

# PROCURA HEURÍSTICA

## CAP 3 (3.5 E 3.6)

---

Parcialmente adaptado de  
<http://aima.eecs.berkeley.edu>

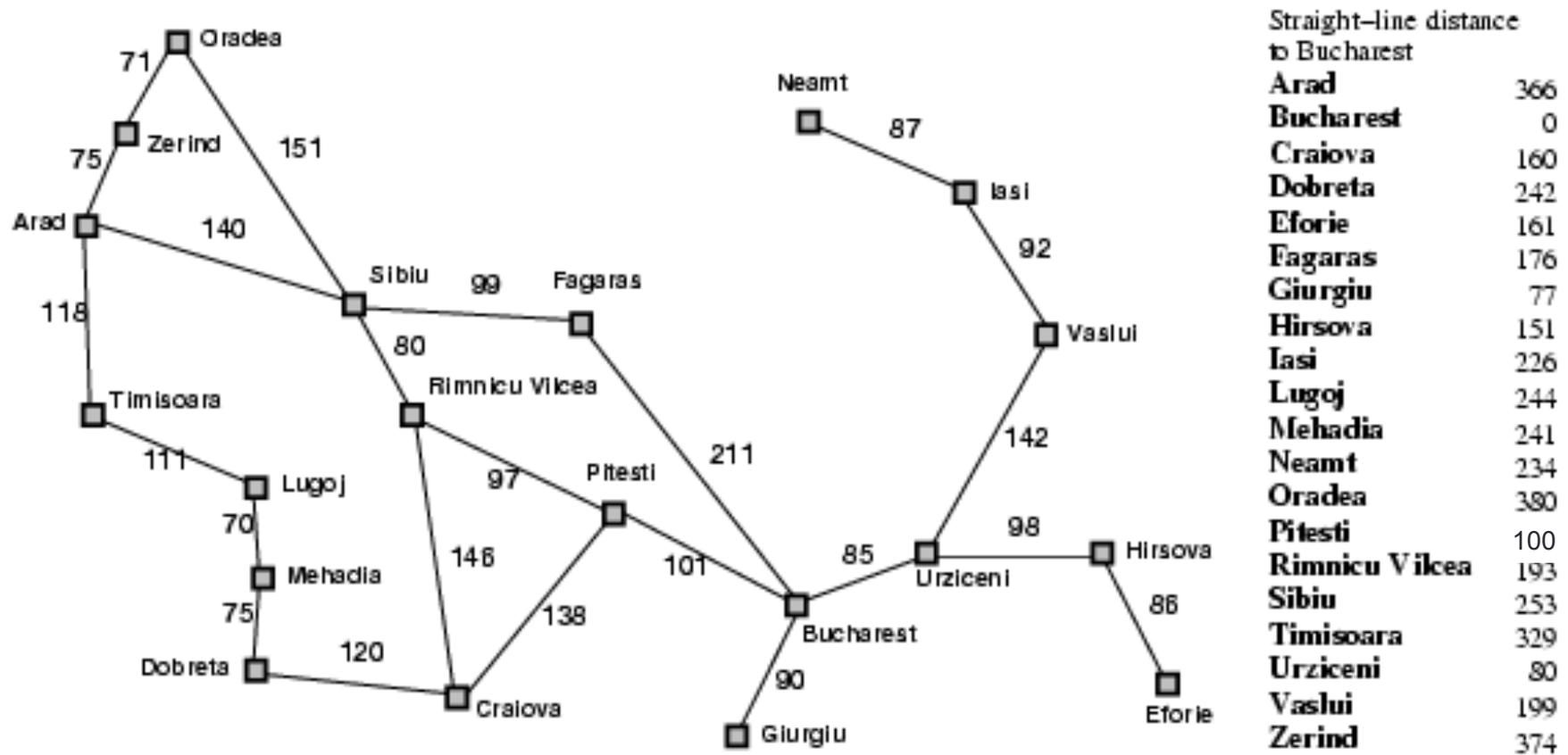
# Resumo

- Procura pelo melhor primeiro
- Procura sôfrega
- Procura  $A^*$
- Heurísticas e suas propriedades
- Procura informada com memória limitada

# Procura pelo melhor primeiro

- Ideia: aplicar uma **função de avaliação**  $f(n)$  a cada nó
  - Indica-nos se o nó é promissor ou não
  - expandir o nó que aparenta ser mais promissor
- Implementação:  
Ordenar os nós na fronteira por ordem crescente (minimizar) ou decrescente (maximizar) da função de avaliação
- Casos especiais:
  - Procura sôfrega
  - Procura  $A^*$

# Roménia com distâncias em linha recta km



# Procura sôfrega

- Função de avaliação  $f(n) = h(n)$  (**h**eurística)
- = estimativa do custo do menor caminho para ir de  $n$  até a um estado objectivo
  - $h_{SLD}(n)$  = distância em linha recta de  $n$  até Bucareste
- Procura sôfrega expande o nó que **aparenta** estar mais próximo do objectivo

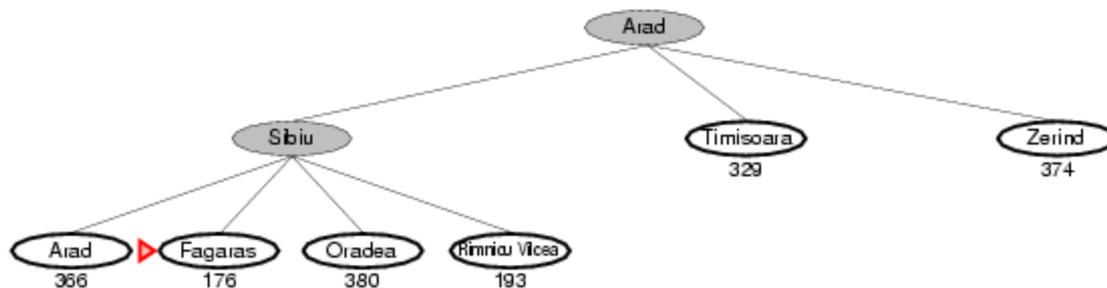
# Exemplo de procura sôfrega



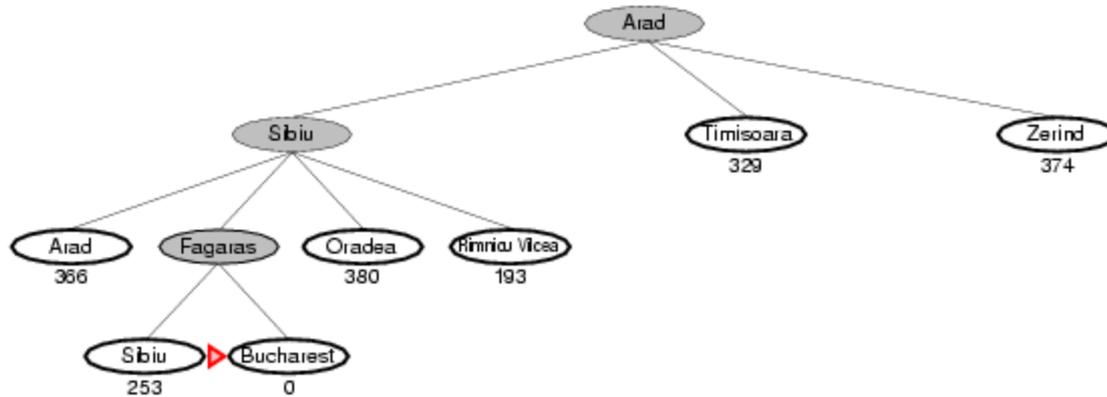
# Exemplo de procura sôfrega



# Exemplo de procura sôfrega



# Exemplo de procura sôfrega



# Propriedades da procura sôfrega

- Completa? Não – pode ficar presa em ciclos, e.g., Ir de Iasi para Oradea, temos Iasi → Neamt → Iasi → Neamt  
Completa em espaços finitos com verificação de estados repetidos
- Tempo?  $O(b^m)$ , mas uma boa heurística pode ter melhorias espectaculares
- Espaço?  $O(b^m)$  – mantém todos os nós em memória
- Óptima? Não

# Procura A\*

- Ideia: evitar expandir caminhos que já têm elevado custo
- Função de avaliação  $f(n) = g(n) + h(n)$ 
  - $g(n)$  = custo atual para atingir  $n$
  - $h(n)$  = custo estimado para atingir o objectivo a partir de  $n$
  - $f(n)$  = custo total estimado do caminho até ao objectivo passando por  $n$

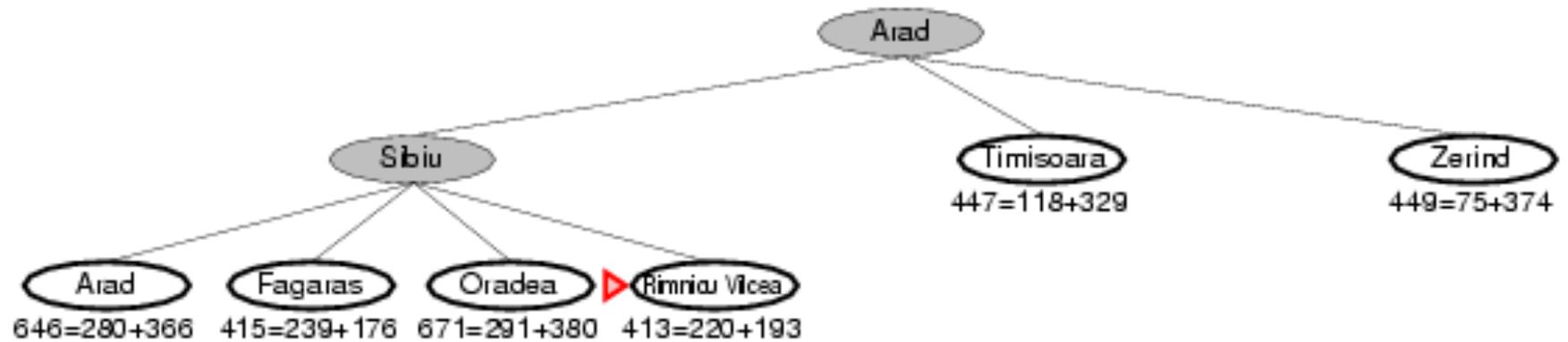
# Exemplo de Procura A\*

▶ Arad  
 $366 = 0 + 366$

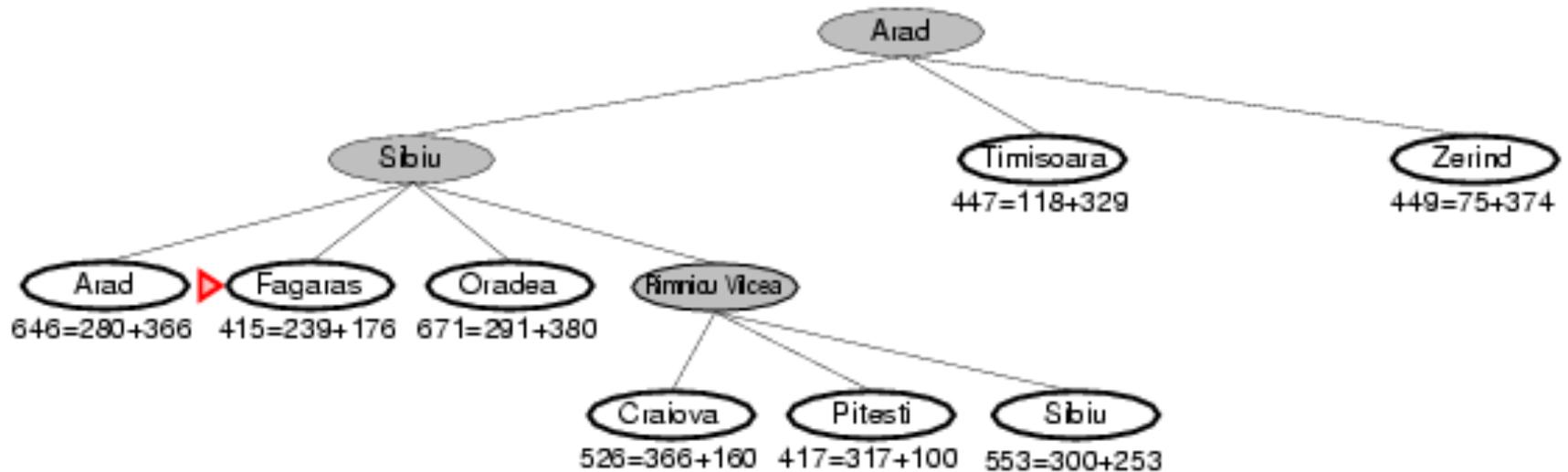
# Exemplo de Procura A\*



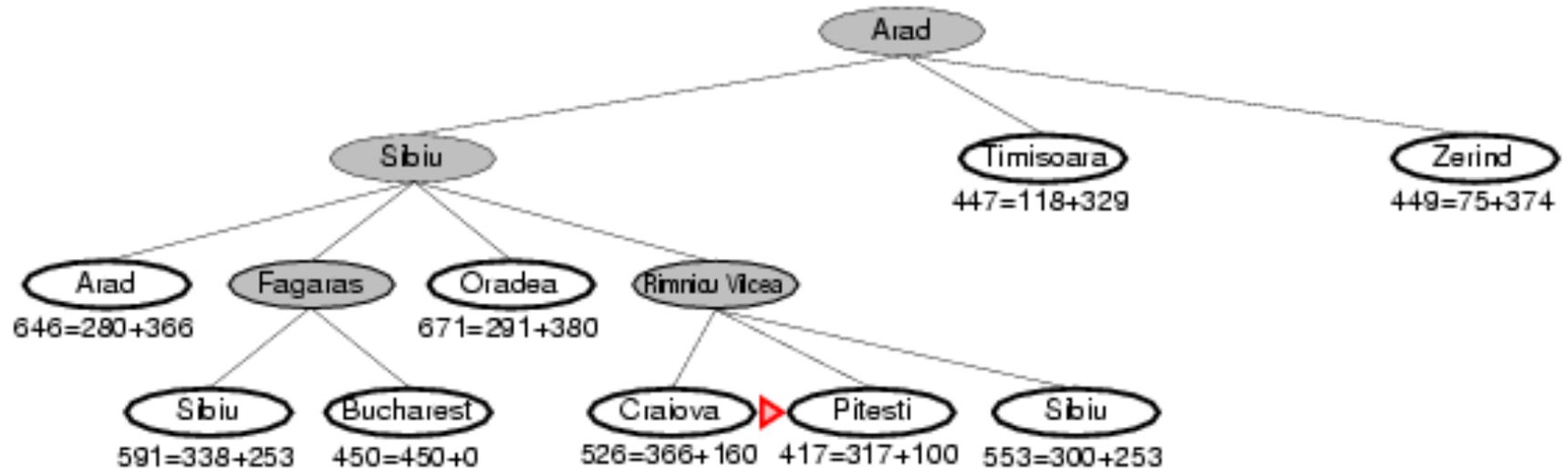
# Exemplo de Procura A\*



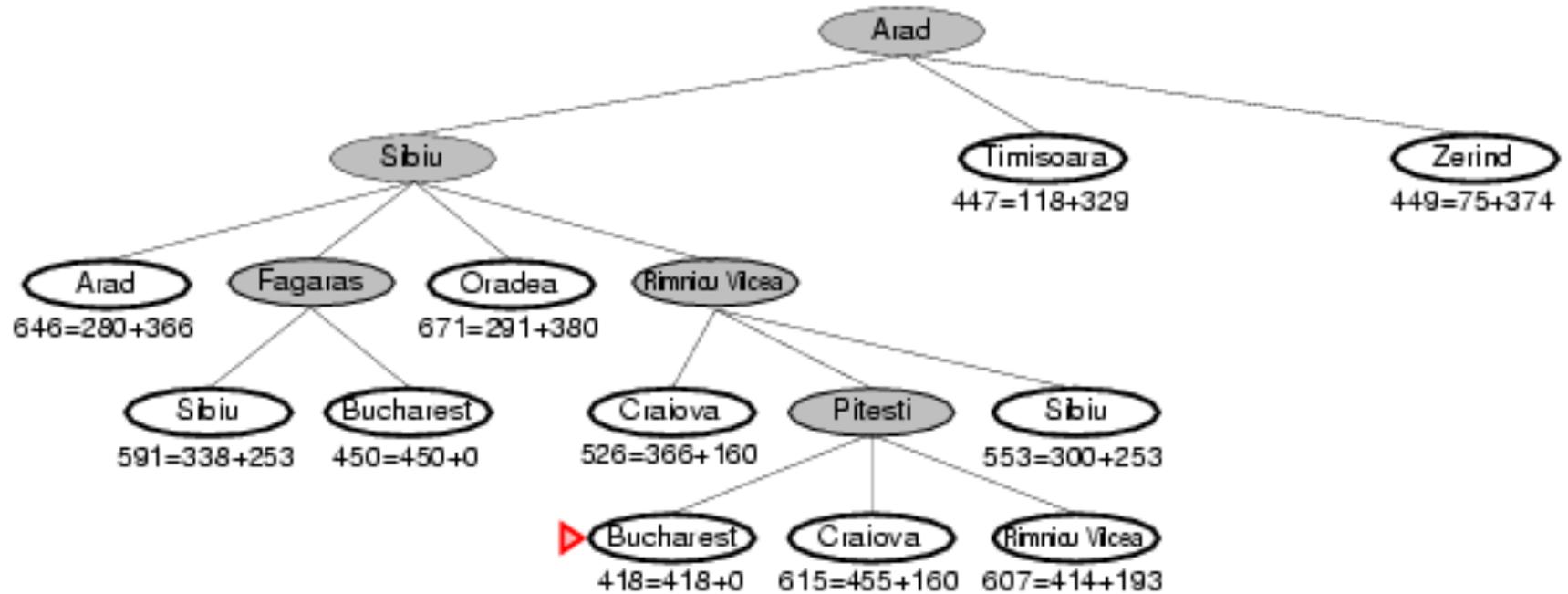
# Exemplo de Procura A\*



# Exemplo de Procura A\*



# Exemplo de Procura A\*

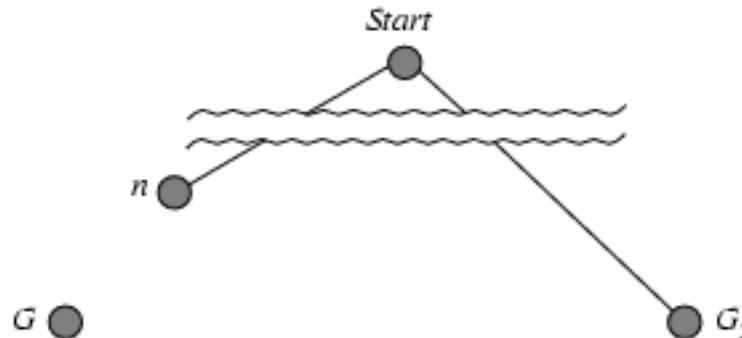


# Heurísticas admissíveis

- Uma heurística  $h(n)$  é **admissível** se para todo o nó  $n$ ,  $h(n) \leq h^*(n)$ , em que  $h^*(n)$  é o custo **real** de atingir o objectivo a partir de  $n$ .
- Uma heurística admissível **nunca sobrestima** o custo de alcançar o objectivo, i.e., é **optimista**.
- Exemplo:  $h_{SLD}(n)$  (nunca sobrestima a distância por estrada)
- **Teorema:** Se  $h(n)$  é admissível, então o algoritmo  $A^*$  usando TREE-SEARCH é óptimo.

# Optimalidade de $A^*$ (demonstração)

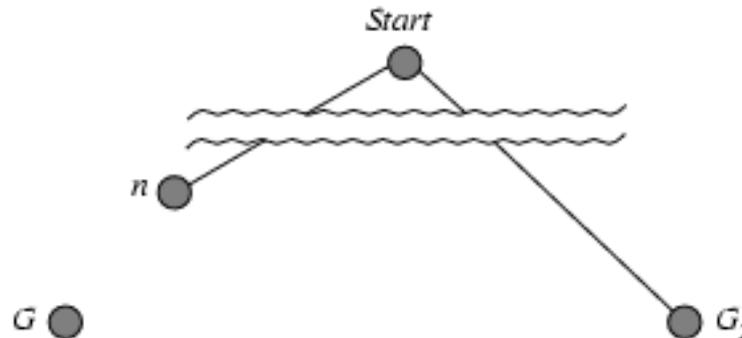
- Suponha-se que um estado final subóptimo  $G_2$  foi gerado e encontra-se na fronteira. Seja  $n$  um nó por expandir na fronteira num caminho mais curto para o objectivo óptimo  $G$ .



- $f(G_2) = g(G_2)$  pois  $h(G_2) = 0$
- $g(G_2) > g(G)$  porque  $G_2$  é subóptimo
- $f(G) = g(G)$  pois  $h(G) = 0$
- Logo  $f(G) < f(G_2)$

# Optimalidade de $A^*$ (redução ao absurdo)

- Suponha-se que um estado final subóptimo  $G_2$  foi gerado e encontra-se na fronteira. Seja  $n$  um nó por expandir na fronteira num caminho mais curto para o objectivo óptimo  $G$ .

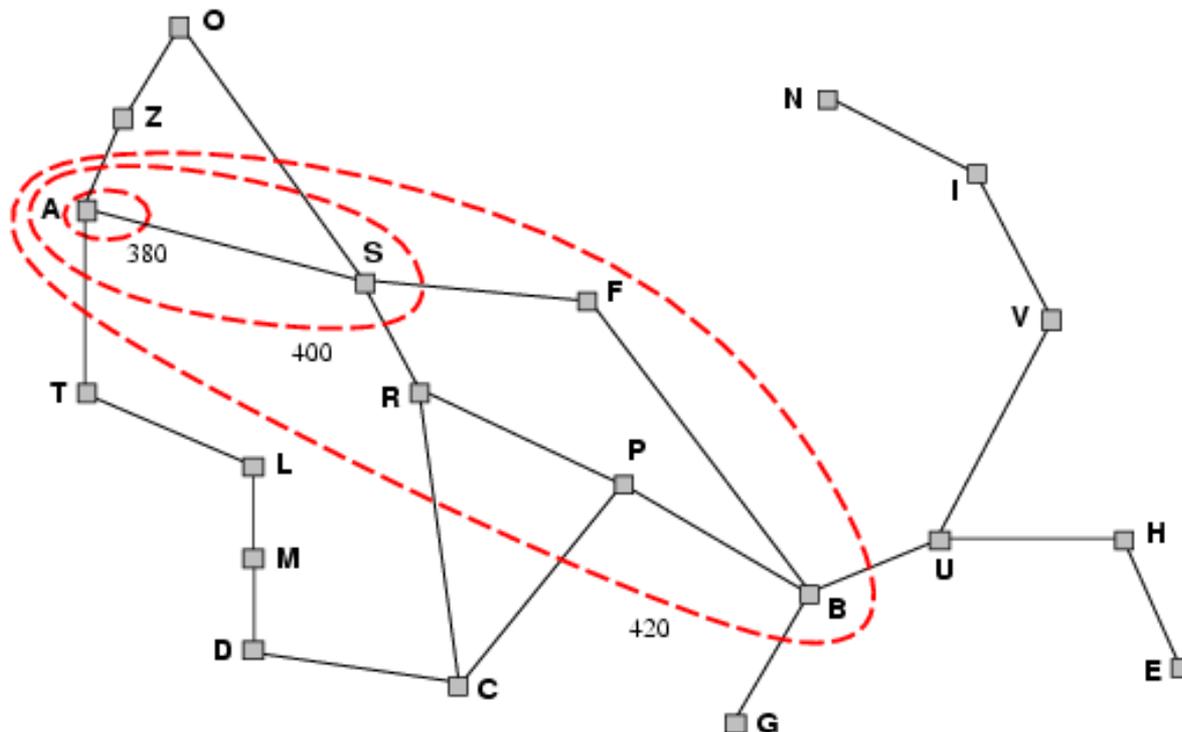


- $f(G_2) > f(G)$  como se viu anteriormente
- $h(n) \leq h^*(n)$  porque  $h$  é admissível
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

Portanto  $f(G_2) > f(n)$ , e o  $A^*$  nunca selecionará  $G_2$  para expansão

# Optimalidade de $A^*$ (mais útil)

- $A^*$  expande nós por ordem crescente de valores da função de avaliação
- Adiciona gradualmente contornos aos nós (c.f. procura em largura adiciona níveis)
- Contorno  $i$  tem todos os nós  $f=f_i$ , em que  $f_i < f_{i+1}$



# Propriedades do A\*

- O A\* expande todos os nós com  $f(n) < C^*$
- O A\* expande alguns nós com  $f(n) = C^*$
- O A\* nunca expande nós com  $f(n) > C^*$

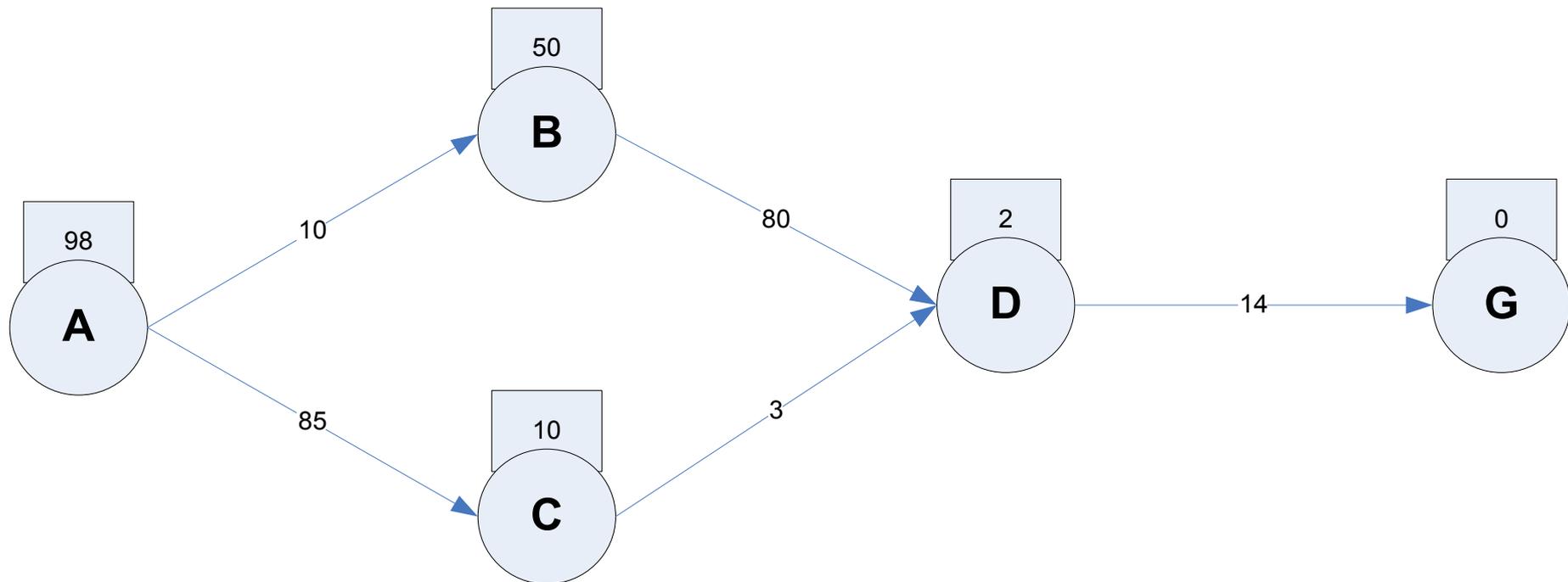
O algoritmo A\* é optimalmente eficiente para qualquer heurística dada:

- Não há outro algoritmo **óptimo** que garantidamente expanda um menor número de nós!

# Propriedades do A\*

- Completo? Sim (a não ser que haja um número infinito de nós com  $f \leq f(G)$  )
- Tempo? Exponencial no [erro relativo de  $h$  x o tamanho da solução]  
Se  $|h(n) - h^*(n)| \leq O(\log h^*(n))$  o algoritmo A\* tem um comportamento subexponencial.
- Espaço? Mantém todos os nós em memória
- Ótimo? Sim, se a heurística for admissível

## Problemas da versão naïve do A\* com procura em grafos



fronteira

A(98)	B(60) C(95)	D(92) C(95)	C(95) G(104)	D(90) G(104)	G(104)
	A	AB	ABD	ABCD	ABCD

explorados

# Heurísticas consistentes

A demonstração de optimalidade do A\* não se generaliza para o algoritmo de procura em grafos (eliminação de estados já explorados)

- Uma heurística é **consistente** se para todo o nó  $n$  e todo o seu sucessor  $n'$ , gerado pela ação  $a$ ,

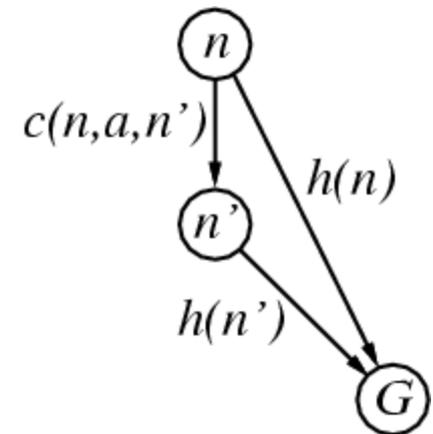
$$h(n) \leq c(n,a,n') + h(n')$$

- Se  $h$  é consistente, temos

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

- ou seja,  $f(n)$  é não decrescente ao longo de qualquer caminho (é **monótona**).

- **Teorema:** Se  $h(n)$  for consistente, então o A\* recorrendo à procura GRAPH-SEARCH é ótimo.



# A\* (versão otimizada em grafos)

```
function A*( problem ) returns a solution, or failure
  node ← a node with STATE=problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by f-value with node as the only element
  explored ← a singleton set with node.STATE
  loop do
    if EMPTY?( frontier ) then return failure
    node ← POP( frontier ) /* chooses the node with lowest f-value 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 do
        frontier ← INSERT(child , frontier )
      else if child.STATE is in frontier with higher f-value then
        replace that frontier node with child
```

Ótimo só para  
heurísticas  
consistentes!!!

# O que fazer com heurísticas inconsistentes?

Solução simples: o conjunto de explorados mantém nós em vez de estados.

- Seja  $n$  um novo nó gerado pelo algoritmo. Se existir um nó  $m$  no conjunto de explorados para o mesmo estado de  $n$  tal que  $f(n) < f(m)$  então retira-se o nó  $m$  do conjunto de explorados e coloca-se o novo nó  $n$  na fronteira.
- Com esta alteração, a utilização de heurísticas admissíveis garantem novamente a optimalidade da primeira solução encontrada pelo algoritmo de procura em grafos  $A^*$ .

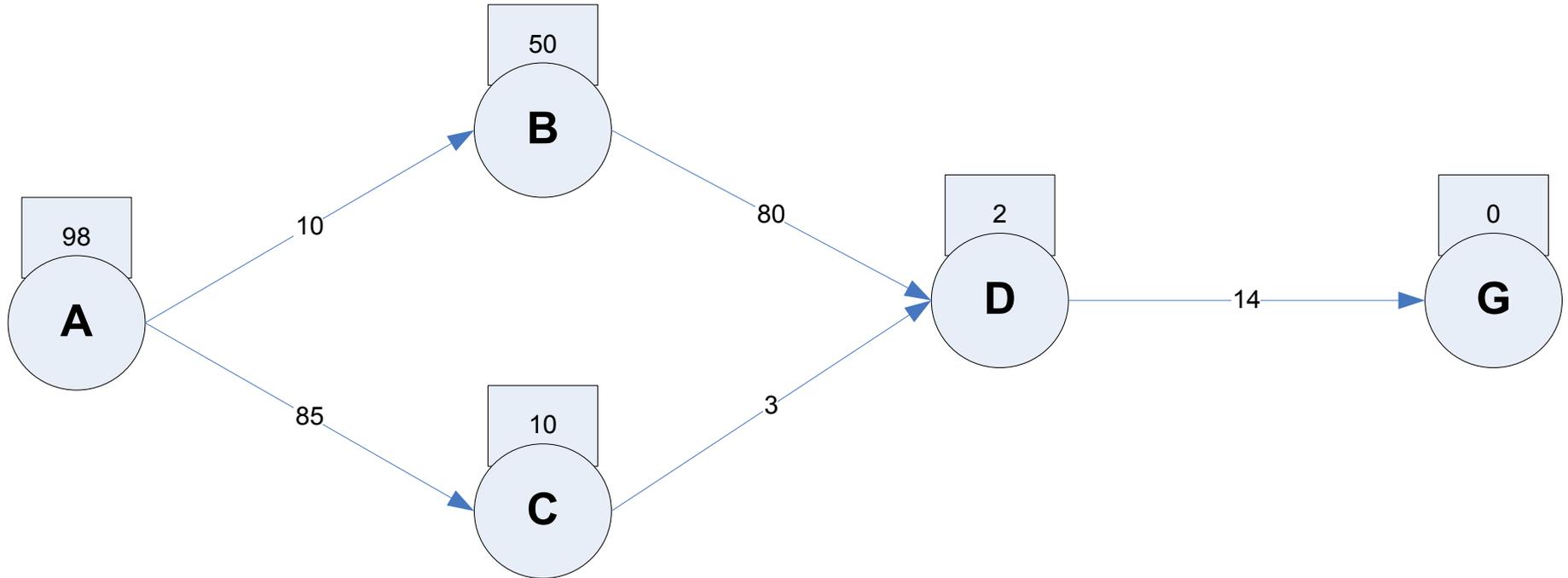
# Procura A\* em grafos corrigida

```
function GRAPH-SEARCH( problem, frontier ) returns a solution, or failure
  explored ← an empty set of nodes
  node ← node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← INSERT(node, frontier) /* priority queue ordered by f-value */
  loop do
    if EMPTY?(frontier) return failure
    node ← POP( frontier ) /* chooses the node with lowest f-value in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    if node.STATE is not in explored then
      add node to explored
      frontier ← INSERT-ALL(EXPAND(node,problem),frontier)
    else if node.STATE = oldnode.STATE such that oldnode in explored
      has higher f-value than node then
      replace oldnode by node in explored
      frontier ← INSERT-ALL(EXPAND(node,problem),frontier)
    endif
```

Garante óptimo  
para heurísticas  
admissíveis!!!



# A\* com procura em grafos (corrigido)



fronteira

...	C(95) G(104)	D(90) G(104)	G(102) G(104)
...	A(98) B(60) D(92)	A(98) B(60) C(95) D(92)	A(98) B(60) C(95) D(90)

explorados

# Estimativa PathMax

Existe uma otimização que tenta manter a heurística consistente (estimativa PathMax) mas mais complexa.

- A ideia consiste em utilizar como valor da função heurística

$$h^{\wedge}(m) = \max \{(h(n) - c(n, a, m)) ; h(m)\}$$

- em que  $m$  é sucessor de  $n$ . O valor de  $h^{\wedge}(m)$  é obtido em tempo de execução e depende do caminho seguido para atingir  $m$ .
- Poderá ser necessário remover na mesma nós do conjunto de explorados.

# Implementação dos algoritmos

Obviamente, deve-se ter algum cuidado na seleção das estruturas de dados para implementar a fronteira e o conjunto de estados explorados. Habitualmente o conjunto de estados explorados é implementado com uma tabela de dispersão (hash table). Quanto à fronteira, normalmente opta-se por:

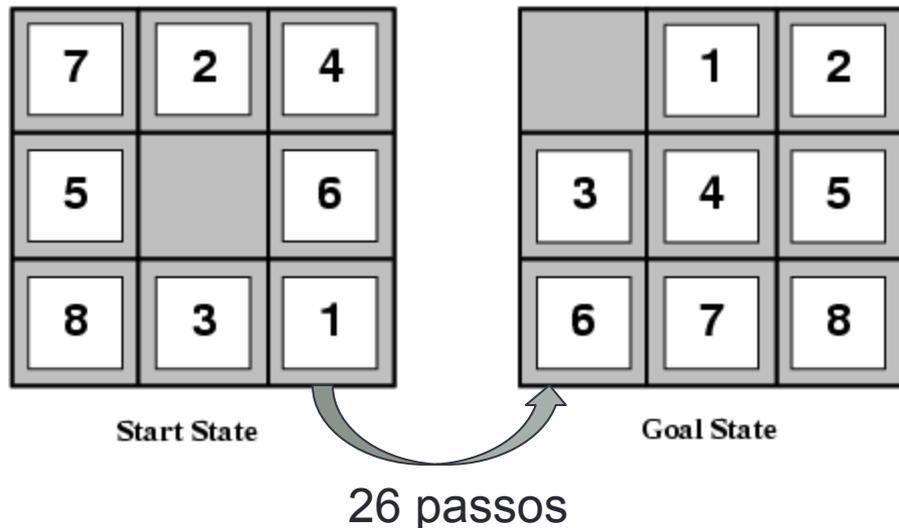
- Fila de prioridade (priority queue) quando o grafo de estados é esparso: número reduzido de nós sucessores limitados por uma constante pequena. Complexidade temporal  $O(N * \log_2 N + L * \log_2 N)$ , em que  $N$  o número de estados e  $L$  o número de arcos. Esta é a situação habitual:
  - No pior caso têm de se retirar  $N$  nós da fila de prioridade, cada uma destas operações da ordem de  $\log_2 N$
  - São necessárias no pior caso  $L$  inserções na fila de prioridade, cada uma com custo  $\log_2 N$ .
- Quando o grafo é denso, então deve-se utilizar uma lista ou tabela de dispersão. Complexidade temporal da ordem de  $O(N^2 + L)$ 
  - Retirar o nó com menor custo é operação  $O(N)$ , no máximo  $N$  vezes.
  - A inserção de um nó sucessor na fronteira pode ser feita com uma operação de  $O(1)$

# Comparação implementações

N	Densidade	L	$N \cdot \log N + L \cdot \log N$	$N \cdot N + L$	Rácio
10	1%	1	37	101	0,36
10	10%	10	66	110	0,60
10	50%	50	199	150	1,33
10	90%	90	332	190	1,75
10	100%	100	365	200	1,83
100	1%	100	1329	10100	0,13
100	10%	1000	7308	11000	0,66
100	50%	5000	33884	15000	2,26
100	90%	9000	60459	19000	3,18
100	100%	10000	67103	20000	3,36
1000	1%	10000	109624	1010000	0,11
1000	10%	100000	1006544	1100000	0,92
1000	50%	500000	4992858	1500000	3,33
1000	90%	900000	8979172	1900000	4,73
1000	100%	1000000	9975750	2000000	4,99
10000	1%	1000000	13420590	101000000	0,13
10000	10%	10000000	133010001	110000000	1,21
10000	50%	50000000	664518496	150000000	4,43
10000	90%	90000000	1196026991	190000000	6,29
10000	100%	100000000	1328904115	200000000	6,64
100000	1%	100000000	1662625011	10100000000	0,16
100000	10%	1000000000	16611301438	11000000000	1,51
100000	50%	5000000000	83049863336	15000000000	5,54
100000	90%	9000000000	149488425234	19000000000	7,87
100000	100%	10000000000	166098065708	20000000000	8,30

# Heurísticas admissíveis

Para a charada-8 uma procura exaustiva explora em média  $3,1 \times 10^{10}$  nós. Mas existem apenas 181440 estados (charada-15 são  $10^{13}$ ). É fundamental a utilização de heurísticas



- $h_1(n)$  = número de peças colocadas erradamente
  - $h_1(S) = 8$
- $h_2(n)$  = soma das distâncias de Manhattan
  - $h_2(S) = 3+1+2+2+3+2+2+3=18$

# Dominância

- Se  $h_2(n) \geq h_1(n)$  para todo o  $n$  (ambas admissíveis) então  $h_2$  **domina**  $h_1$ , sendo  $h_2$  melhor na procura
- Custos típicos de procura para charada-8 (número médio de nós expandidos):
  - $d=12$ 

AP = 3,644,035 nós	( $b^* = 2,78$ )
$A^*(h_1) = 227$ nós	( $b^* = 1,42$ )
$A^*(h_2) = 73$ nós	( $b^* = 1,24$ )
  - $d=24$ 

AP $\approx$ 54.000.000.000 nós	
$A^*(h_1) = 39,135$ nós	( $b^* = 1,48$ )
$A^*(h_2) = 1,641$ nós	( $b^* = 1,26$ )
- Caso  $h_2(n)$  não domine  $h_1(n)$ , e vice-versa, pode-se sempre combinar as heurísticas com a expressão  $\max \{h_1(n), h_2(n)\}$
- O factor de ramificação efetivo  $b^*$  caracteriza a qualidade da heurística utilizada. O valor  $b^*$  é obtido resolvendo a equação  $N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$ , em que  $N$  é o número de nós gerados pelo  $A^*$  e  $d$  a profundidade da solução obtida.

# Problemas relaxados

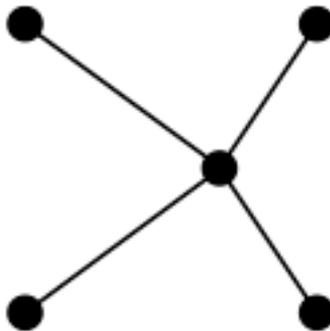
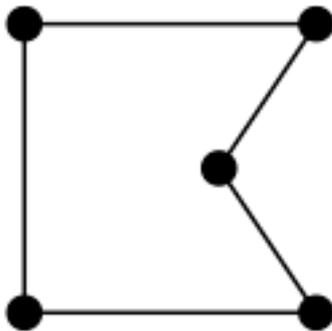
- Um problema com menos restrições nas acções é designado por **problema relaxado**.
- O custo exato de uma solução óptima para o problema relaxado é uma heurística admissível para o problema original!

Considere-se a charada- $n$  novamente

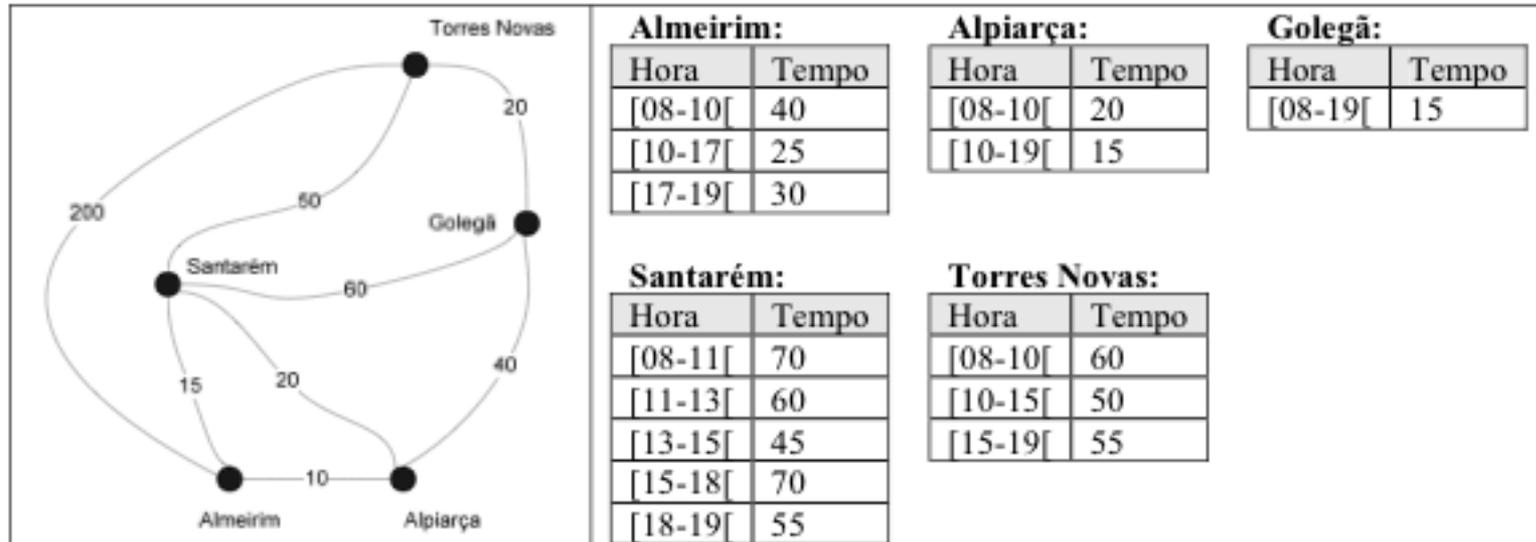
- Se as regras da charada- $n$  forem relaxadas de maneira a que uma peça se possa movimentar para **qualquer posição**, então  $h_1(n)$  dá-nos a melhor solução.
- Se as regras da charada- $n$  forem relaxadas de maneira a que uma peça se possa movimentar para **qualquer posição adjacente**, então  $h_2(n)$  dá-nos a melhor solução.

# Problema do caixeiro viajante

- Encontrar o circuito mais curto que visita todas as cidades exatamente uma vez.
- Árvore de cobertura mínima pode ser obtida em  $O(n^2)$  e é um limite inferior ao menor circuito (aberto)



# Caixeiro viajante dependente do tempo



Pretende-se partir de uma cidade, entregando produtos em cada uma das cidades, voltando ao início. O tempo de entrega nas cidades depende da hora de chegada. A viatura parte às 8 horas da manhã.

- Heurística admissível ?

# Procura informada com memória limitada

- Procura informada com memória limitada
  - Algoritmo IDA\*
  - Algoritmo recursivo de procura pelo melhor primeiro (RBFS)
  - Algoritmo A\* de memória limitada simplificado

# IDA\* - A\* por aprofundamento progressivo

- Reduzir requisitos de memória do A\*, adaptando os conceitos do aprofundamento progressivo.
  - Resulta no algoritmo IDA\*
- Em vez de usar a profundidade, usa-se o custo  $f$ , ( $g+h$ ), sendo o valor do corte o menor valor de  $f$  de um nó que excede o valor de corte da iteração anterior.

# A\* por aprofundamento progressivo

**function** IDA\*(*problem*) **returns** a solution sequence

**inputs:** *problem*, a problem

**local variables:** *f-limit*, the current *f*- COST limit  
*root*, a node

*root* ← MAKE-NODE(INITIAL-STATE[*problem*])

*f-limit* ← *f*- COST[*root*]

**loop do**

*solution, f-limit* ← DFS-CONTOUR(*root, f-limit*)

**if** *solution* is non-null **then return** *solution*

**if** *f-limit* =  $\infty$  **then return** failure; **end**

# A\* por aprofundamento progressivo

```
function DFS-CONTOUR(node, f-limit) returns a solution
sequence and a new f- COST limit
  inputs: node, a node
           f-limit, the current f- COST limit
  local variables: next-f, the f- COST limit for the next contour, initially  $\infty$ 
  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
    solution, new-f  $\leftarrow$  DFS-CONTOUR(s, f-limit)
    if solution is non-null then return solution, f-limit
    next-f  $\leftarrow$  MIN(next-f, new-f); end
  return null, next-f
```

# Propriedades do IDA\*

- Completo
- Espaço linear
- Óptimo
- Prático se os custos do passos forem unitários
- Dificuldade em lidar com custos reais, podendo acarretar grande tempo de processamento motivado por regenerações sucessivas de nós.

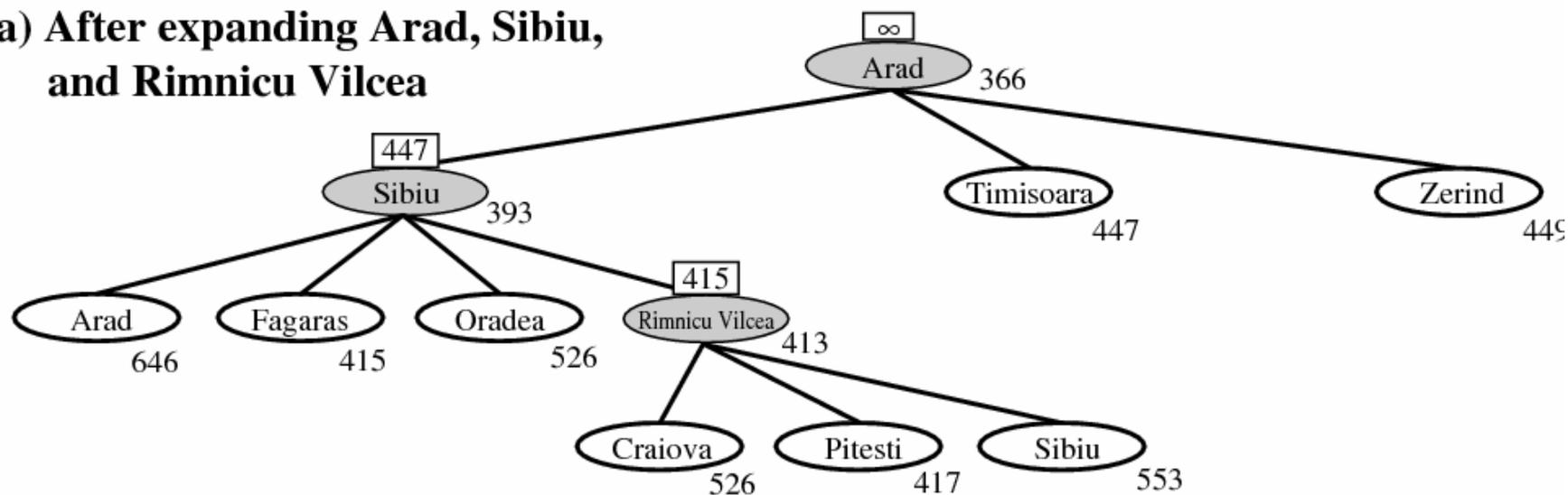
A análise de sobrecarga efectuada para as versões cegas não é válida aqui!

# RBFS - Recursive Best-First Search

- Semelhante à procura em profundidade primeiro recursiva.
- Em vez de continuar indefinidamente por um caminho, usa uma variável ( $f\_limit$ ) como registo da melhor alternativa a partir de um qualquer antecessor do nó corrente.
- Se chegar a um nó com um valor  $f$  superior a  $f\_limit$ , retrocede até à alternativa, substituindo, à medida que a recursão retrocede, o valor  $f$  de cada nó com o melhor valor de  $f$  de cada um dos seus filhos, memorizando assim o melhor valor da sub-árvore abandonada.

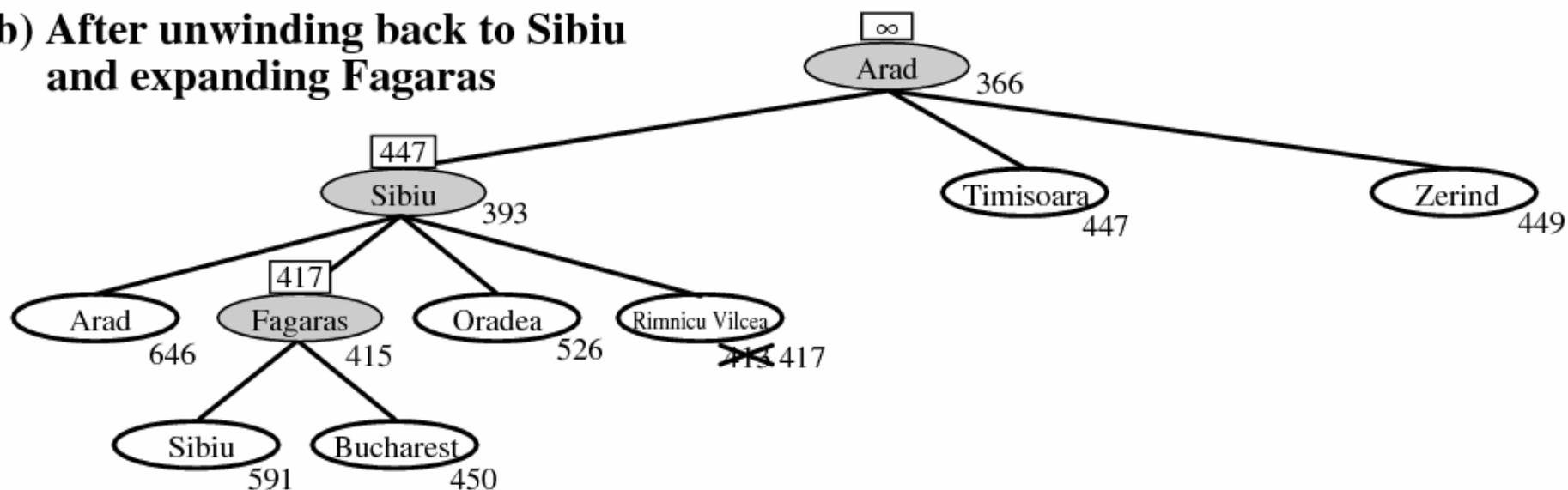
# Exemplo RBFS (1)

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



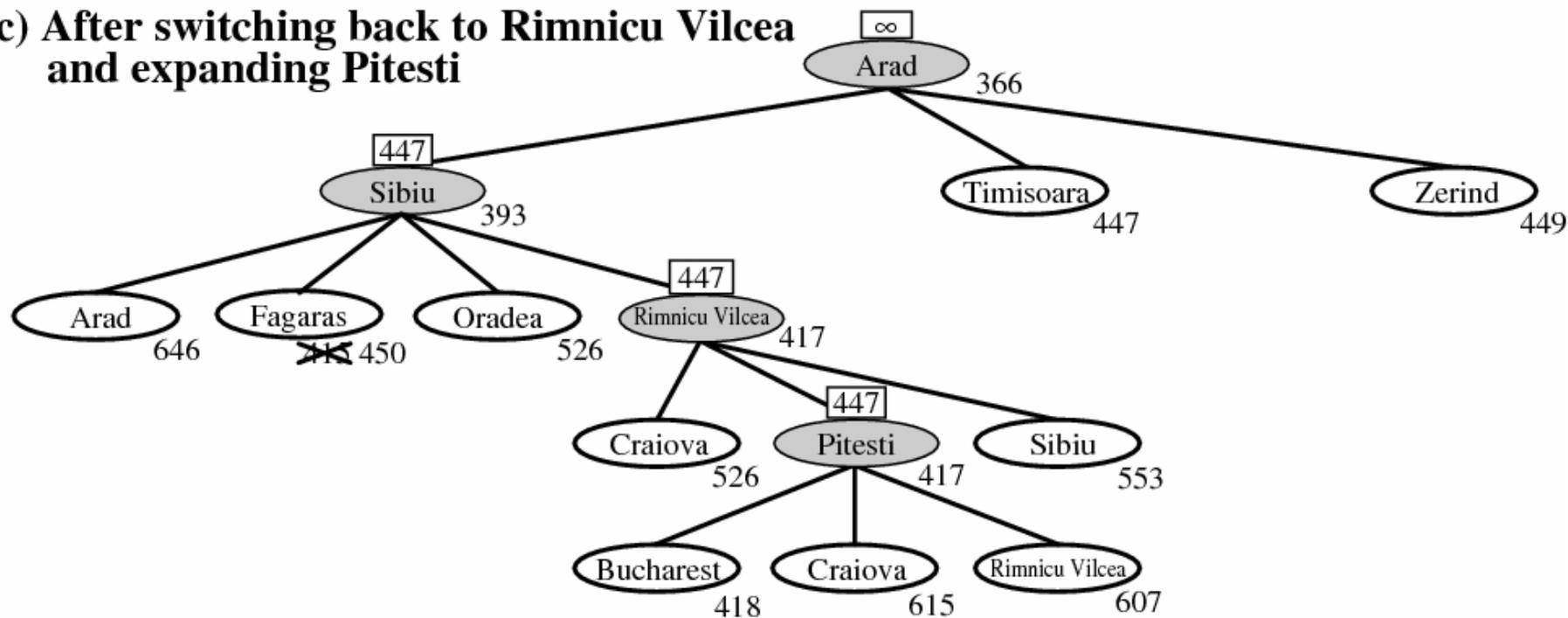
# Exemplo RBFS (2)

(b) After unwinding back to Sibiu and expanding Fagaras



# Exemplo RBFS (3)

(c) After switching back to Rimnicu Vilcea and expanding Pitesti



**function** RECURSIVE-BEST-FIRST-SEARCH(*problem*)  
**returns** a solution, or failure  
RBFS(*problem*, MAKE-NODE(INITIAL-STATE[*problem*]),  $\infty$ )

---

**function** RBFS(*problem*, *node*, *f-limit*) **returns** a solution, or failure and a new *f*-cost limit

**if** GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*

*successors*  $\leftarrow$  EXPAND(*node*, *problem*)

**if** *successors* is empty **then return** *failure*,  $\infty$

**for each** *s* **in** *successors* **do**

$f[s] \leftarrow \text{MAX}(g(s)+h(s), f[\textit{node}])$

**repeat**

*best*  $\leftarrow$  the lowest *f*-value node in *successors*

**if**  $f[\textit{best}] > \textit{f-limit}$  **then return** *failure*,  $f[\textit{best}]$

*alternative*  $\leftarrow$  the second lowest *f*-value node among *successors*

*result*,  $f[\textit{best}] \leftarrow$  RBFS(*problem*, *best*,  $\min(\textit{f-limit}, \textit{alternative})$ )

**if** *result*  $\neq$  *failure* **then return** *result*

# Propriedades do RBFS

- Completo
- Espaço linear
- Ótimo (se a heurística for admissível)
- Melhor que o IDA\* em termos de de complexidade temporal, mas difícil de caracterizar pois depende da qualidade da heurística
- Continua a ter problemas com regenerações sucessivas de nós: utiliza pouca memória...

# SMA\* - Simplified Memory-bounded A\*

- Tal como no A\*, expande-se a melhor folha até ficar com a memória cheia
- Quando a memória fica toda ocupada, esquece a folha mais antiga com o pior valor, e guarda no pai o seu valor, para possível regeneração
- Um nó só é regenerado quando todos os outros caminhos se mostrarem piores do que aqueles do nó esquecido

```

function SMA*(problem) returns a solution sequence
inputs: problem, a problem
local variables: Queue, a queue of nodes ordered by f-cost

Queue ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
  if Queue is empty then return failure
  n ← deepest least-f-cost node in Queue
  if GOAL-TEST(n) then return success
  s ← NEXT-SUCCESSOR(n)
  if s is not a goal and is at maximum depth then  $f(s) \leftarrow \infty$ 
  else  $f(s) \leftarrow \text{MAX}(f(n), g(s)+h(s))$ 
  if all of n's successors have been generated then
    update n's f-cost and those of its ancestors if necessary
  if SUCCESSORS(n) all in memory then remove n from Queue
  if memory is full then
    delete shallowest, highest-f-cost node in Queue
    remove it from its parent's successor list
    insert its parent on Queue if necessary
  insert s on Queue
end

```

# Propriedades do SMA\*

- O SMA\* utiliza *toda* a memória disponível para levar a cabo a procura
- O SMA\* é completo se existir uma solução alcançável (cujo caminho caiba em memória)
- Óptimo se existir uma solução óptima alcançável, caso contrário devolve a melhor solução cujo caminho cabe em memória
- Retira da fronteira nós superficiais com valores elevados da função de avaliação. Um nó retirado da fronteira só é regenerado se todos os irmãos forem piores do que ele.
- SMA\* é o melhor algoritmo para procurar soluções óptimas, nomeadamente quando o espaço de estados é um grafo, os custos não são uniformes e a geração de nós é mais dispendiosa do que manter listas de nós abertos e fechados.
- Mas as limitações de memória podem tornar um problema intratável...

# Outros cenários de procura

- Podem-se resolver problemas de procura “online” com ações deterministas em que se sabem as ações possíveis em cada estado, mas desconhece-se o seu efeito antes das executar.
  - Algoritmo cego: Online-DFS (assume ações reversíveis)
  - Algoritmo informado: LRTA\*
- Existem ainda outros algoritmos que permitem a alteração dos custos dos arcos em runtime (exemplo navegação robótica):
  - Dynamic A\* (D\*) e D\* Lite
- Outros algoritmos são incrementais e permitem ir melhorando a solução obtida, caso o tempo o permita:
  - ARA\* (Anytime repairing A\*)
  - AD\* (Anytime dynamic A\*) = ARA\* + D\*

# Bibliografia

- Capítulos 4.1 e 4.2 (4.5 versões online)
- Para verem as versões dinâmicas dos algoritmos consultar [http://www.ri.cmu.edu/pubs/pub\\_4975.html](http://www.ri.cmu.edu/pubs/pub_4975.html)