?

- Nós rasos são gerados várias vezes, mas a maior parte dos nós está nos níveis mais profundos.
- Para $b = 10$ e $d = 5$:

$$N(IDS) = 123.450 \qquad N(BFS) = 111.110.$$

- A diferença no número de nós pode ser pequena frente à redução de memória.
- IDS é particularmente útil quando a profundidade da solução é desconhecida.

Comparando buscas não informadas

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

Pergunta

Qual estratégia oferece a melhor combinação de completude, otimalidade e memória em todos os problemas?

Atividade: escolhendo uma busca não informada

Atividade

Para cada situação, indique uma estratégia e justifique usando completude, otimalidade, tempo e memória:

1. solução esperada em poucos passos, todos com o mesmo custo;
2. rotas com custos diferentes e necessidade de menor custo total;
3. espaço muito profundo, memória limitada e qualquer solução é aceitável;
4. profundidade da solução desconhecida, custos de passo iguais.

Pergunta

Em quais casos a estrutura usada para a fronteira praticamente determina a resposta?

Ponto de parada: busca não informada

Retomando

Até aqui, a estratégia decidiu a expansão usando apenas informações presentes na formulação do problema.

- BFS prioriza profundidade.
- UCS prioriza custo acumulado $g(n)$.
- DFS prioriza o caminho mais recentemente explorado.
- IDS combina limites sucessivos com baixa memória.

Para pensar

Se dois caminhos têm custos semelhantes, como usar conhecimento do domínio para estimar qual deles está mais próximo da meta?

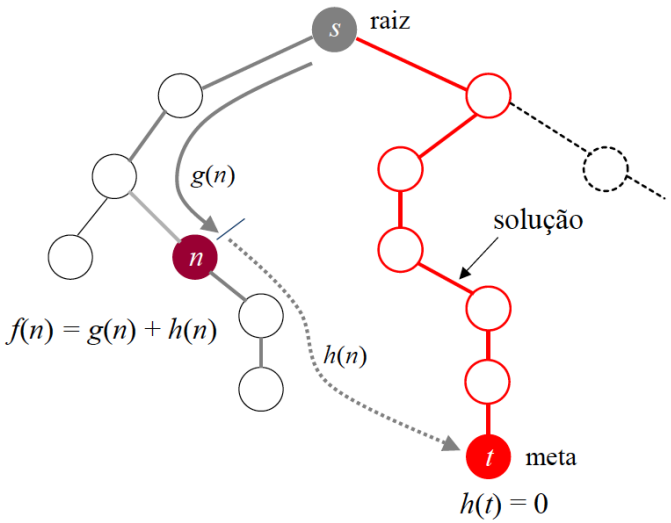
Da busca cega para a busca informada

- Em muitos problemas, sabemos mais do que apenas estados, ações e custos já percorridos.
- Podemos estimar quão promissor é um estado sem explorar todo o caminho até a meta.
- Essa estimativa é representada por uma **função heurística** $h(n)$.
- A heurística não precisa conhecer o custo real restante; ela fornece uma estimativa útil para ordenar a fronteira.

Observação

Heurística é conhecimento sobre o problema transformado em informação operacional para a busca.

$g(n)$, $h(n)$ e $f(n)$



- $g(n)$: custo conhecido do estado inicial até n.
- $h(n)$: estimativa do custo de n até uma meta.
- $f(n)$: função usada para ordenar a fronteira.

UCS: $f(n) = g(n)$ Greedy: $f(n) = h(n)$ A*: $f(n) = g(n) + h(n)$

Best-First Search

```
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
```

Observação

Best-first é uma família de estratégias: o comportamento depende da função $f(n)$ usada para priorizar a fronteira.

Busca gulosa — Greedy Best-First Search

- Usa $f(n) = h(n)$.
- Seleciona o nó que parece estar mais próximo da meta segundo a heurística.
- Ignora o custo já pago para chegar até o nó.
- Pode encontrar rapidamente uma solução, mas não há garantia de que ela seja a de menor custo.

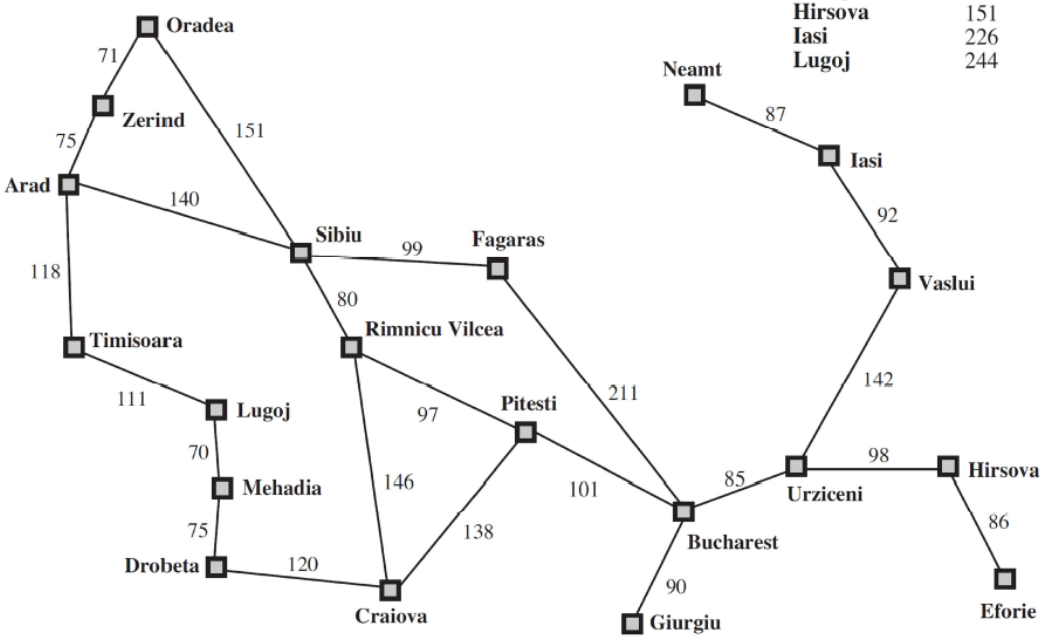
Pergunta

Qual é o risco de olhar apenas para a distância estimada até a meta e ignorar o caminho percorrido?

Heurística de distância em linha reta

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

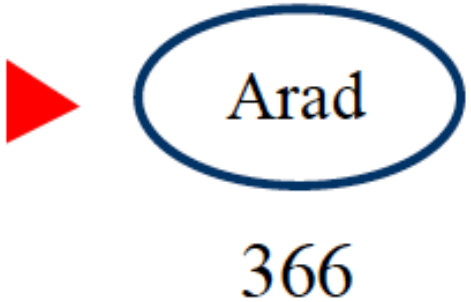


Observação

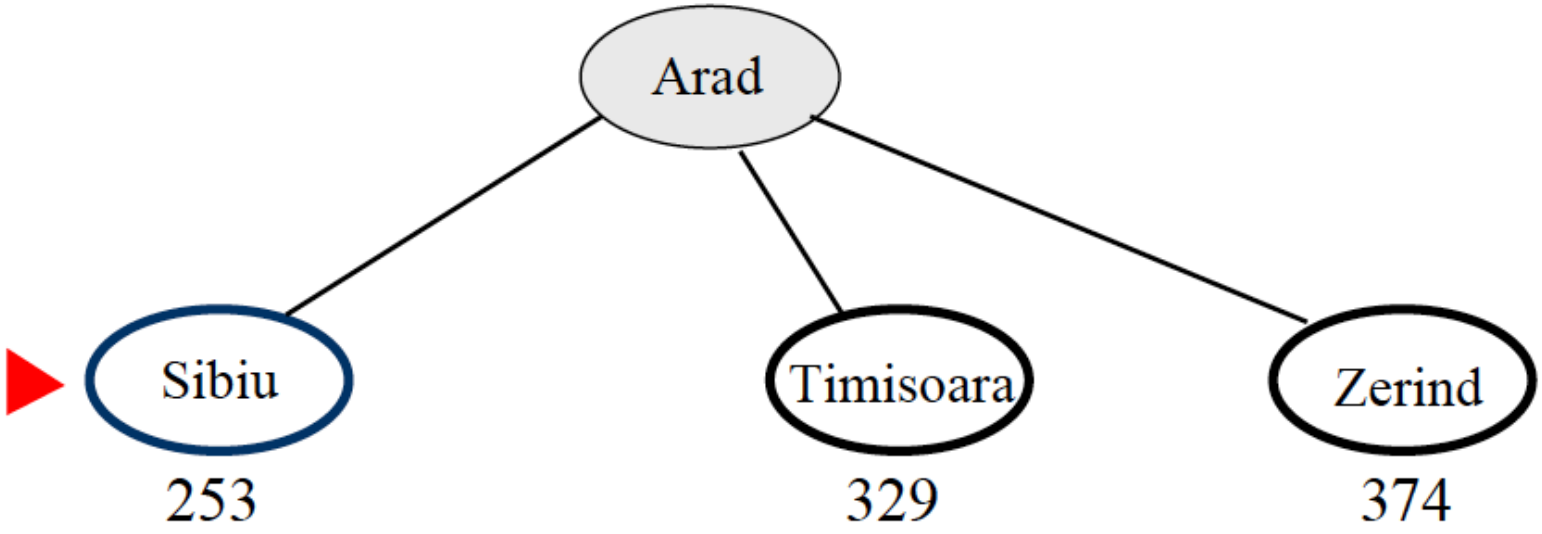
Para o problema de rotas, uma heurística natural é a distância em linha reta até Bucareste.

Busca gulosa: estado inicial

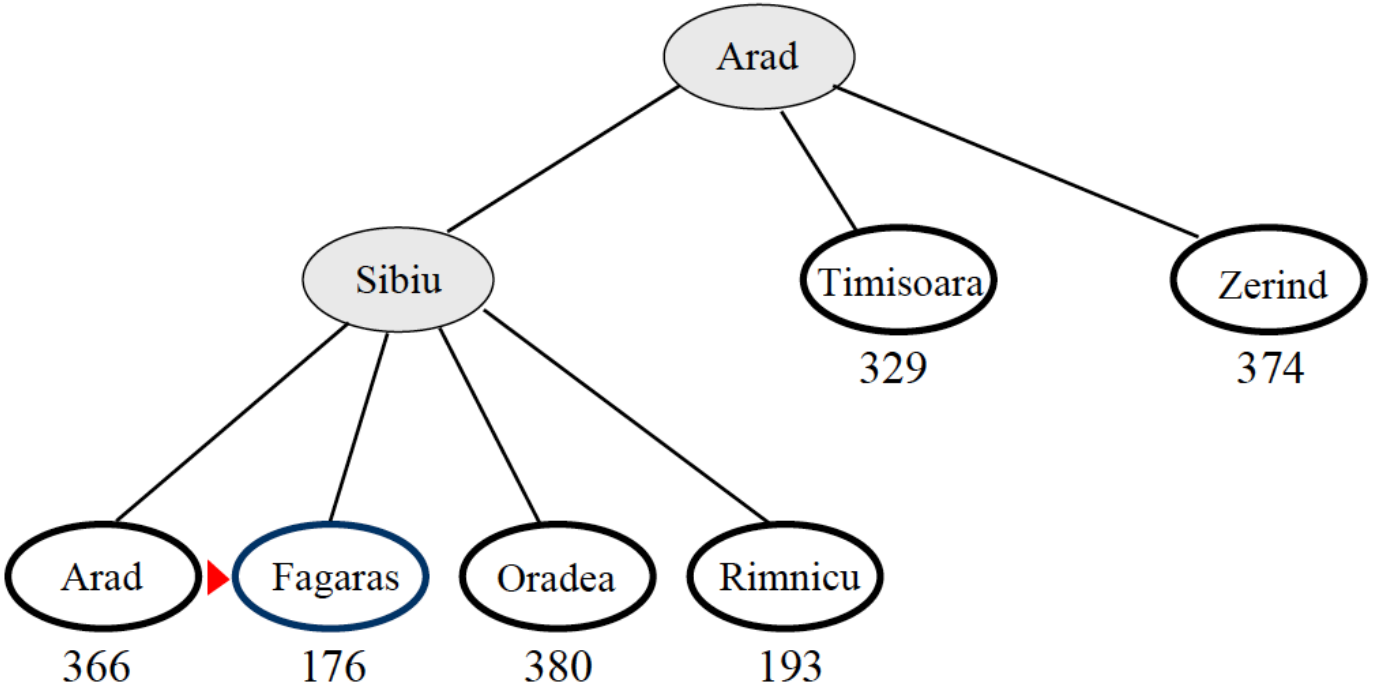
Estado inicial *In* (*Arad*)



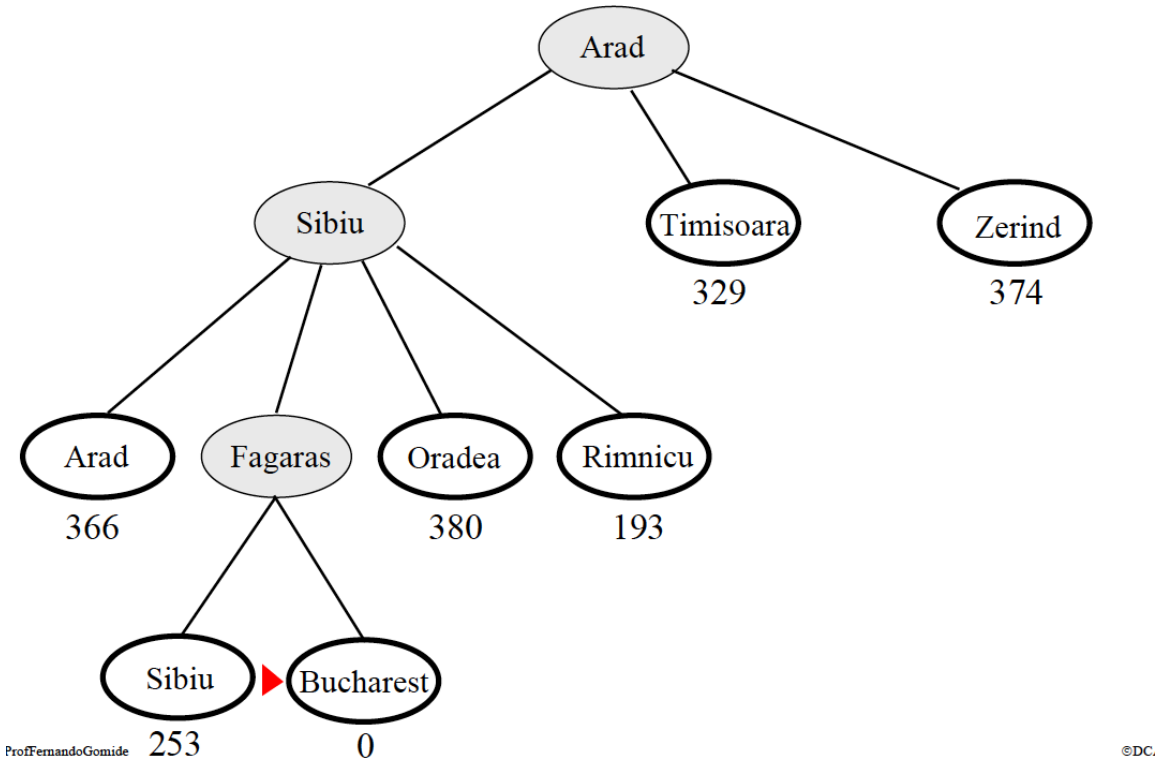
Busca gulosa: primeira escolha



Busca gulosa: expandindo o mais promissor



Busca gulosa: solução encontrada



Busca gulosa: propriedades

- Pode ter baixo custo de busca quando $h(n)$ direciona bem a exploração.
- Não é ótima em geral.
- Em busca em árvore, pode ser incompleta por causa de ciclos ou caminhos infinitos.
- Em espaço de estados finito com busca em grafo, pode ser completa.
- No pior caso, tempo e memória continuam exponenciais.

Observação

No exemplo de rotas, a busca gulosa pode chegar a Bucareste por um caminho de custo maior do que o ótimo.

Da busca gulosa para A*

Discussão

A busca gulosa usa uma estimativa do futuro, mas ignora o custo do passado. A busca de custo uniforme considera o passado, mas não usa uma estimativa do futuro.

$$\underbrace{g(n)}_{\text{custo já percorrido}} + \underbrace{h(n)}_{\text{custo estimado restante}}$$

Pergunta

O que acontece se combinarmos as duas informações para ordenar a fronteira?

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
```

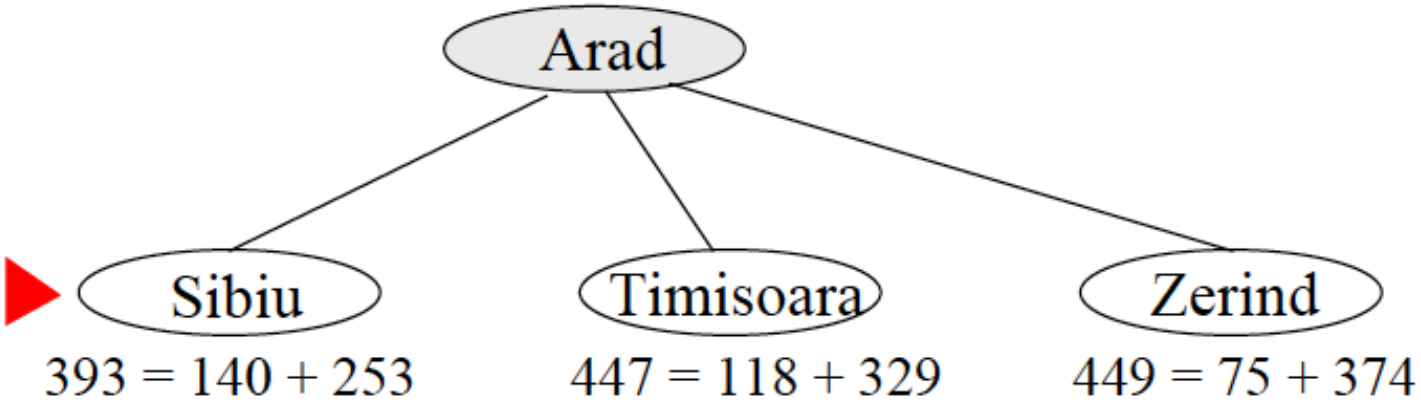
$$f(n) = g(n) + h(n)$$

A*: estado inicial

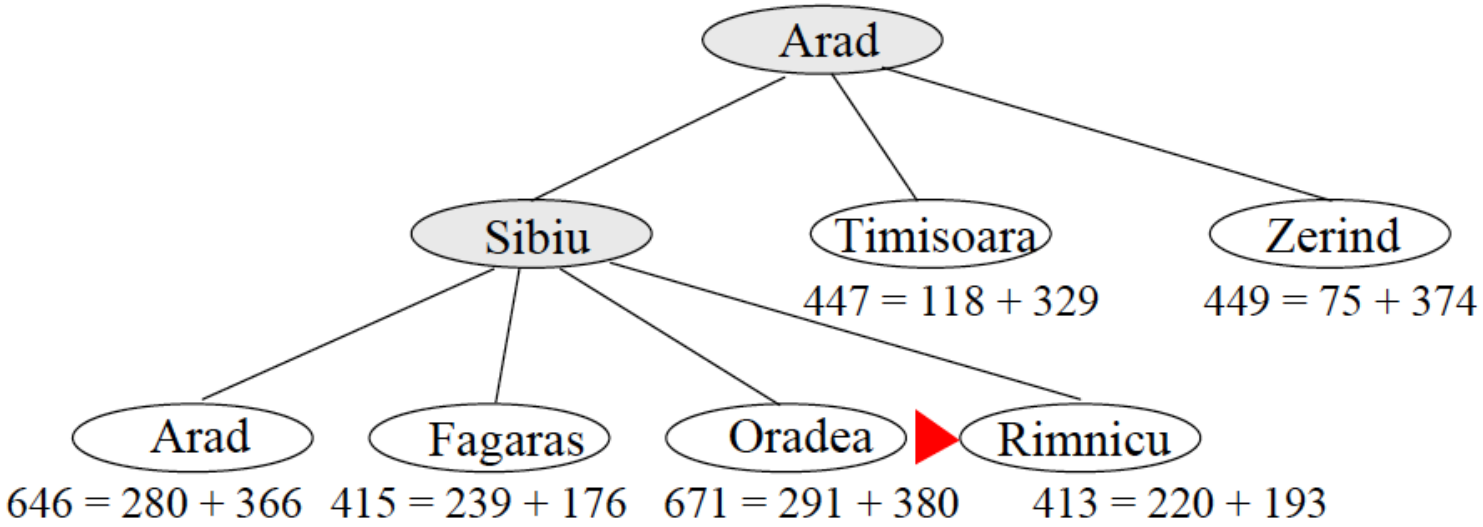
Estado inicial *In* (*Arad*) ▶ Arad

$366 = 0 + 366$

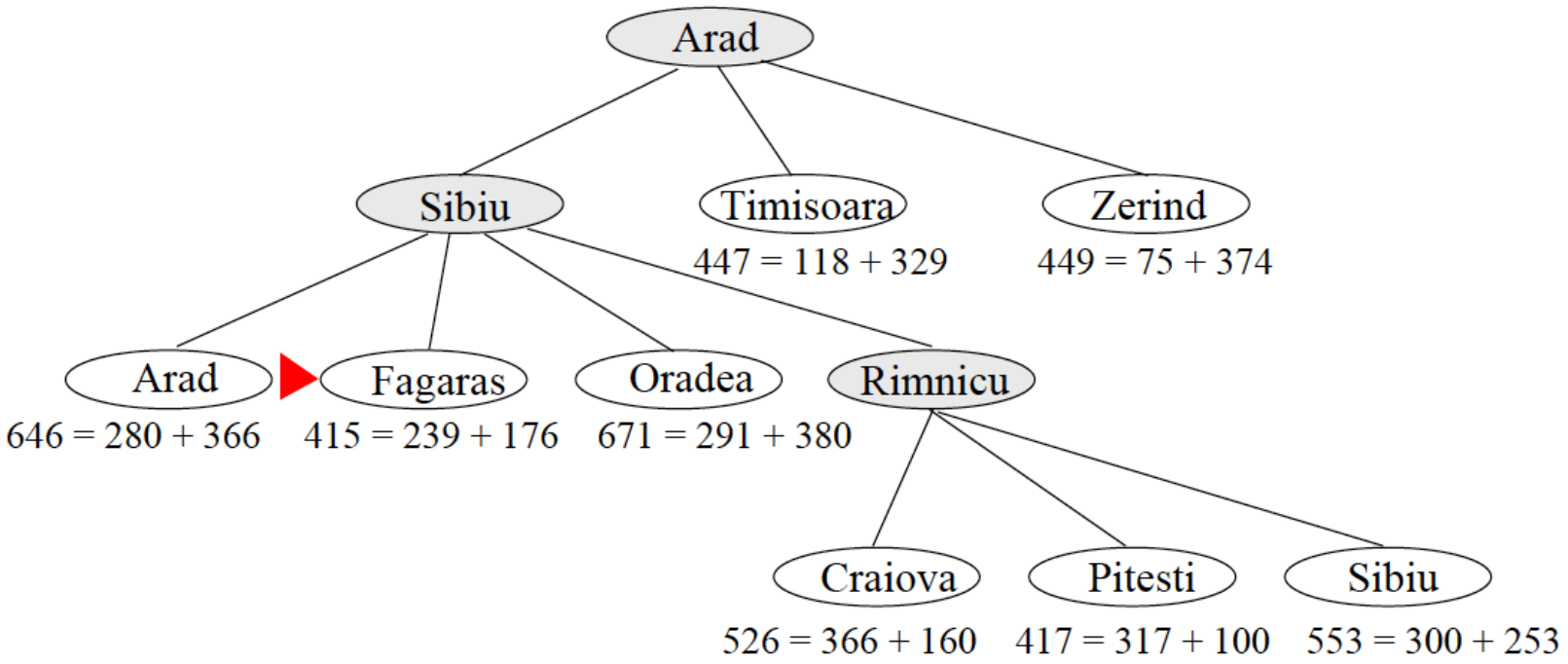
A*: avaliando os primeiros caminhos



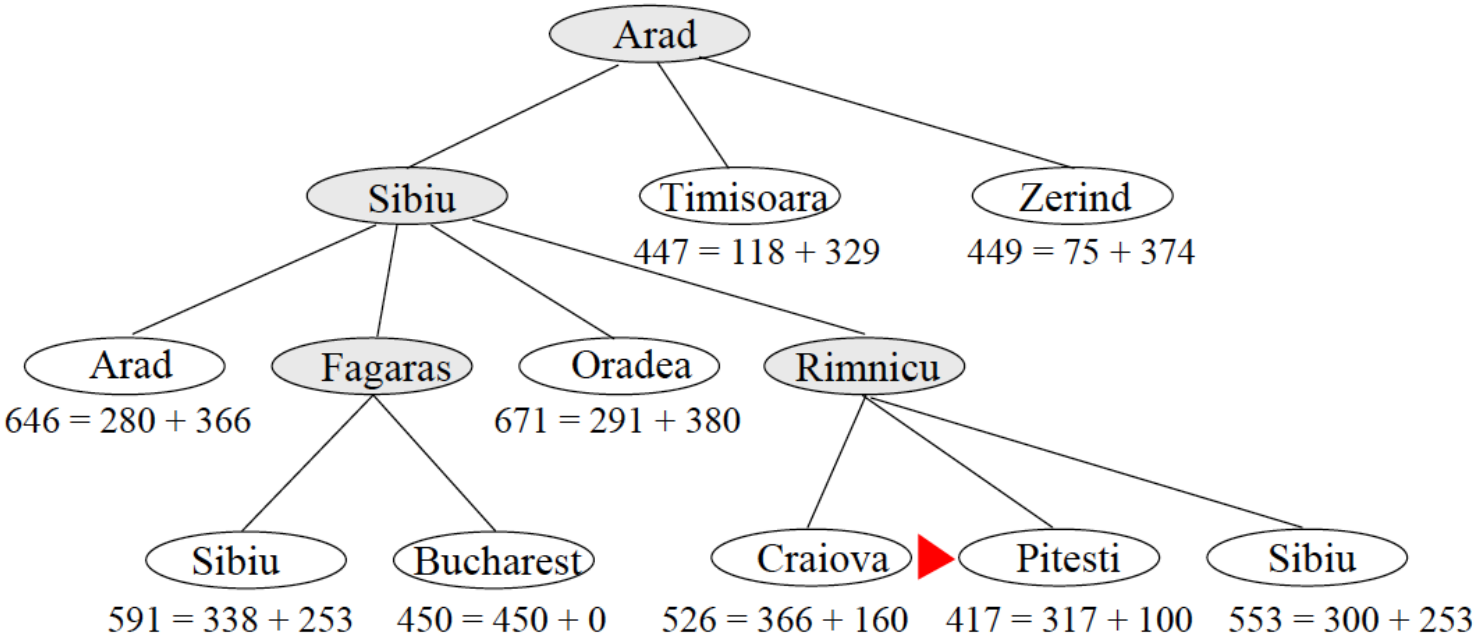
A*: custo percorrido + estimativa



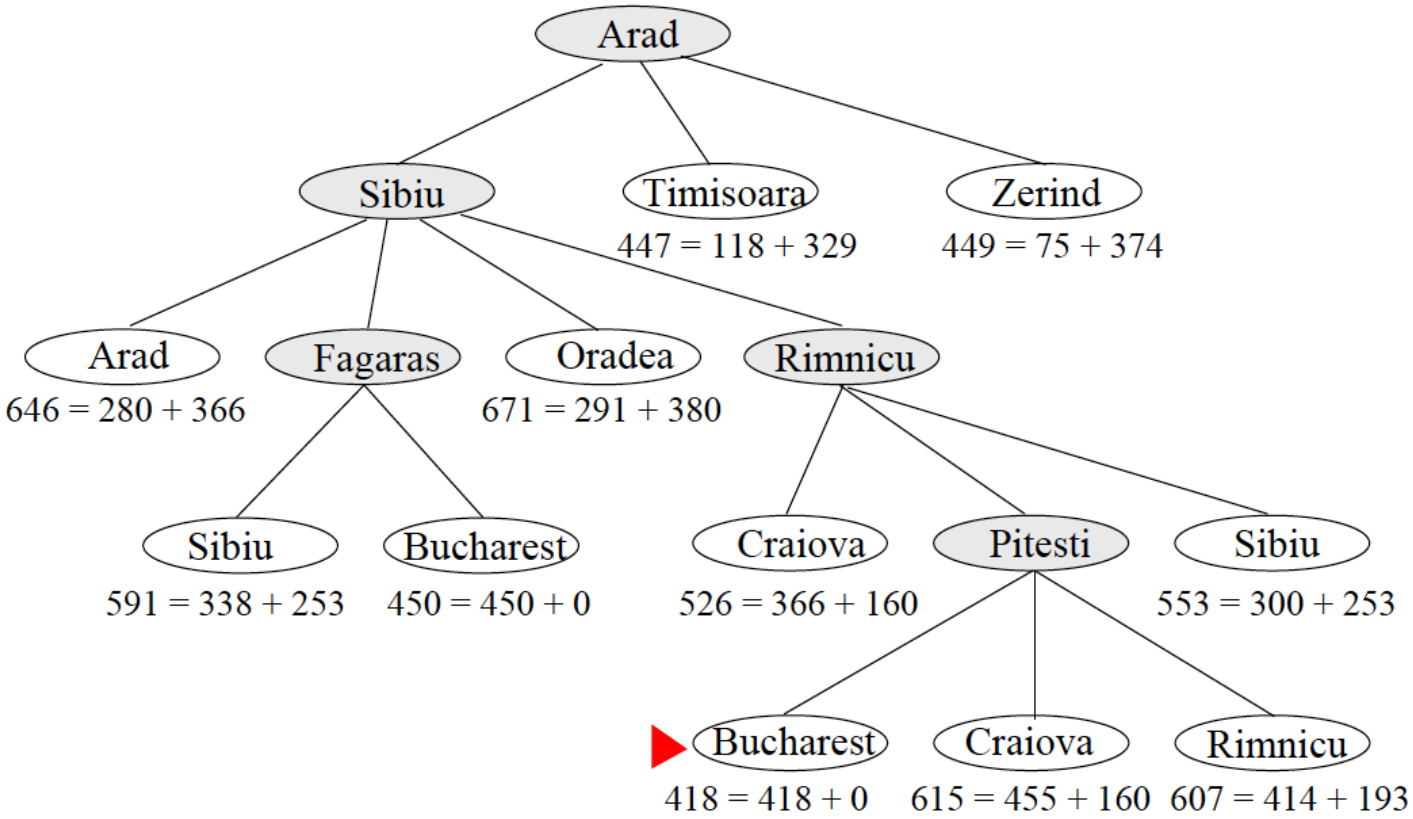
A*: atualizando a fronteira



A*: comparando alternativas



A*: solução


Arad
Sibiu
Timisoara
447 = 118 + 329
Zerind
449 = 75 + 374
Arad
646 = 280 + 366
Fagaras
Oradea
671 = 291 + 380
Rimnicu
Sibiu
591 = 338 + 253
Bucharest
450 = 450 + 0
Craiova
526 = 366 + 160
Pitesti
Sibiu
553 = 300 + 253
Bucharest
418 = 418 + 0
Craiova
615 = 455 + 160
Rimnicu
607 = 414 + 193

Quando A* encontra a solução ótima?

- Uma heurística é **admissível** quando nunca superestima o custo real mínimo até a meta:
$$0 \leq h(n) \leq h^*(n).$$
- Em busca em árvore, admissibilidade é suficiente para a otimalidade de A*.
- Para busca em grafo na formulação clássica, é desejável uma heurística **consistente**:
$$h(n) \leq c(n, a, n') + h(n').$$
- Consistência implica admissibilidade quando $h(\text{meta}) = 0$.

Observação

A* continua podendo consumir memória exponencial; sua eficiência prática depende fortemente da qualidade da heurística.

Funções heurísticas: problema das 8 peças

7	2	4
5		6
8	3	1

estado inicial

	1	2
3	4	5
6	7	8

meta

- O problema das 8 peças possui $9!/2 = 181.440$ estados atingíveis.
- Para 15 peças, o número de estados cresce para mais de dez trilhões.
- Uma boa heurística procura reduzir a quantidade de estados que precisam ser expandidos.

Duas heurísticas para as 8 peças

7	2	4
5		6
8	3	1

estado inicial

	1	2
3	4	5
6	7	8

meta

- h_1: número de peças fora da posição correta.
- h_2: soma das distâncias de Manhattan de cada peça até sua posição correta.
- Ambas podem ser admissíveis, mas h_2 normalmente fornece mais informação.

Pergunta

Se duas heurísticas são admissíveis, qual delas tende a reduzir mais a busca?

Qualidade da heurística e esforço de busca

	Search Cost (nodes generated)			Effective Branching Factor		
d	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

Observação

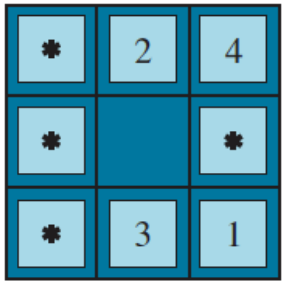
Quanto mais informativa a heurística, menor tende a ser o número de nós expandidos — desde que preservadas as propriedades necessárias para a garantia desejada.

Como obter uma heurística? Relaxação

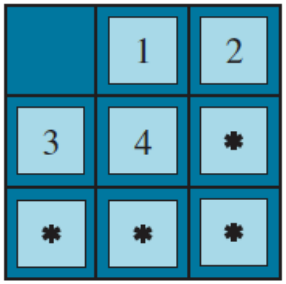
- Uma técnica clássica é criar um problema **relaxado**, removendo algumas restrições do problema original.
- No quebra-cabeça, a regra original exige que a peça se mova para um quadrado adjacente e vazio.
- Ao remover restrições, obtemos problemas mais simples cujo custo ótimo pode ser usado como estimativa.
- Exemplos:
  - considerar apenas a distância até a posição correta $\rightarrow$ distância de Manhattan;
  - permitir deslocamento mais livre $\rightarrow$ número de peças fora do lugar.

Outras formas de construir heurísticas

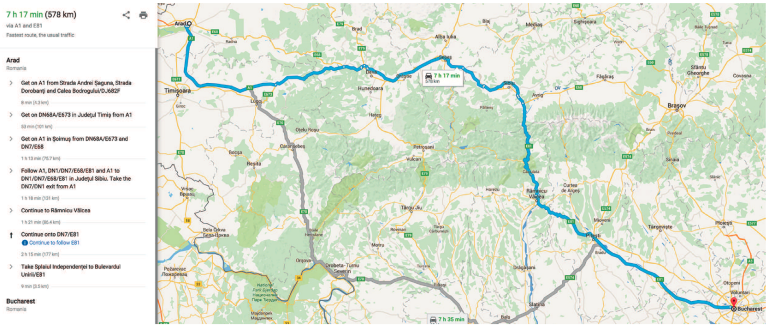
- **Pattern databases:** pré-calcular custos exatos para subproblemas menores e reutilizá-los como estimativas.
- **Pontos de referência:** usar distâncias previamente calculadas entre estados selecionados.
- **Experiência e aprendizagem:** aprender parâmetros ou estimativas a partir de problemas resolvidos anteriormente.



Start State



Goal State



Aprofundamento: A* e limitação de memória

- A* pode manter um número muito grande de nós na fronteira.
- Estratégias como **Recursive Best-First Search (RBFS)** procuram reproduzir o comportamento best-first usando espaço aproximadamente linear na profundidade.
- O algoritmo mantém o melhor valor alternativo disponível e retorna quando o caminho atual deixa de ser competitivo.
- O custo é recomputar partes da busca quando caminhos são revisitados.

Observação

RBFS é um aprofundamento importante, mas o conceito central desta aula é compreender a relação entre $g(n)$, $h(n)$ e $f(n)$.

RBFS: estrutura do algoritmo

```
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 ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
  add CHILD-NODE(problem, node, action) into sucessors
  if successors is empty then return failure, ∞
   for each s in successors do      /* update f with values from previous search, if any */
      s.f ← max (s.g + s.h, node.f )
   loop do
      best ←  the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS ( problem, best, min (f_limit, alternative))
      if result ≠ failure then return result
```

Síntese: estratégias de busca

Estratégia	Prioriza	Ponto forte	Limitação típica
BFS	profundidade	completa	memória
UCS	$g(n)$	ótima	custo de busca
DFS	profundidade/LIFO	pouca memória	não ótima
IDS	profundidade crescente	pouca memória	retrabalho
Greedy	$h(n)$	direcionamento	não ótima
A*	$g(n) + h(n)$	ótima com condições	memória

Pergunta

A melhor estratégia depende apenas do algoritmo ou também da estrutura do problema e do conhecimento disponível?

Atividade final: qual estratégia escolher?

Atividade

Para cada cenário, escolha uma estratégia e justifique:

1. encontrar qualquer sequência curta de movimentos em um jogo com custos iguais;
2. encontrar a rota de menor custo em um grafo com pesos positivos;
3. explorar um espaço muito profundo com memória restrita;
4. encontrar uma rota quando existe uma boa estimativa de distância até o destino;
5. encontrar a rota ótima aproveitando essa estimativa.

Pergunta

Para os itens 4 e 5, explique por que Greedy e A* podem expandir nós diferentes mesmo usando a mesma heurística.

Ponto de parada: para a Parte II

Retomando

Nesta aula, buscamos caminhos em um espaço de estados conhecido e, em geral, determinístico.

- Estratégias não informadas diferem pela organização da fronteira.
- Estratégias informadas usam heurísticas para direcionar a exploração.
- A* combina custo percorrido e custo estimado restante.

Para pensar

E se não precisarmos do caminho, apenas de um bom estado? E se uma ação puder produzir resultados diferentes do esperado?

Dúvidas

Dúvida é o começo da sabedoria.