Procura heurística
Cap 4 (4.1,4.2 e 4.5)

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

Revisão de procura em Árvores

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

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

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*

Ripar vídeos (bin-packing)

- Suponha-se que existe um conjunto de **n** vídeos de dimensões dadas por uma função **size** e DVDs de tamanho fixo **V**.
- Assume-se que qualquer vídeo cabe num DVD (dimensão ≤ V)
- Qual o menor número de DVDs necessários?
- Problema NP-difícil…
- Existem algoritmos que aproximam a solução óptima com erro percentual fornecido, para números suficientemente grandes.
- No pior caso necessitaremos de

  ceiling(n*max size/V) DVDs.

- No melhor caso necessitaremos de

  ceiling(dimensão total dos vídeos/V).

Representação do problema

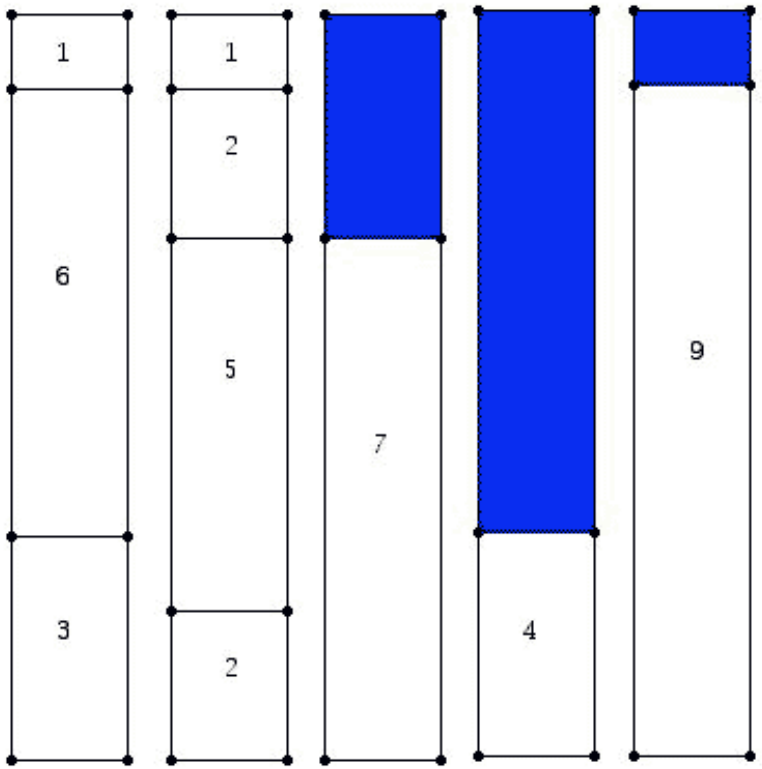
- Suponhamos que temos 9 vídeos de dimensão 3, 6, 2, 1, 5, 7, 2, 4, 1, e 9Gb e DVDs de 10 Gb.
- Um estado é representado por uma sequência numerada de DVDs com pelo menos um vídeo
- O estado inicial é uma sequência vazia
- Teste objectivo: todos os vídeos colocados
- Os operadores são
  - adicionar um vídeo a um DVD existente (caso caiba lá) com custo 0.
  - colocar um novo DVD no final do conjunto, contendo o vídeo (custo 1).

Solução por procura pelo melhor primeiro

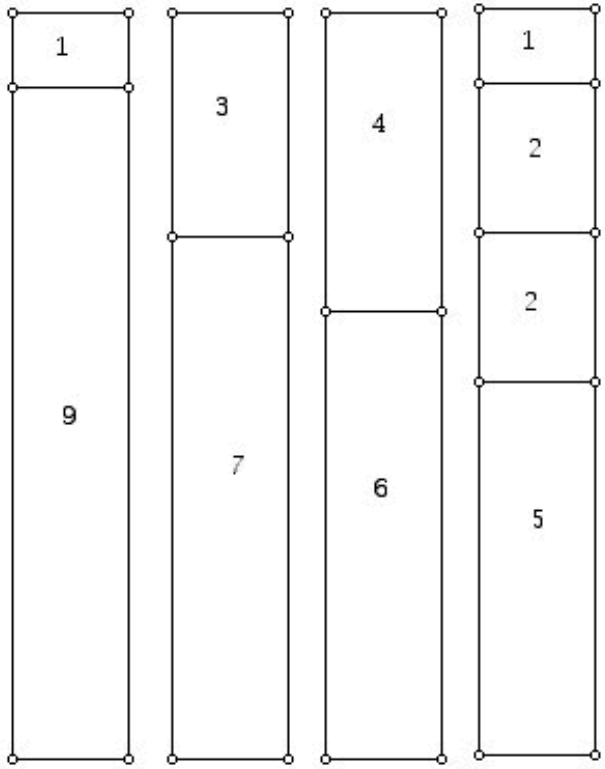
- Função de avaliação
  - Índice onde foi colocado o último vídeo (First Fit) + (n+1)*(n - # já colocados)
  - Espaço livre no DVD ocupado com último vídeo (Best Fit) + (V+1)*(n - # já colocados)
- Se os vídeos são colocados por ordem decrescente de dimensão, temos as versões First Fit Decreasing e Best Fit Decreasing.
- Os métodos *First Fit* e *Best Fit* obtêm uma solução com o custo no pior caso de ceiling(1,7 * OPT).
- Os métodos *First Fit Decreasing* e *Best Fit Decreasing* obtêm uma solução com custo no pior caso de ceiling(11/9 OPT) + 1 ≈ ceiling(1,22 OPT) + 1
- O algoritmo é polinomial (porquê ?)

Soluções (AMS)

Sequência 3, 6, 2, 1, 5, 7, 2, 4, 1, 9



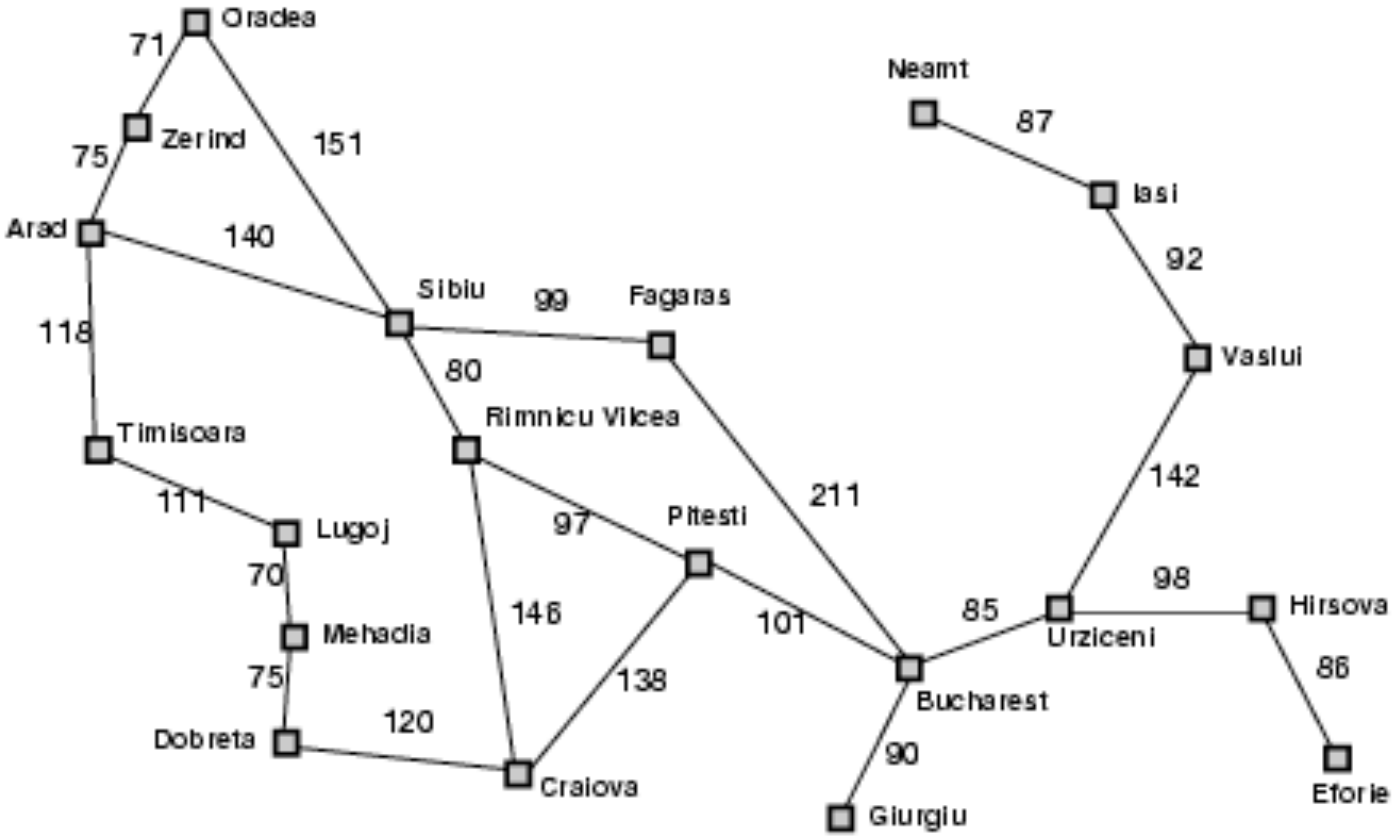
First Fit



Best Fit Decreasing

[http://www.ams.org/featurecolumn/archive/bins1.html]

Roménia com distâncias em linha recta km

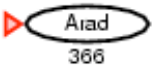


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Procura sôfrega

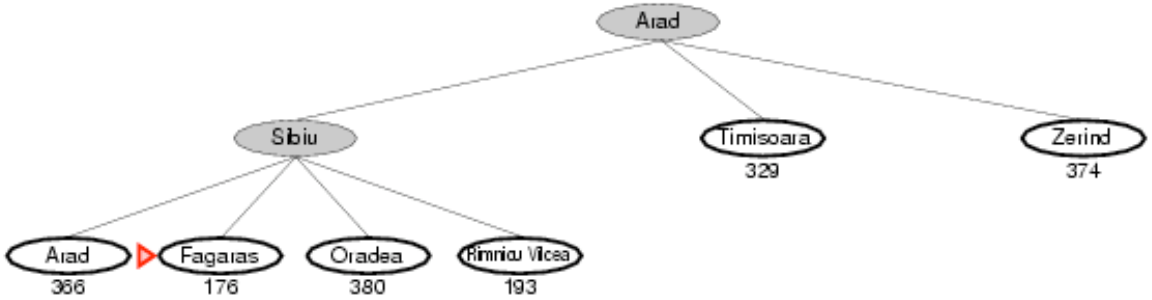
- Função de avaliação $f(n) = h(n)$ (heurística)
- = estimativa do custo do menor caminho para ir de n até a um estado objectivo
  - $h_{SLD}(n)$ = distância em linha recta de n até Bucareste
- Procura sôfrega expande o nó que *aparenta* estar mais próximo do objectivo

Exemplo de procura sôfrega

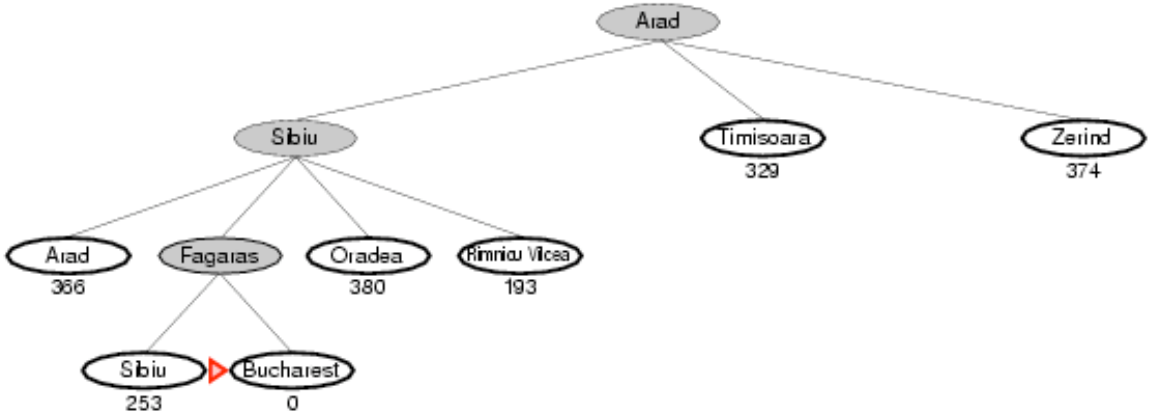


Exemplo de procura sôfrega

Exemplo de procura sôfrega



Exemplo de procura sôfrega



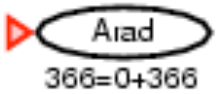
Propriedades da procura sôfrega

- Completa? Não – pode ficar presa em ciclos, e.g., com Oradea como objectivo 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*

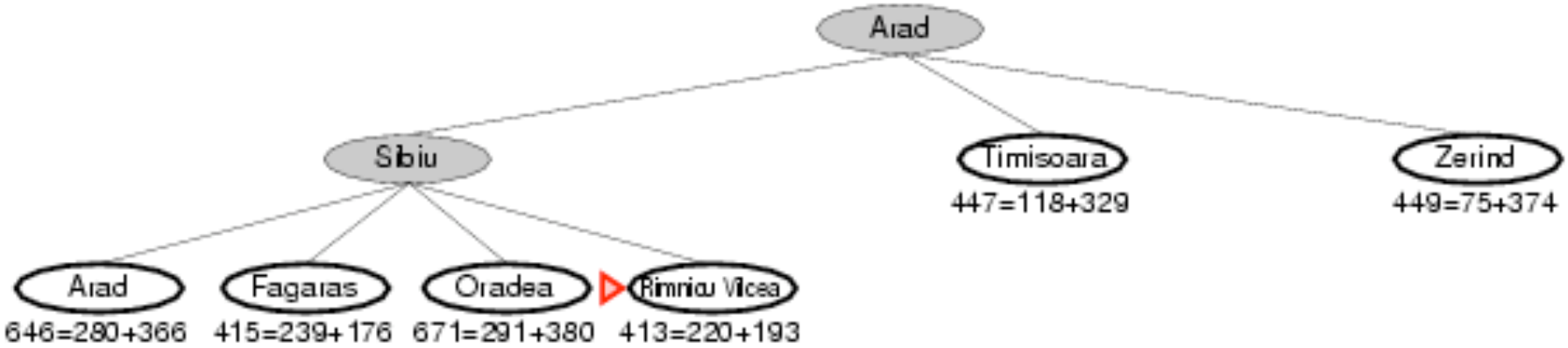
- Ideia: evitar expandir caminhos que já têm elevado custo
- Função de avaliação $f(n) = g(n) + h(n)$
  - $g(n)$ = custo actual para atingir n
  - $h(n)$ = custo estimado para atingir o objectivo a partir de n
  - $f(n)$ = custo total estimado do caminho até ao objectivo passando por n

Exemplo de Procura A*

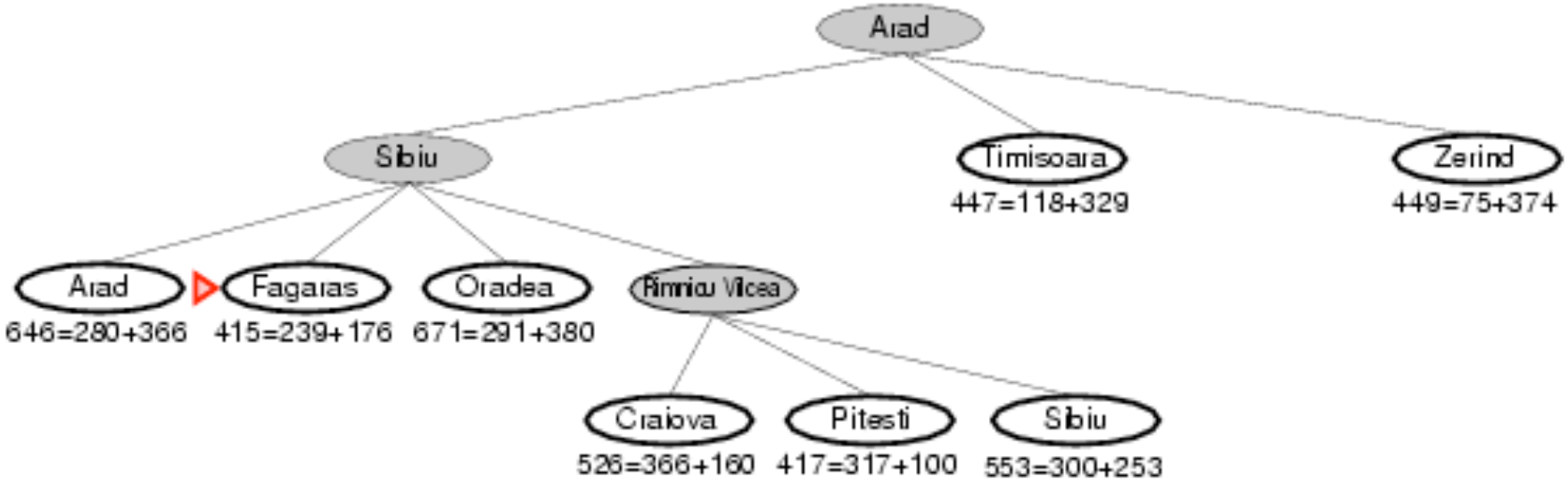


Exemplo de Procura A*

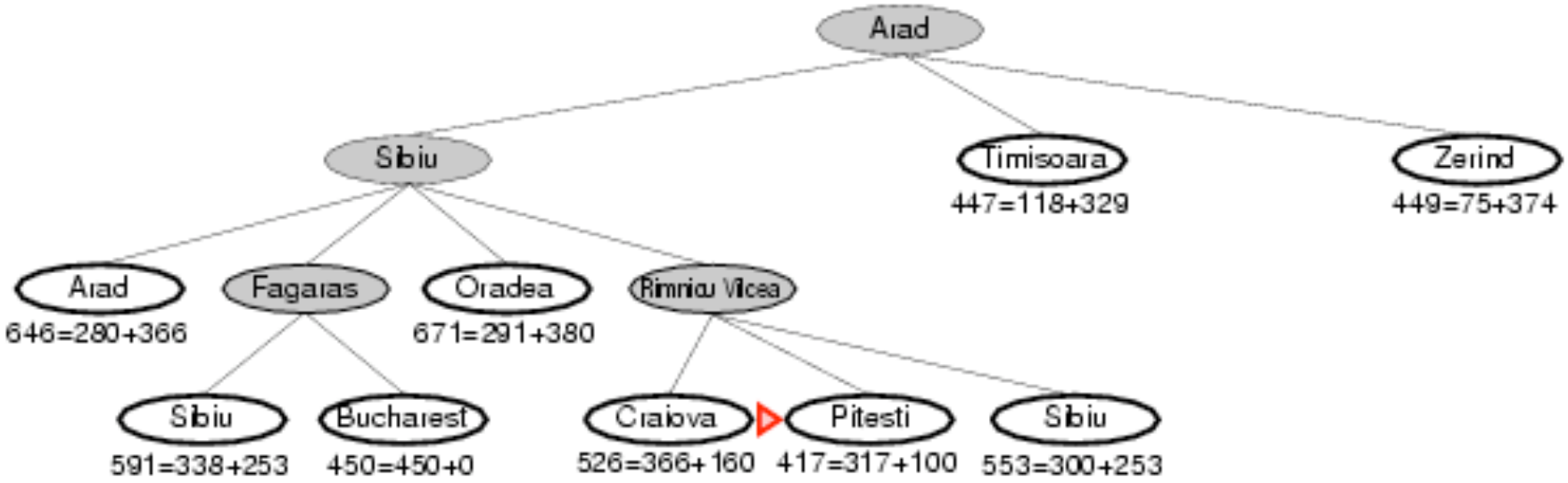
Exemplo de Procura A*



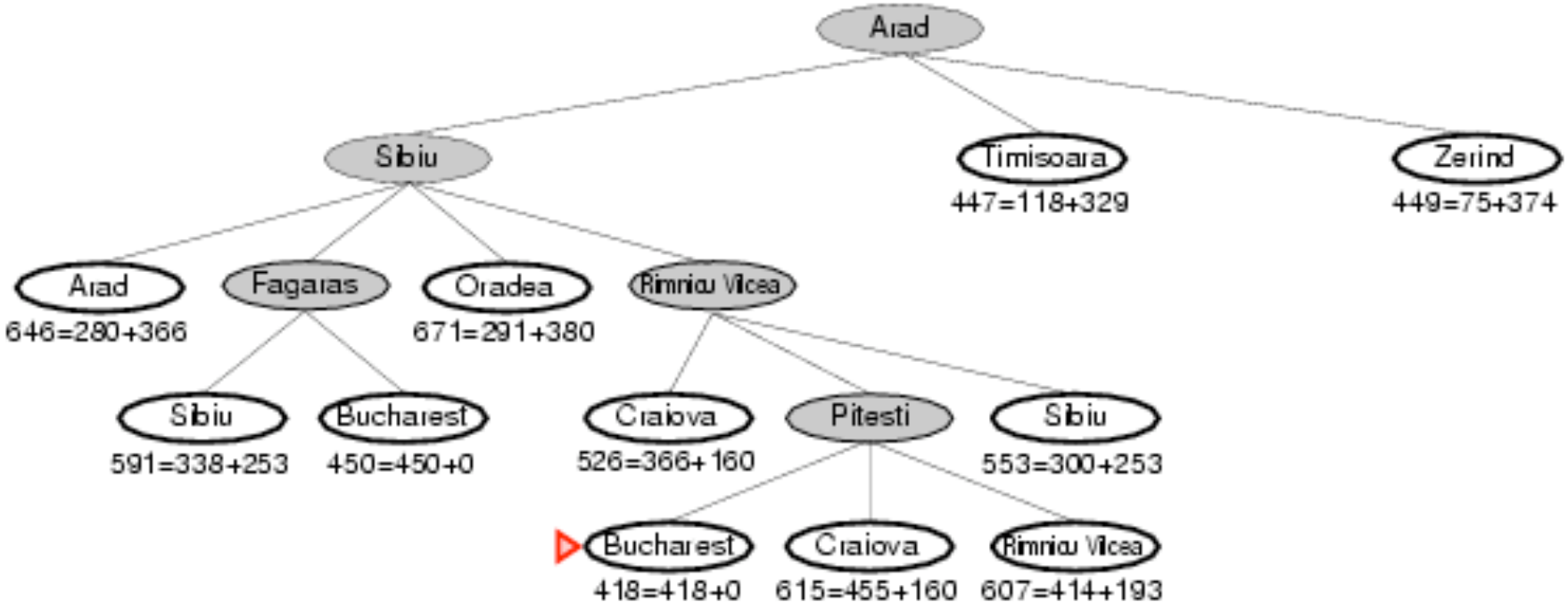
Exemplo de Procura A*



Exemplo de Procura A*



Exemplo de Procura A*



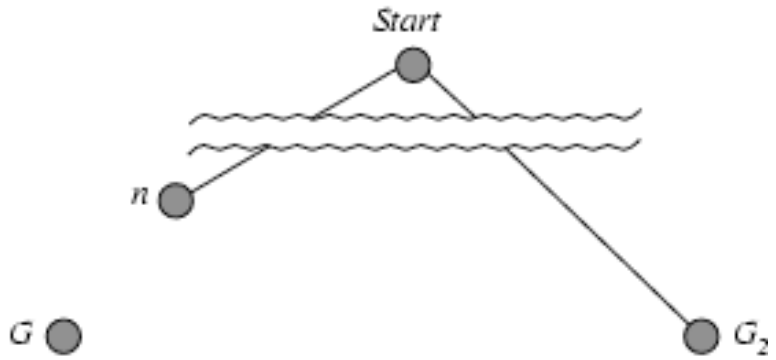
Heurísticas admissíveis

- 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)

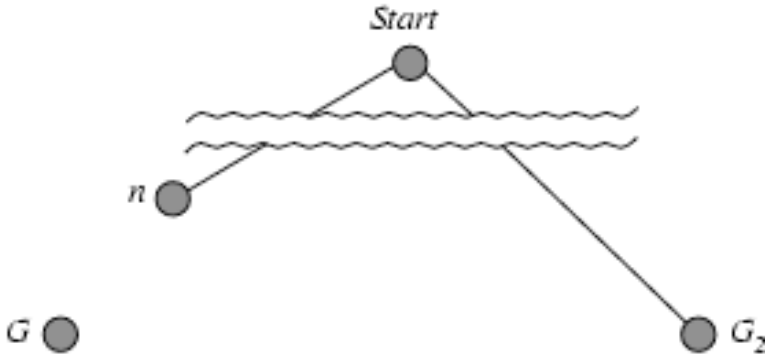
- Suponha-se que um estado final subóptimo G_2 foi gerado e encontra-se na fronteira. Seja *n* um nó por expandir na fronteira num caminho mais curto para o objectivo óptimo *G*.



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

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*.

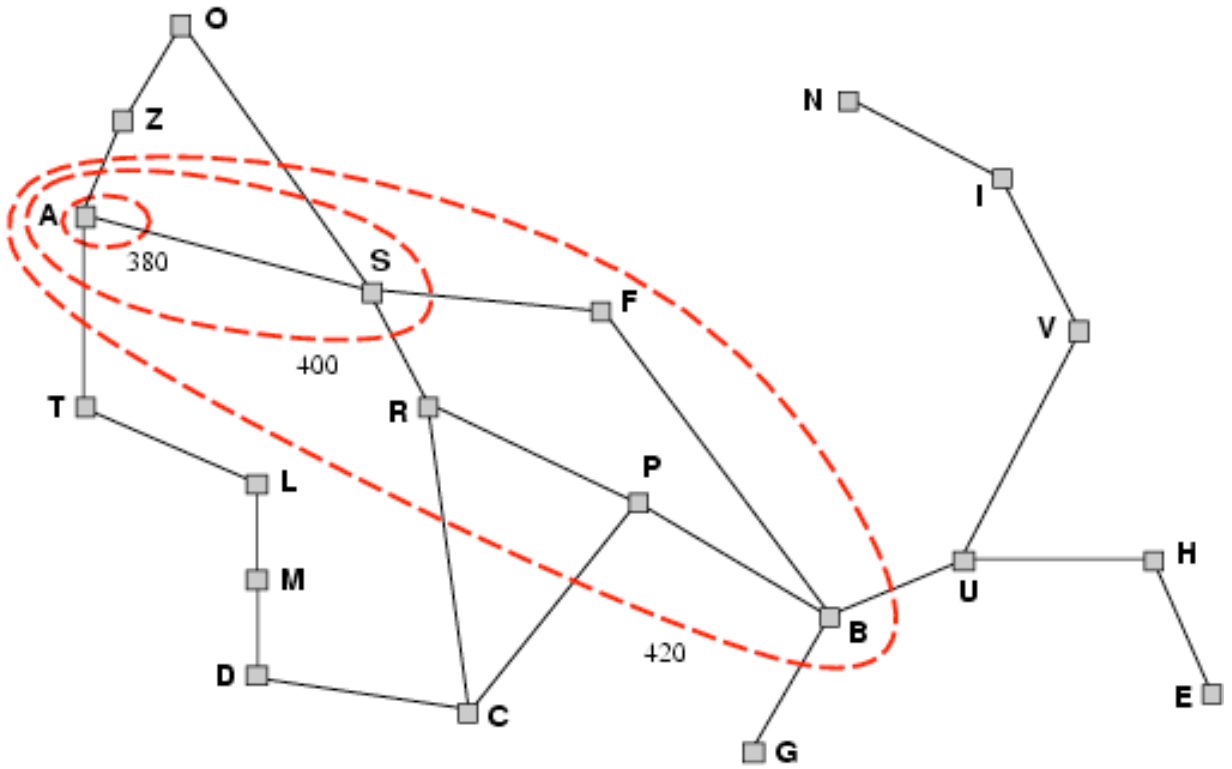


- $f(G_2)$ > $f(G)$ como se viu anteriormente
- $h(n)$ ≤ $h^*(n)$ porque h é admissível
- $g(n) + h(n)$ ≤ $g(n) + h^*(n)$
- $f(n)$ ≤ $f(G)$

Portanto $f(G_2) > f(n)$, e o A* nunca seleccionará 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:

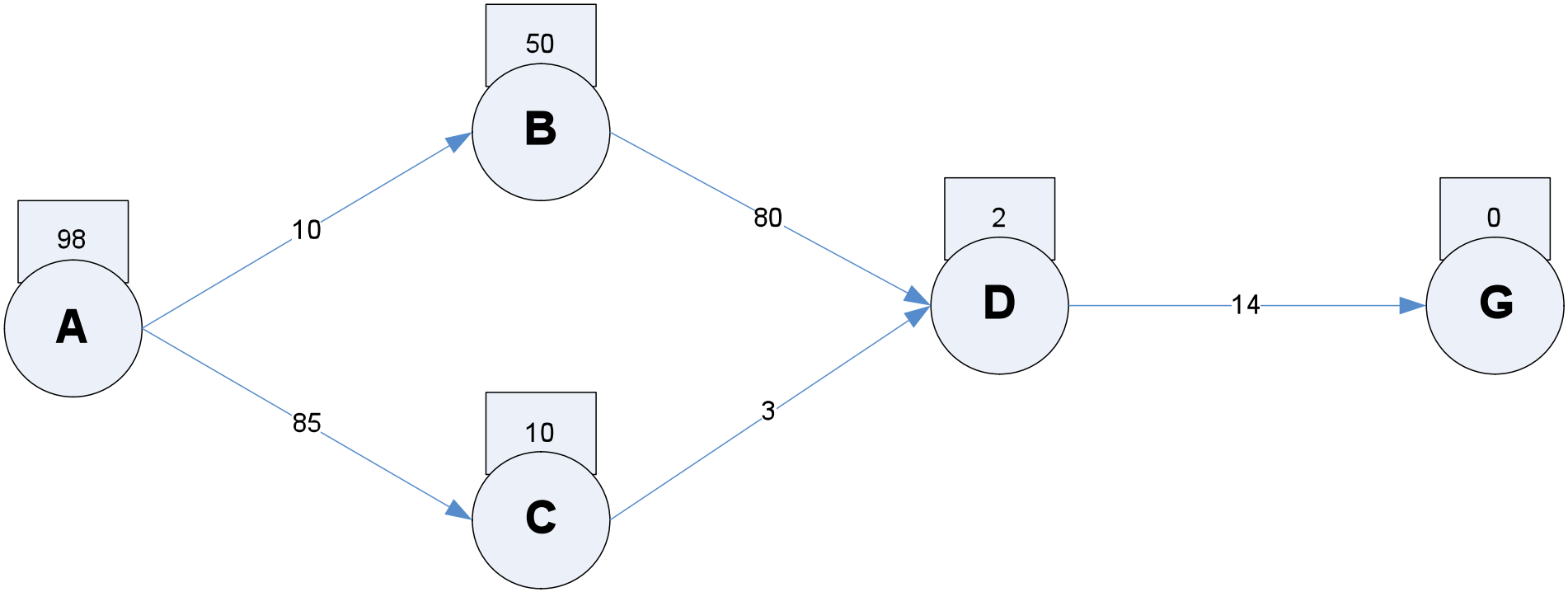
- Não há outro algoritmo **óptimo** que garantidamente expanda um menor número de nós!

Propriedades do A*

- Completo? Sim (a não ser que haja um número infinito de nós com f ≤ *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
- Óptimo? Sim, se a heurística for admissível

Problemas da versão naïve do A* com procura em grafos



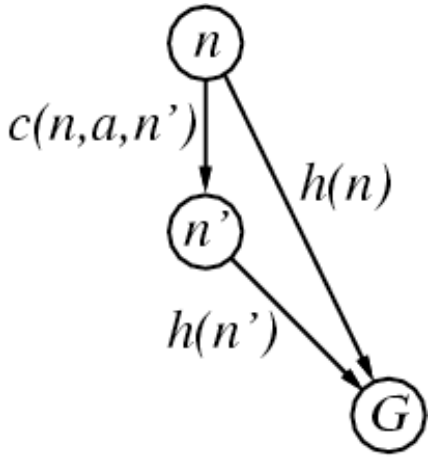
fronteira					
A(98)	B(60) C(95)	D(92) C(95)	C(95) G(104)	D(90) G(104)	G(104)
explorados	A	A B	A B D	A B C D	A B C D

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 *e* todo o seu sucessor *n', gerado pela acção 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-

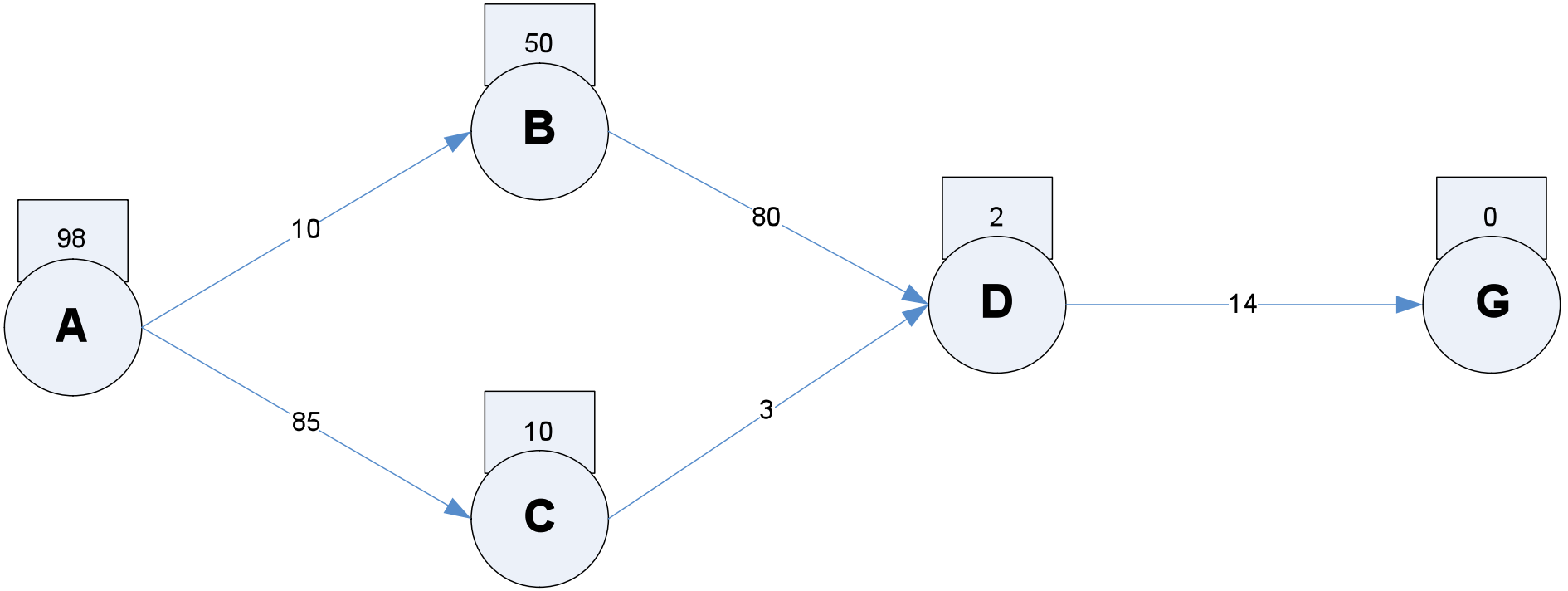
O que fazer com heurísticas inconsistentes?

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*.

A* com procura em grafos (corrigido)



…	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)

Estimativa PathMax

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

- A ideia consiste em utilizar como valor da função de avaliação

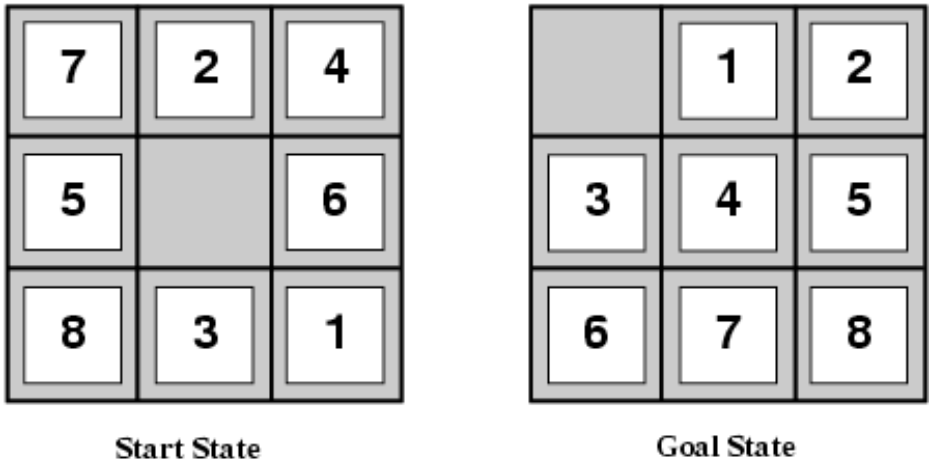
  $f\hat{}(m) = \max \{f(n); f(m)\}$

- em que *m* é sucessor de *n*. O valor de $f\hat{}(m)$ 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.

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
  - $h_1(S) = 7$
- $h_2(n)$ = soma das distâncias de Manhattan
  - $h_2(S) = 4+0+3+3+1+0+2+1=14$

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 ≈ 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 efectivo b* caracteriza a qualidade da heurística utilizada. O valor b* é obtido resolvendo a equação* $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$, *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 exacto de uma solução óptima para o problema relaxado é uma heurística admissível para o problema original!

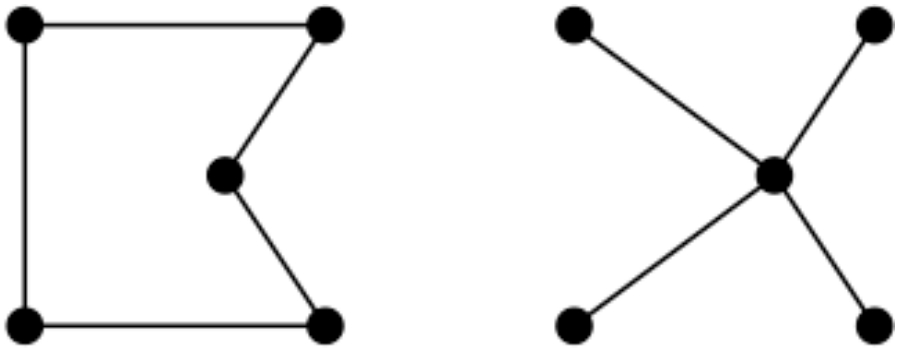
Considere-se a charada-*n* novamente

- 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.

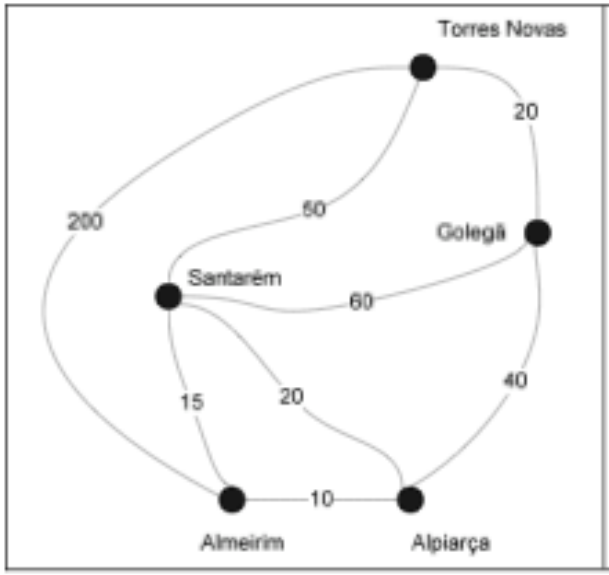
- Se as regras da charada-*n* forem relaxadas de maneira a que uma peça se possa movimentar para *qualquer* posição adjacente, então $h_2(n)$ dá-nos a melhor solução.

Problema do caixeiro viajante

- Encontrar o circuito mais curto que visita todas as cidades exactamente 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



Almeirim:

Hora	Tempo
[08-10[	40
[10-17[	25
[17-19[	30

Alpiarça:

Hora	Tempo
[08-10[	20
[10-19[	15

Golegã:

Hora	Tempo
[08-19[	15

Santarém:

Hora	Tempo
[08-11[	70
[11-13[	60
[13-15[	45
[15-18[	70
[18-19[	55

Torres Novas:

Hora	Tempo
[08-10[	60
[10-15[	50
[15-19[	55

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

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 = ∞ 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 ∞

    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 ← DFS-CONTOUR(s, f-limit)
        if solution is non-null then return solution, f-limit
        next-f ← 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!

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure

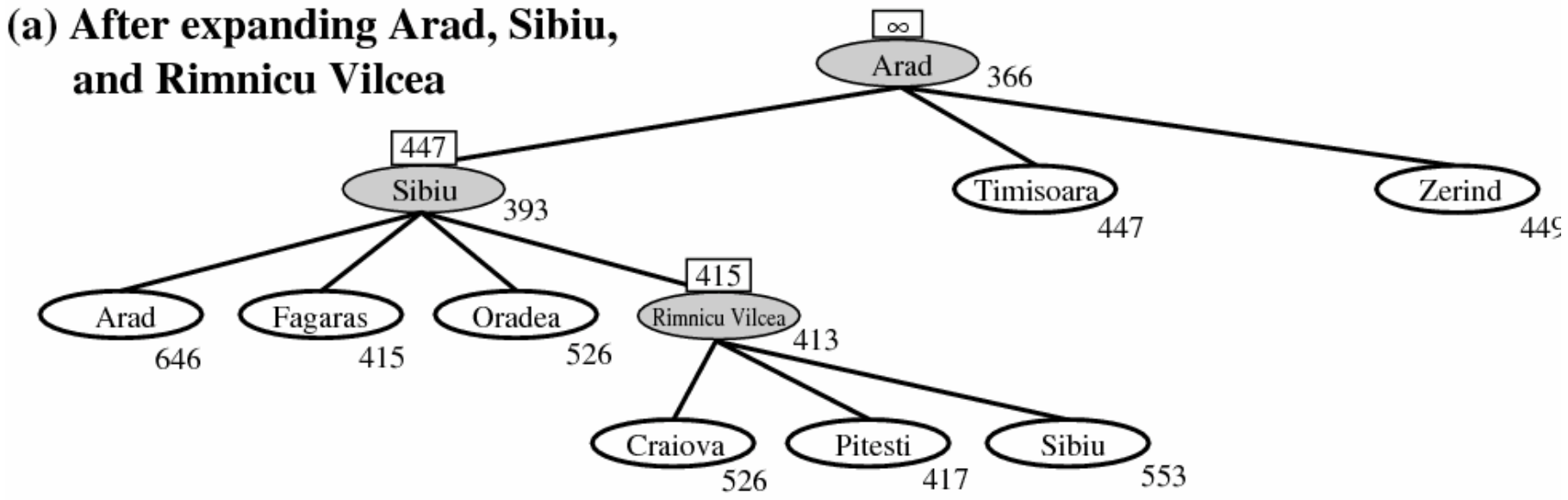
RBFS(*problem*,MAKE-NODE(INITIAL-STATE[*problem*]),∞)

---

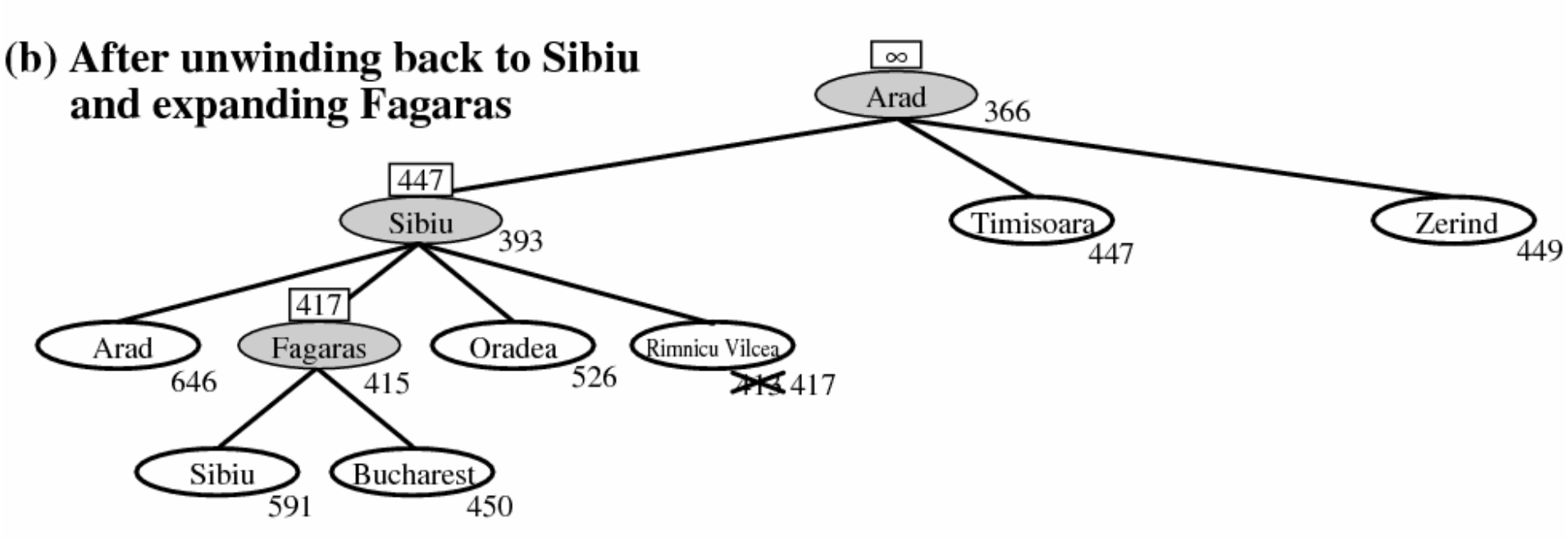
function RBFS(*problem*,*node*, *f-limit*) **returns** a solution, or failure and a new f-cost limit

if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
successors $\leftarrow$ EXPAND(*node*,*problem*)
if *successors* is empty **then return** *failure*,∞
for each *s* **in** *successors* **do**
f[*s*] $\leftarrow$ MAX(g(*s*)+h(*s*), f[*node*])
repeat
best $\leftarrow$ the lowest f-value node in *successors*
if f[*best*] $>$ *f-limit* **then return** failure, f[*best*]
alternative $\leftarrow$ the second lowest f-value node among *successors*
result, f[*best*] $\leftarrow$ RBFS(*problem*, *best*, min(*f-limit*,alternative))
if *result*$\neq$*failure* **then return** *result*

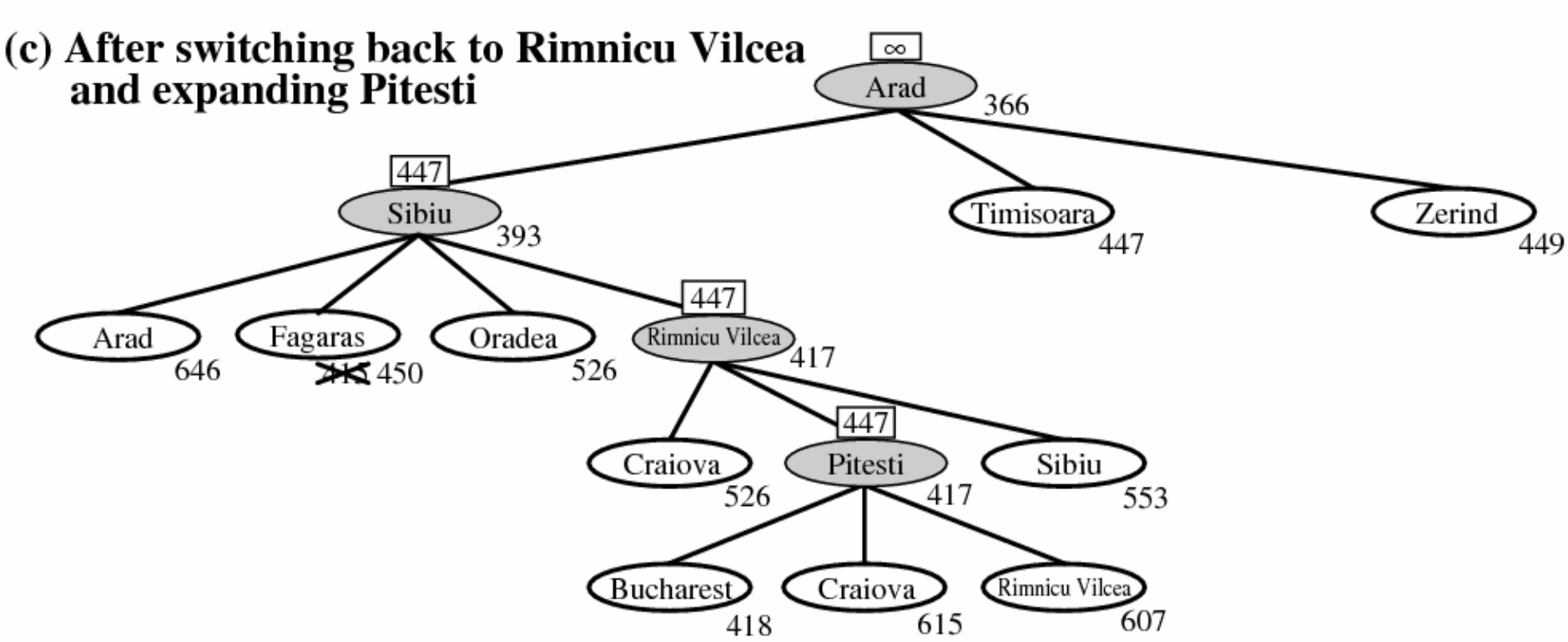
Exemplo RBFS (1)



Exemplo RBFS (2)



Exemplo RBFS (3)



Propriedades do RBFS

- Completo
- Espaço linear
- Óptimo (se a heurística for admissível)
- Melhor que o IDA*
- Continua a ter problemas com regenerações sucessivas de nós: utiliza pouca memória...

SMA*

- 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 $\leftarrow$ MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
loop do
  if *Queue* is empty **then return** failure
  $n \leftarrow$ deepest least-f-cost node in *Queue*
  if GOAL-TEST(n) **then return** success
  $s \leftarrow$ NEXT-SUCCESSOR(n)
  if s is not a goal and is at maximum depth **then** f(s) $\leftarrow \infty$
  else f(s) $\leftarrow$ 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 acções deterministas em que se sabem as acções possíveis em cada estado, mas se desconhece o seu efeito antes das executar.
  - Algoritmo cego: Online-DFS (assume acções 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