ESTADOS E BUSCAS II

Prof. Dr. Igor da Penha Natal

Busca Informada - Heurística

Busca informada (heurística)

- Porque todas as estratégias anteriores são chamadas de **desinformadas**?
  - Porque elas não consideram qualquer informação sobre os estados para decidir qual caminho expandir em primeiro lugar na fronteira.
- Em outras palavras, elas são gerais e não levam em conta a **natureza específica do problema**.

Busca informada (heurística)

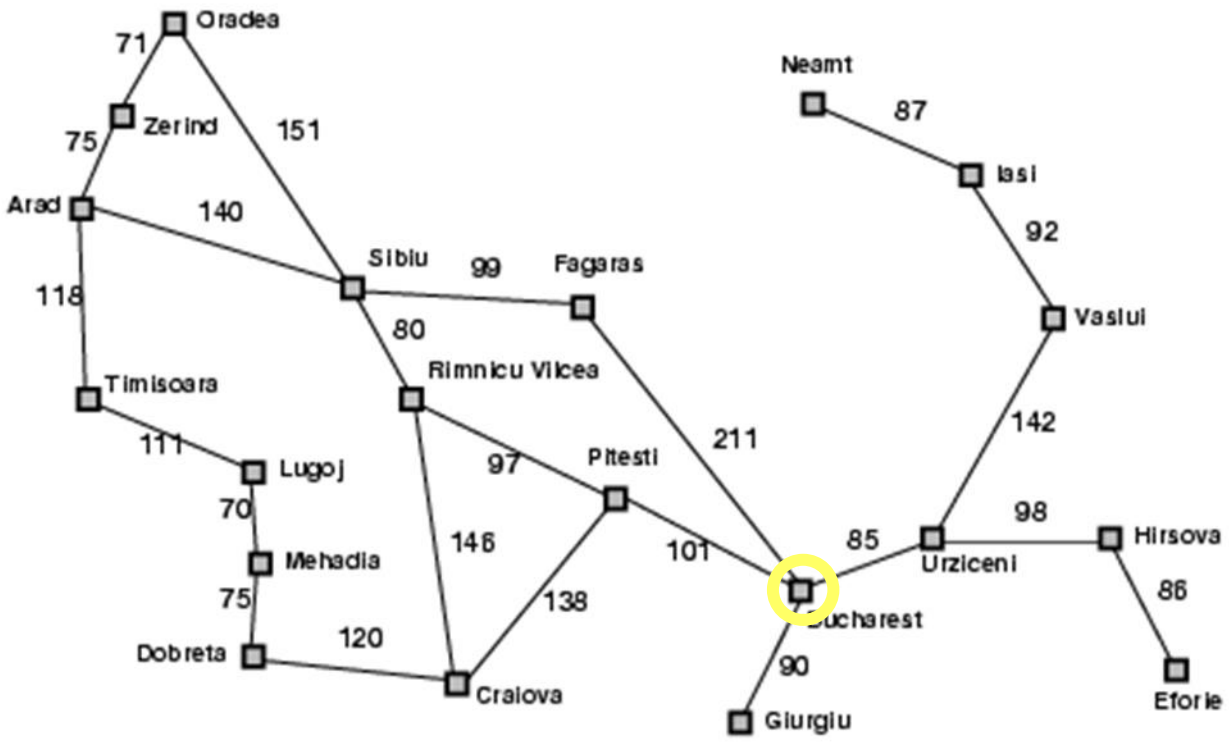
- **Idéia**: não ignore o objetivo ao selecionar os caminhos.
- Muitas vezes há conhecimento extra que pode ser usado para orientar a busca: uma **estimativa** da distância do nó ***n*** para um nó objetivo, por exemplo.
- Uma **heurística**, ***h(n)***, é uma estimativa do custo do caminho mais curto do nó ***n*** para um nó objetivo.
  - ***h*(*n*)** usa apenas informações que podem ser facilmente obtidas (que são fáceis de calcular) sobre um nó.
  - ***h*** pode ser estendido para caminhos: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
  - ***h*(*n*)** é uma subavaliação se não houver nenhum caminho de ***n*** para um nó objetivo que tenha comprimento inferior a ***h*(*n*)**.

Funções Heurísticas

- Definição de **função heurística admissível:**
  - Uma função heurística ***h(n)*** é admissível se ela nunca é uma sobrestimação do custo do nó ***n*** para um nó objetivo.
- As heurísticas admissíveis tem natureza **otimista**, pois elas sempre indicam que o custo da solução é melhor do que ele realmente é.
  - Desta maneira se uma solução ainda não foi encontrada sempre existirá um nó com *custo* menor do que ela.
- Nunca existe um caminho do nó ***n*** para um nó objetivo que tenha caminho de comprimento inferior a ***h(n)***. Outra maneira de dizer isto:
  - ***h(n)*** é um limite inferior sobre o custo de ir do nó ***n*** para o nó objetivo mais próximo.

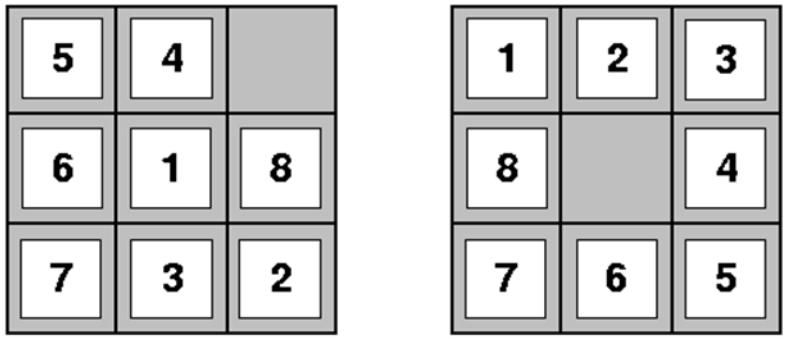
Exemplo de funções heurísticas admissíveis

- Se os nós são pontos em um plano Euclidiano e o custo é a distância, então podemos usar a distância linear entre o nó ***n*** e o nó objetivo mais próximo como o valor da ***h(n)***.



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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Exemplo de funções heurísticas admissíveis



- Exemplo de heurísticas para o quebra-cabeças de 8 peças
  - $h_1(n)$ = número de quadrados em locais errados
  - $h_2(n)$ = distância Manhattan total
    - número de espaços na vertical e horizontal que devem ser movidos para chegar no local correto.
  - $h_1(s) = 7$ (só o número 7 está no local correto)
  - $h_2(s) = 2+3+3+2+4+2+0+2 = 18$

Como construir uma heurística

- Você deve identificar uma **versão relaxada do problema**:
  - Na qual uma ou mais restrições sobre as ações foram descartadas.

- **Exemplo**:
  - Robô: o agente **pode mover-se através das paredes.**
  - Motorista: **o agente pode mover-se em linha reta ao local.**
  - 8-*puzzle*:
    - 1) os elementos **podem mover para qualquer lugar.**
    - 2) os elementos **podem mover para qualquer casa adjacente.**

- **Resultado**:
  - O custo de uma solução ideal para o problema relaxado é uma heurística admissível para o problema original (porque sempre é fracamente menos custoso resolver um problema menos restrito!)

Como construir uma heurística

- Você deve identificar restrições que, quando removidas, fazem o problema extremamente fácil de ser resolvido.
  - Isso é importante porque as heurísticas não são úteis se eles são tão difíceis de resolver quanto o problema original!

- Este é o caso em nossos exemplos:
  - Robô: ***permitindo*** que o agente se mova através das paredes, a solução ideal para este problema relaxado é a **distância Manhattan.**
  - Motorista: ***permitindo*** que o agente se mova em linha reta ao local, a solução ideal para este problema relaxado é **distância em linha reta**.
  - 8-*puzzle*: (1) se os elementos podem mover-se para qualquer lugar, a solução ideal para esse problema relaxado é o **número de peças no local errado**; (2) se os elementos podem mover-se para qualquer local adjacente ...

Heurística: dominância

- Se $h_2(n) \geq h_1(n)$ para todos os ***n*** (sendo ambas admissíveis) então h_2 **domina** h_1.
  - Qual é a melhor para a busca (porque?)

- **8-*puzzle***:
  - (1) os elementos podem mover-se para qualquer lugar.
  - (2) os elementos podem mover-se para qualquer casa adjacente.

- No problema original os elementos podem mover-se para um quadrado adjacentes se ele estiver vazio.

- Custos da busca para o 8-*puzzle* (número médio de caminhos expandidos):

d = 12	IDS = 3,644,035 caminhos A*(h_1) = 227 caminhos A*(h_2) = 73 caminhos	onde *d* é a profundidade da solução
d = 24	IDS = caminhos demais A*(h_1) = 39,135 caminhos A*(h_2) = 1,641 caminhos	

Heurística: dominância

- Se $h_2(n) \geq h_1(n)$ para todos os ***n*** (sendo ambas admissíveis) então h_2 **domina** h_1.
  - h_2 é melhor para a pesquisa
- **8-*puzzle***:
  - (1) os elementos podem mover-se para qualquer lugar.
  - (2) os elementos podem mover-se para qualquer casa adjacente.
- No problema original os elementos podem mover-se para um quadrado adjacentes se ele estiver vazio.
- Custos da busca para o 8-*puzzle* (número médio de caminhos expandidos):

d = 12	IDS = 3,644,035 caminhos A*(h_1) = 227 caminhos A*(h_2) = 73 caminhos	onde *d* é a profundidade da solução
d = 24	IDS = caminhos demais A*(h_1) = 39,135 caminhos A*(h_2) = 1,641 caminhos	

Combinando heurísticas

- **Como combinar heurística quando não há nenhuma dominância?**

- Se $h_1(n)$ é admissível e $h_2(n)$ também é admissível, então $h(n) = \max(h_1, h_2)$ também é admissível e domina todos seus componentes.

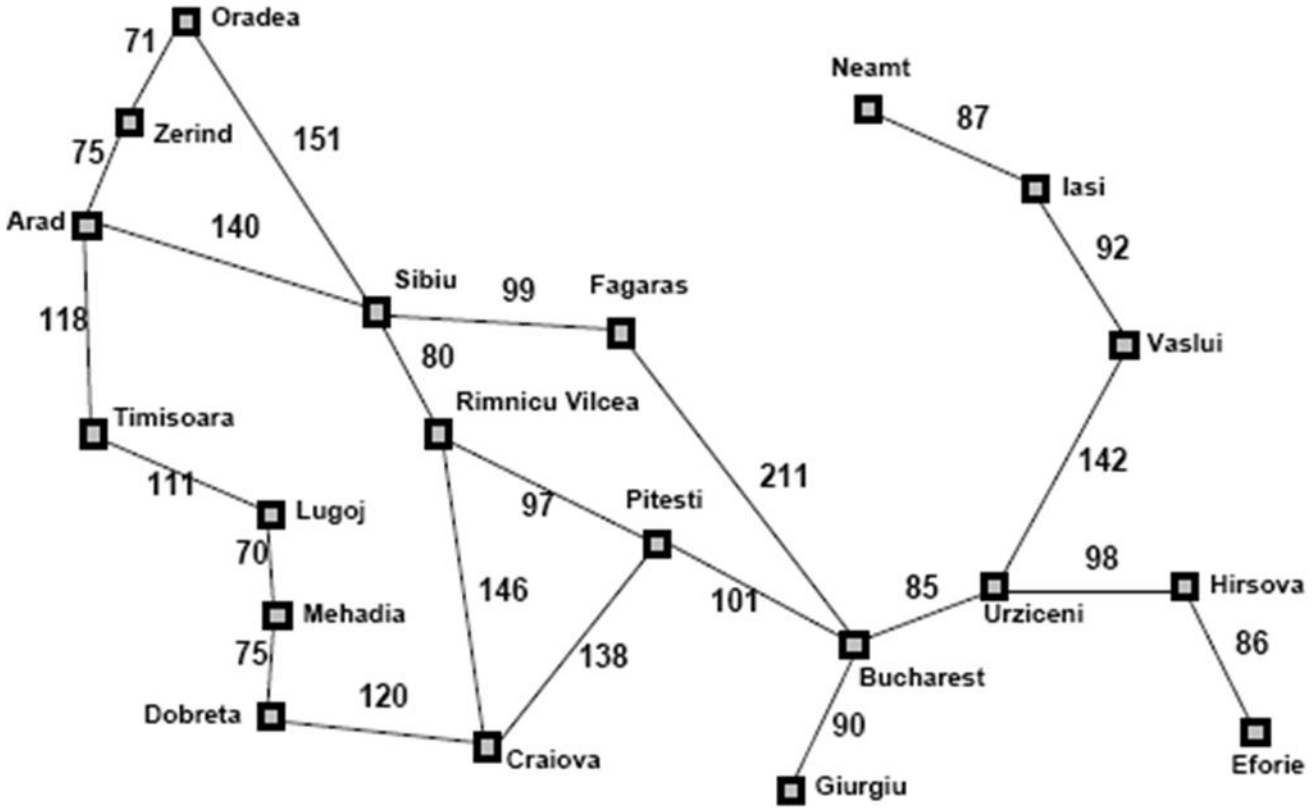
Busca heurística sistemática

- **Busca informada**
  - Busca pelo melhor-primeiro (*greedy* – gulosa)
  - Busca A*
  - Busca em aprofundamento por enumeração e poda

Busca pelo melhor-primeiro (*greedy*)

- **Ideia:** selecione o caminho cujo fim está mais próximo a um nó objetivo de acordo com a função heurística.
- A busca pelo melhor-primeiro seleciona um caminho na fronteira com o valor ***h*** mínimo.
- Ela trata a fronteira como uma ***fila de prioridade*** ordenada por ***h(n)***.
- É um algoritmo **guloso** que seleciona o melhor caminho local.

Gráfico ilustrativo – busca pelo melhor-primeiro



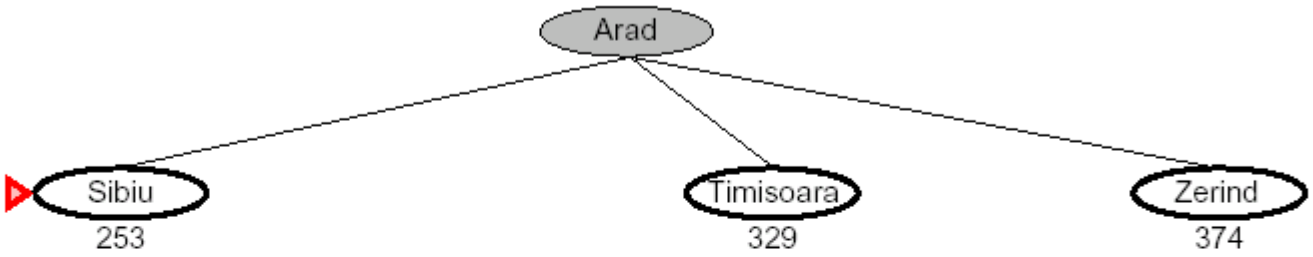
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Exemplo: busca pelo melhor primeiro (gulosa)

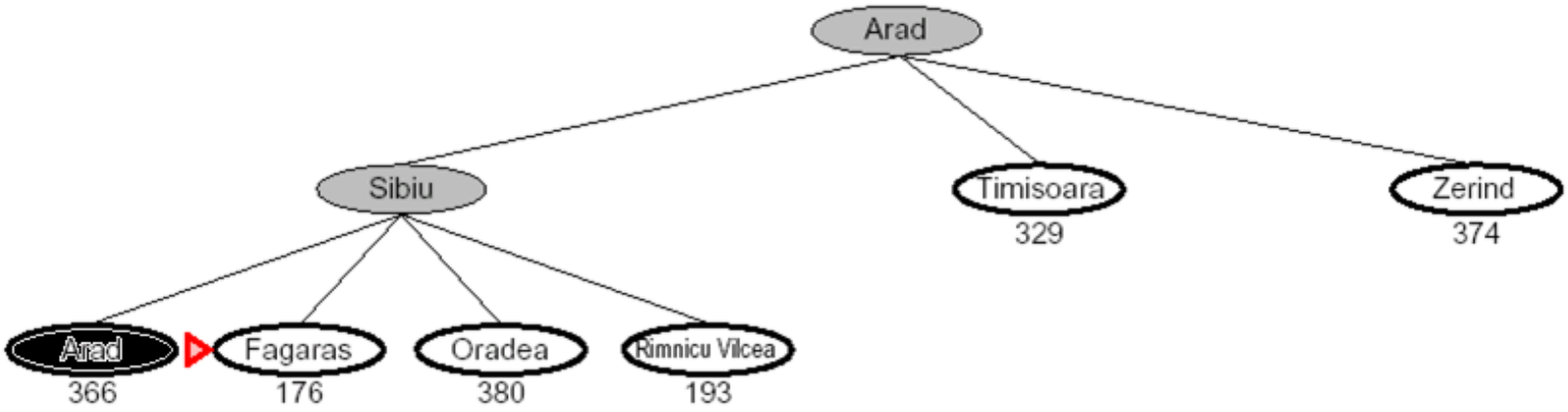
- ***Estado Inicial***: Em(Arad) = Arad
- ***Estado Final***: Em(Bucharest) = Bucharest
- ***Obs.***: Nós que podem gerar ciclos são marcados com *, e não são inseridos na fronteira

Exemplo: busca pelo melhor primeiro (gulosa)

Exemplo: busca pelo melhor primeiro (gulosa)

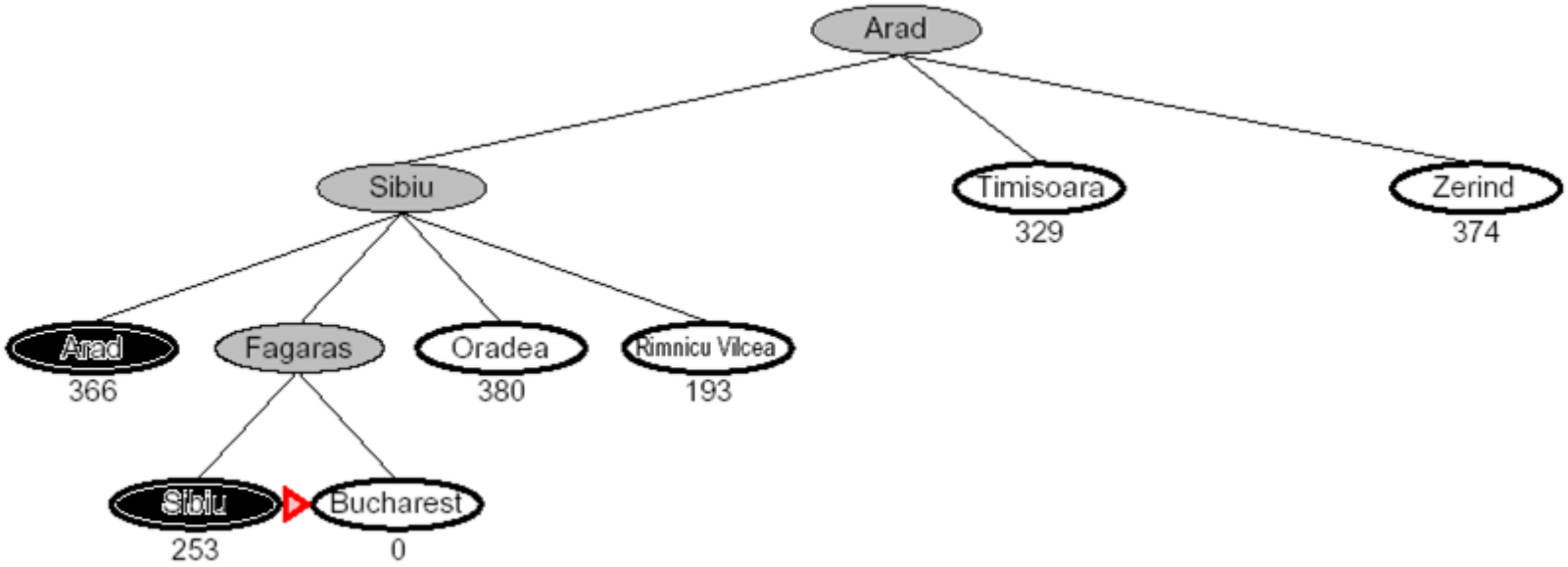


Exemplo: busca pelo melhor primeiro (gulosa)



- Nós em cinza já foram expandidos
- Nós em preto não são inseridos por já estarem no caminho.

Exemplo: busca pelo melhor primeiro (gulosa)



- O solução encontrada não é a solução ótima.
- A solução ótima passa por *Rimnicu Vilcea* e *Pitest* e tem 32 km a menos.

Busca pelo melhor primeiro (gulosa)

def busca_pelo_melhor_primeiro(G, S, *goal*):

entrada: G # a grafo
S # um conjunto de nós iniciais
objetivo # função booleana que testa se *S* é um nó objetivo

saída: um caminho de um membro de **S** para um nó para o qual a função ***objetivo*** é verdadeira, ou ⊥ se não existir solução.

local: *fronteira* # um conjunto de caminhos

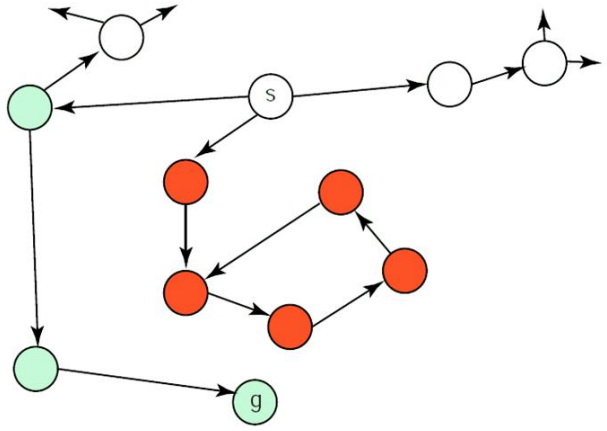
fronteira ← {<s>|s ∈ S}
while *fronteira* :
selecione e **remova** o **1º** nó $\langle s_0,\ldots,s_k\rangle$ da *fronteira*

if *objetivo*(s_k): **return** $\langle s_0,\ldots,s_k\rangle$

adicione(ordem crescente de *h(n)*, fronteira, $\{\langle s_0,\ldots,s_k,s\rangle | \langle s_k,s\rangle \in S\}$)
return ⊥

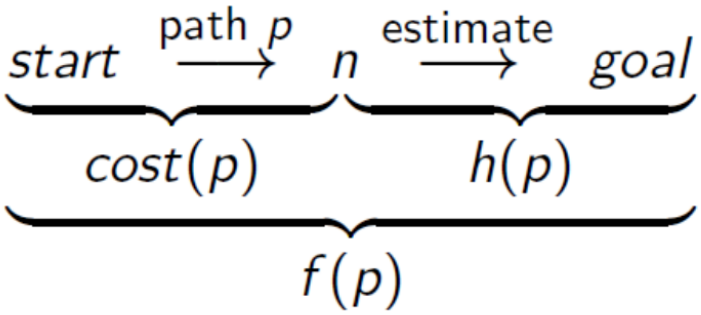
Complexidade do busca pelo melhor-primeiro

- Usa espaço exponencial no comprimento do caminho $O(b^m)$.
- Usa tempo exponencial no comprimento do cominho $O(b^m)$.
- Não é garantido que ela encontra uma solução, mesmo se houver uma, pois um valor de heurística baixo pode fazer com que a busca fique em um ciclo para sempre.
  - No exemplo abaixo os nós laranja tem valor heurístico melhores que os verde, os quais são o caminho certo para a solução
- Nem sempre encontra o caminho mais curto.



Busca A*

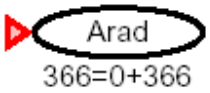
- Usa tanto o custo do caminho até o nó atual quanto o valor da heurística do nó atual.
- ***cost*(*p*)** é o custo do caminho até ***p***.
- ***h*(*p*)** estima o custo do final do caminho de ***p*** até um nó objetivo.
- Logo, ***f*(*p*) = *cost*(*p*) + *h*(*p*)**.
  - ***f*(*p*)** estima o custo total do caminho para ir de um nó de início até um nó objetivo passando por ***p***.



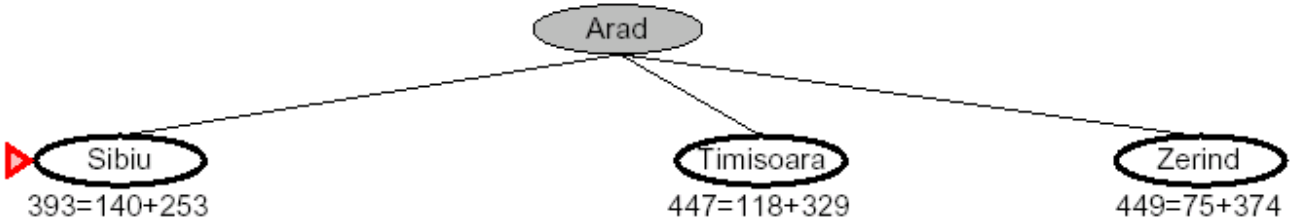
Exemplo: busca A*

- Problema:
  - Ir de **Arad** → **Bucharest**.
- Função heurística:
  - **Distância em linha reta** entre a cidade *n* e Bucharest.
  - Satisfaz a condição de admissibilidade, pois não existe distância menor entre dois pontos do que uma reta.
  - É uma boa heurística, pois induz o algoritmo a atingir o objetivo mais rapidamente.

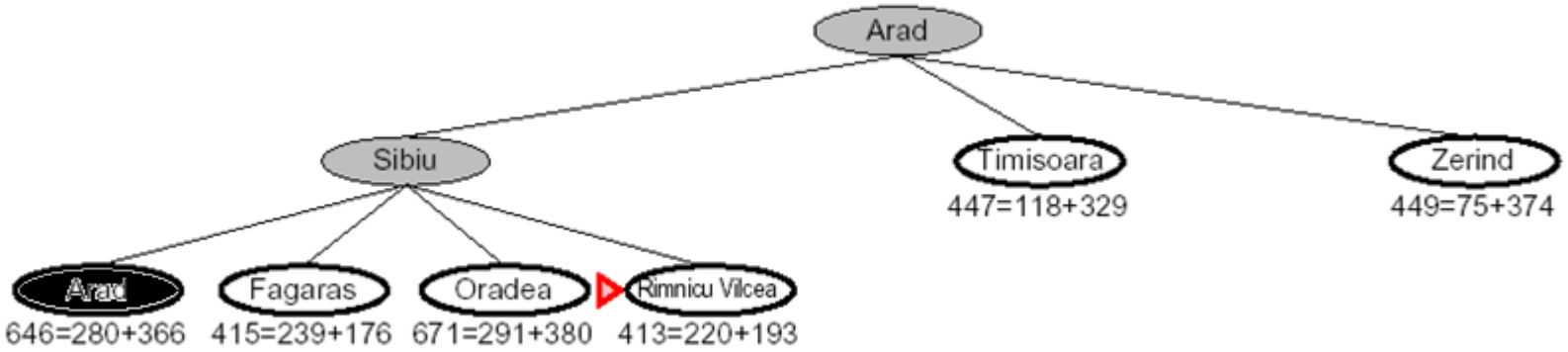
Exemplo: de busca A*



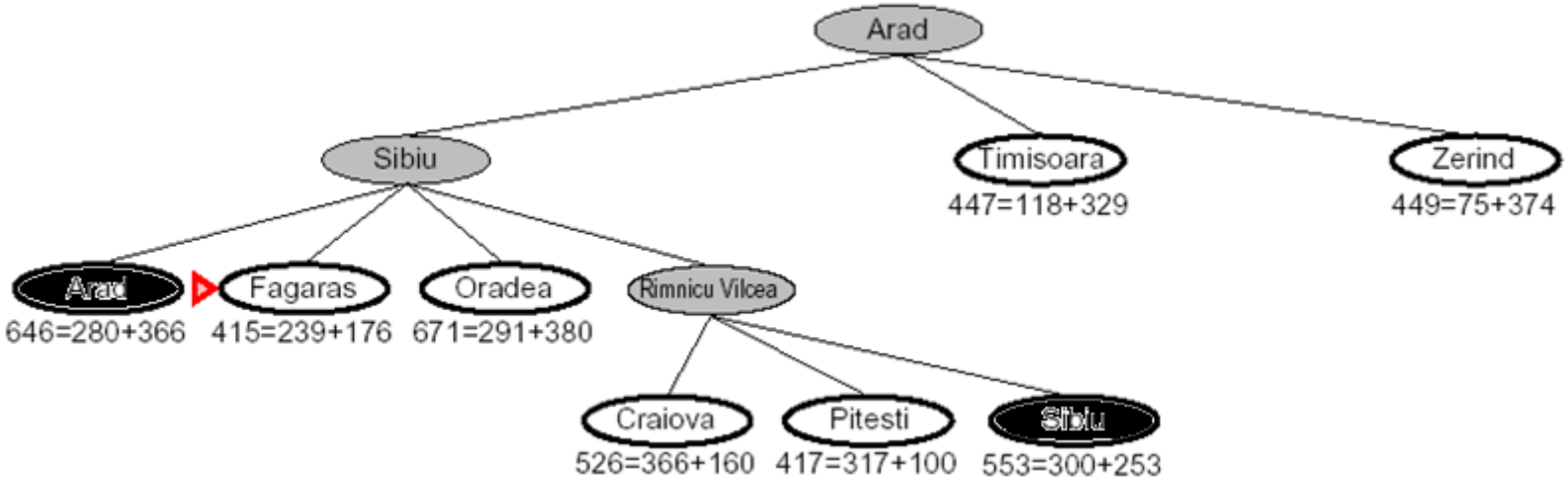
Exemplo: de busca A*



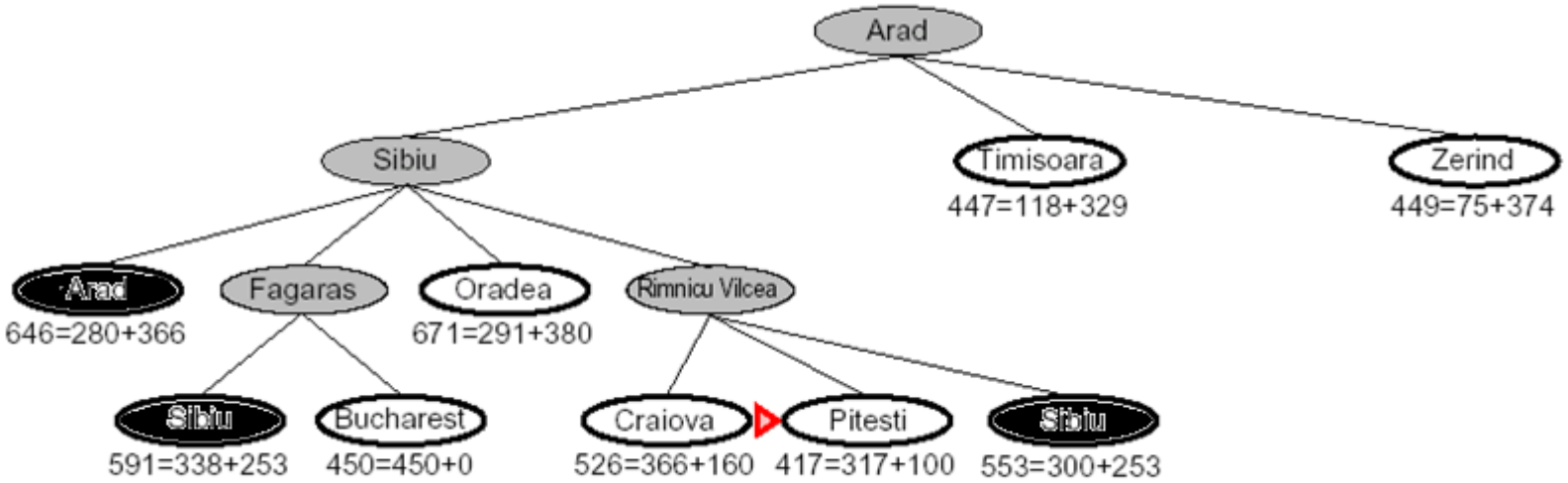
Exemplo: de busca A*



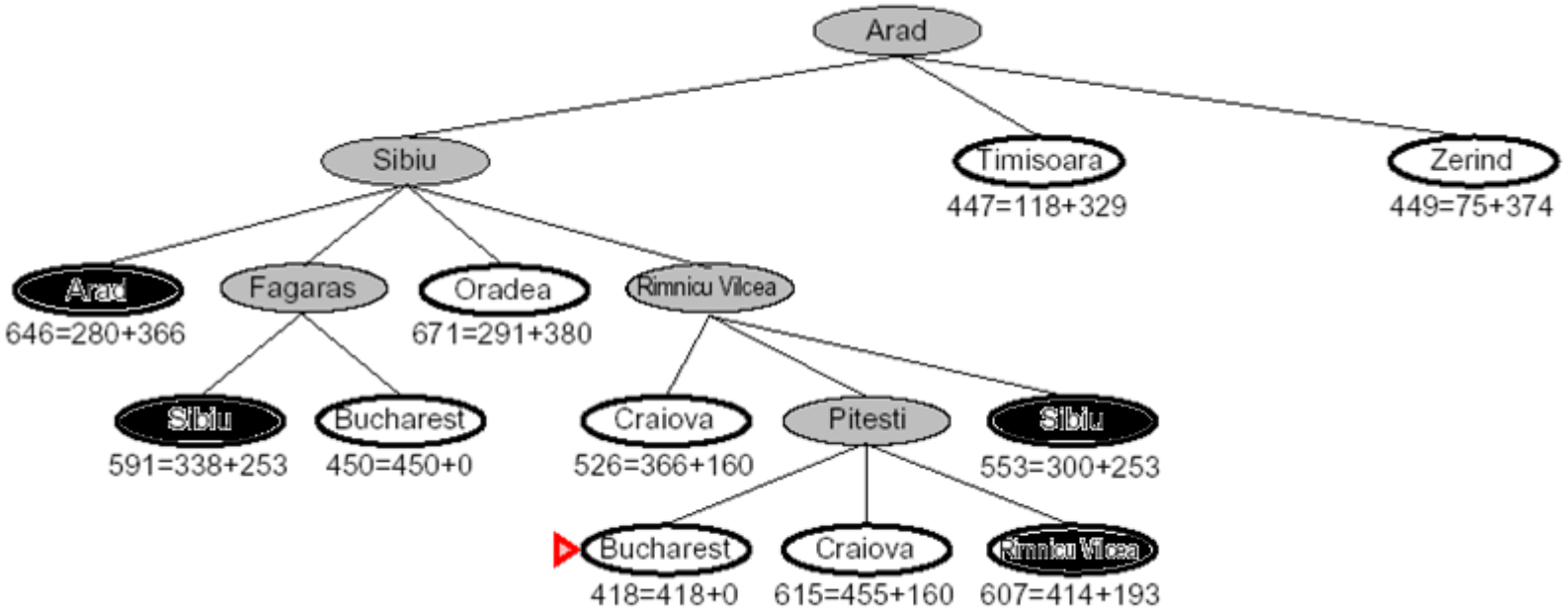
Exemplo: de busca A*



Exemplo: de busca A*



Exemplo: de busca A*



Algoritmo da busca A*

- A* é uma mistura de busca pelo menor-custo-primeiro (baseada em *cost*(*n*)) e busca pelo melhor-primeiro (baseada em *h*(*n*)).

- Ela trata a fronteira como uma ***fila de prioridade*** ordenada por ***f*(*p*)**.

- Ela sempre seleciona o nó ***p*** na fronteira com a menor distância estimada a partir do início até um nó objetivo, restringida a passar pelo nó ***p***.

Algoritmo da busca A*

def busca_A*(G, S, *goal*):

entrada: G # a grafo

S # um conjunto de nós iniciais

objetivo # função booleana que testa se *S* é um nó objetivo

saída: um caminho de um membro de **S** para um nó para o qual a função ***objetivo*** é verdadeira, ou ⊥ se não existir solução.

local: *fronteira* # um conjunto de caminhos

fronteira ← {<s>|s ∈ S}

while *fronteira* :

selecione e **remova** o **1º** nó $\langle s_0,\ldots,s_k\rangle$ da *fronteira*

if *objetivo*(s_k): **return** $\langle s_0,\ldots,s_k\rangle$

adicione(ordem crescente de *f(n)*, fronteira, $\{\langle s_0,\ldots,s_k,s\rangle | \langle s_k,s\rangle \in S\}$)

return ⊥

Análise da busca A*

- Vamos supor que os custos do arco são estritamente positivos.
- **Complexidade de tempo** é $O(b^m)$
  - Se a heurística for completamente não informativa e os custos dos arcos forem os mesmos, a busca A* faz a mesma coisa que a busca em largura.
- **Complexidade de espaço** é $O(b^m)$ como na busca em largura, a busca A* mantém uma fronteira que cresce com o tamanho da árvore
- **Completa**: Sim.
- **Ótima**: Sim.

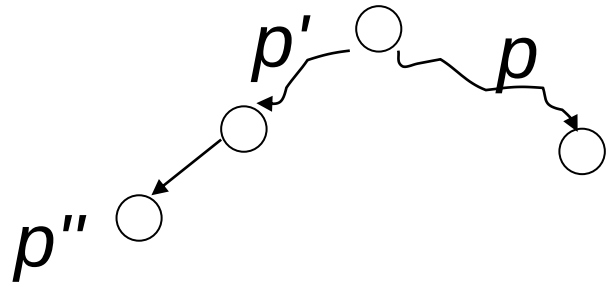
Admissibilidade da busca A*

- Se houver uma solução, a busca A* sempre encontra a solução ótima – i.e. o primeiro caminho para um nó objetivo selecionado – se:
  - O fator de ramificação é finito.
  - O custo dos arcos são delimitados acima de zero (existe algum $\varepsilon > 0$ tal que todos os custos dos arcos são superiores a ε).
  - $h(n)$ é uma subavaliação do comprimento do caminho mais curto da n para um nó de objetivo (i.e. uma heurística admissível).
- No entanto, a complexidade de espaço ainda é um problema.

Porque a busca A* é ótima?

- Se um caminho ***p*** para um nó objetivo é selecionado da fronteira, pode haver um caminho mais curto para um nó objetivo?
- Suponha que caminho ***p´*** está na fronteira. Porque ***p*** foi escolhido antes de ***p´*** e ***h*(*p*)** = 0:

$$\text{cost}(p) \leq \text{cost}(p') + h(p'):$$



- Porque ***h*** é uma subavaliação:

$$\text{cost}(p') + h(p') \leq \text{cost}(p'')$$

para qualquer caminho ***p´´*** até um nó objetivo que estende ***p´***

- Logo cost(***p***) ≤ cost(***p´***) para quaisquer outros caminhos ***p´´*** até um nó objetivo.

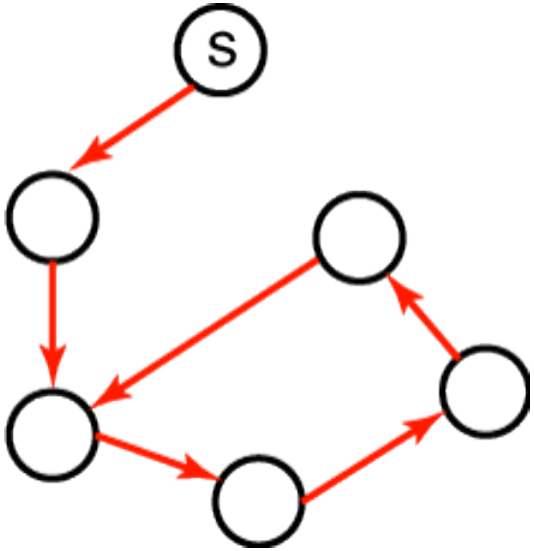
Porque a busca A* é ótima?

- Há sempre um elemento do caminho da solução ótima na fronteira, antes de um nó objetivo ter sido selecionado.
  - Isso ocorre porque, no algoritmo de pesquisa abstrata, existe a parte inicial de cada caminho para um nó objetivo.

- A busca A* para, pois os custos dos caminhos na fronteira continuam a aumentar e, eventualmente, eles excederão qualquer número finito.

- Se os custos forem positivos não há necessidade de verificar a existência de ciclos na busca A*, pois os custos dos caminhos com ciclos serão sempre maiores e, portanto, não serão escolhidos.

Verificação de Ciclos

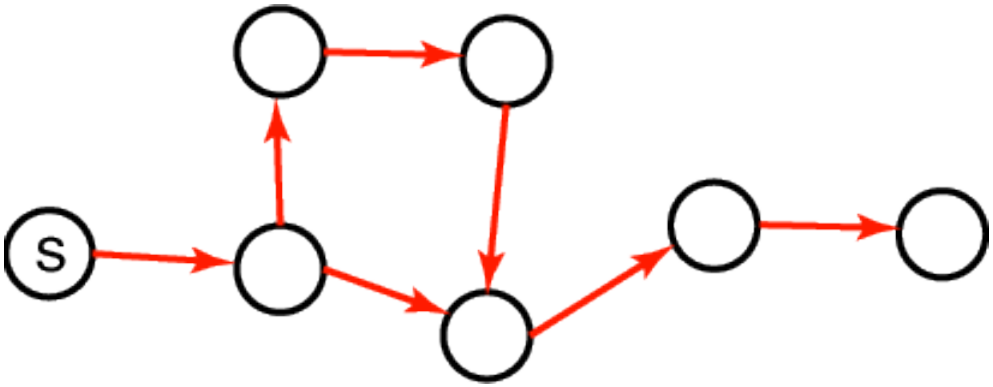


- Um buscador pode podar um caminho que termina em um nó que já está no caminho, sem remover a solução ideal.
  - Usando métodos de busca em profundidade, com o grafo explicitamente armazenado, isso pode ser feito em tempo constante.
  - Para outros métodos, o custo é linear no comprimento do caminho.

Algoritmo genérico de busca em grafo com detecção de ciclo

```
def busca_em_grafo(G, S, goal):
   entrada: G # a grafo
            S # um conjunto de nós iniciais
            objetivo # função booleana que testa se S é um nó objetivo

   saída: um caminho de um membro de S para um nó para o qual a função
          objetivo é verdadeira, ou ⊥ se não existir solução.

   local: fronteira # um conjunto de caminhos
          fechado # conjunto de nós já visitados

   fechado ← {}
   fronteira ← {<s>|s ∈ S}

   while fronteira ≠ {}:
      seleciona e remove <s_0,…,s_k> da fronteira

      if not s_k ∈ fechado:
         if objetivo(s_k): return <s_0,…,s_k>

         fechado ← {s_k}
         fronteira ← fronteira ∪ {<s_o,...,s_k, s>|<s_k,s> ∈ S} # varia com o algoritmo
   return ⊥
```

Remoção de caminhos múltiplos



- A remoção de caminho múltiplos consiste em podar um caminho para o nó ***n*** para o qual o buscador já tenha encontrado um outro caminho.
- A remoção de caminhos múltiplos engloba a verificação de ciclos.
- Isto implica em armazenar na memória todos os nós para os quais já se encontrou caminhos.
- É importante garantir que após a remoção uma solução ótima ainda pode ser encontrada.

Remoção de caminhos múltiplo e soluções ótimas

- Problema: o que fazer se um caminho subsequente para ***n*** é menor do que o primeiro caminho para ***n***?
  - Remova todos os caminhos da fronteira que usam o caminho mais longo.
  - Altere o segmento inicial dos caminhos na fronteira para usar o caminho mais curto.
  - Certifique-se de que isto não aconteça, garantindo que o caminho mais curto para um nó é encontrado antes.

Remoção de caminhos múltiplo e busca A*

- Suponha que o caminho ***p*** para o nó ***n*** foi selecionado, mas ainda há um caminho mais curto para ***n***. Suponha que esse caminho mais curto é via o nó ***p´*** na fronteira.
- Suponha que caminho ***p´*** termina no nó ***n´***.
- cost(*p*) + h(*n*) ≤ cost(*p´*) + h(*n´*) porque ***p*** foi selecionado antes de ***p´***.
- cost(*p´*) + cost(*n´*, *n*) < cost(*p*) porque o caminho para ***n*** via ***p´*** é mais curto.

  cost(n´, n) < cost(p) - cost(p´) ≤ h(n´) - h(n):

- Você pode garantir que isso não ocorrerá caso:

  | h(*n´*) - h(*n*) | ≤ cost(*n´*, *n*).

Restrição monotônica

- A função heurística ***h*** satisfaz a **restrição monotônica** se $|h(m) - h(n)| \leq$ custo (m, n) para cada arco $<m, n>$.
- Se ***h*** satisfaz a restrição monotônica, a busca A* com remoção de caminhos múltiplos sempre encontra o caminho mais curto para um objetivo.
- Este é um reforço ao critério de admissibilidade.

Quando usar?

- Remoção de múltiplos caminhos engloba a checagem de ciclos.
  - Pode ser feito em tempo constante se todos os nós encontrados estiverem armazenados.
- Remoção de múltiplos caminhos.
  - Métodos em largura primeiro – quando temos que armazenar virtualmente todos os nós considerados.
- Checagem de ciclos.
  - Métodos em profundidade primeiro – quando armazenamos somente o caminho para o qual estamos realizando a busca.

Busca em aprofundamento por enumeração e poda (*Depth-first Branch-and-Bound*)

- É uma forma de combinar a busca em profundidade com informações de heurísticas.
- Os nós vizinhos (nós gerados) são ordenados antes de serem colocados na fronteira (que ainda é uma pilha).
  - Seleciona localmente qual subárvore expandir, mas continua fazendo busca em profundidade.
- A complexidade de espaço é linear no tamanho do caminho.
- Não garante encontrar uma solução (como a busca em profundidade), mas se encontrar é a solução ideal.
- Mais útil quando há várias soluções e queremos uma solução ótima.

Busca em aprofundamento por enumeração e poda

- Ideia: manter o custo do caminho de menor-custo encontrado até um objetivo até agora, chamar isto de *limite*.
- Se a busca encontrar um caminho ***p*** tal que *cost*(*p*) + h(p) ≤ *limite* o caminho ***p*** pode ser removido.
- Se for encontrado um caminho não-podado para um objetivo, ele deve ser melhor do que o melhor caminho anterior. Esta nova solução é lembrada e o *limite* é definido como o seu custo.
- A busca pode ser uma busca em profundidade para economizar espaço.
- Como o *limite* deve ser inicializado?

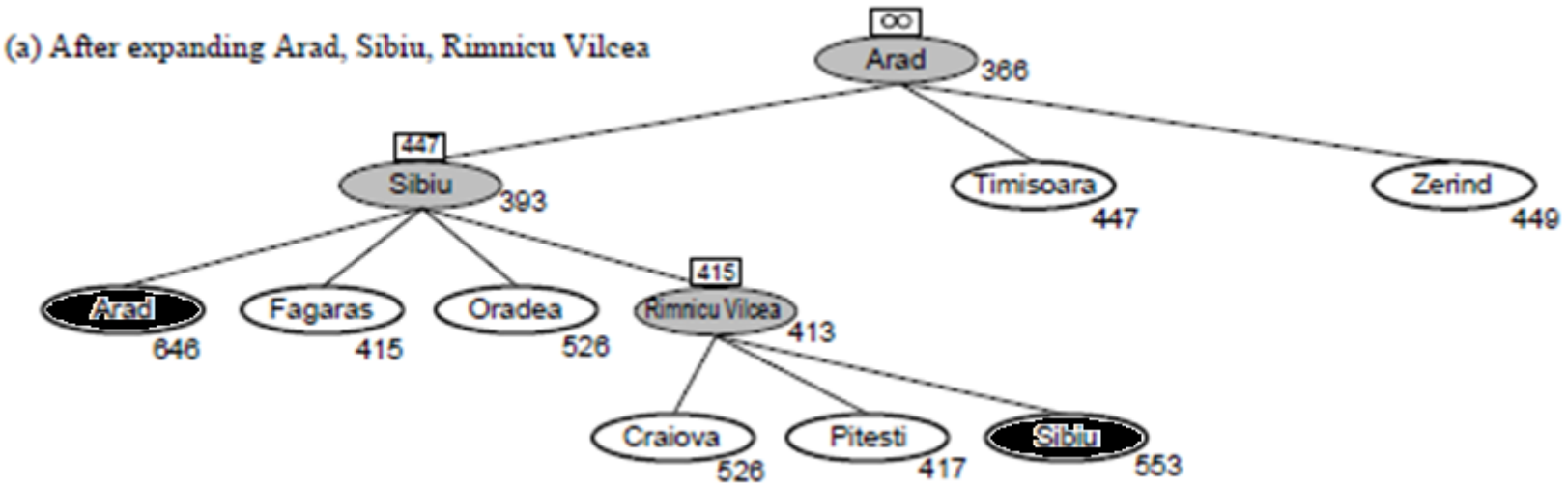
Busca em aprofundamento por enumeração e poda: Inicializando o *limite*

- O *limite* pode ser inicializado em ∞.
- O *limite* pode ser definido como uma estimativa do custo do caminho ótimo. Após a busca em profundidade terminar podemos ter as seguintes situações:
  - Uma solução foi encontrada.
  - Nenhuma solução foi encontrada e nenhum caminho foi podado.
  - Nenhuma solução foi encontrada e um caminho foi removido.

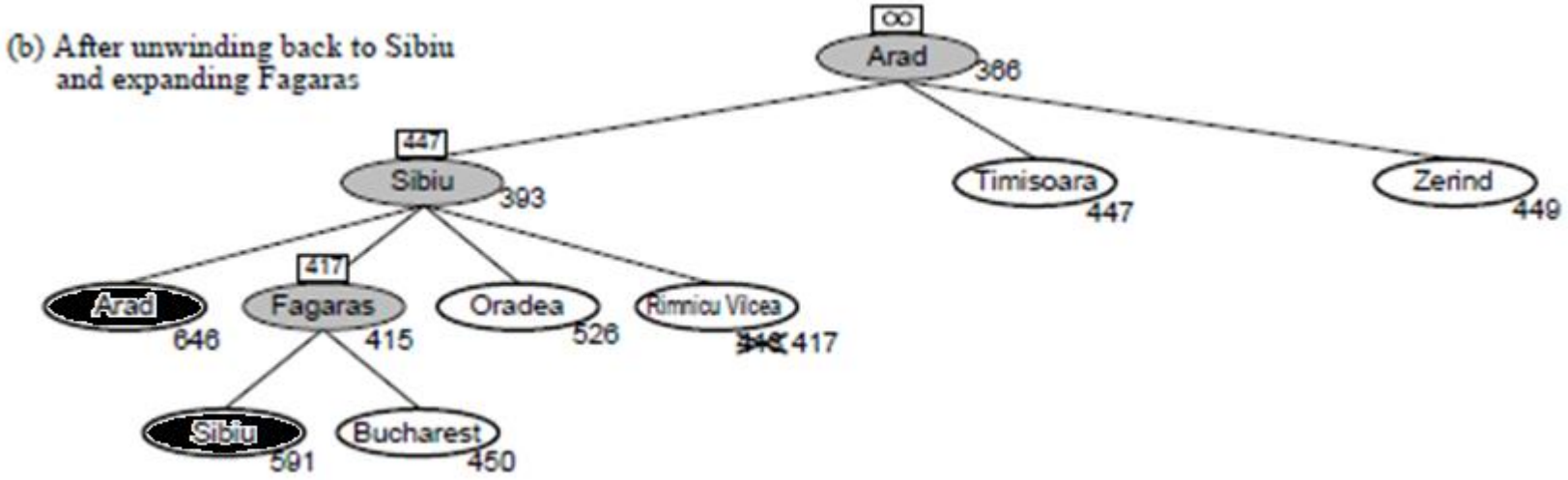
Algoritmo da Busca em aprofundamento por enumeração e poda

```
def busca_por_aprofundamento_enumeração_poda(G, S, goal):
  entrada: G # a grafo
           S # um conjunto de nós iniciais
           objetivo # função booleana que testa se S é um nó objetivo

  saída: um caminho de um membro de S para um nó para o qual a
         função objetivo é verdadeira, ou ⊥ se não existir solução.

  local: fronteira # um conjunto de caminhos


  fronteira ← {<s>|s ∈ S}
  while fronteira:
     selecione e remova o 1º nó <s_0,…,s_k> da fronteira

     if objetivo(s_k): return <s_0,…,s_k>

     adicione(início, fronteira, {ordenar(crescente de f(p), <s_0,…,s_k,s>|<s_k,s> ∈ S})
  return ⊥
```

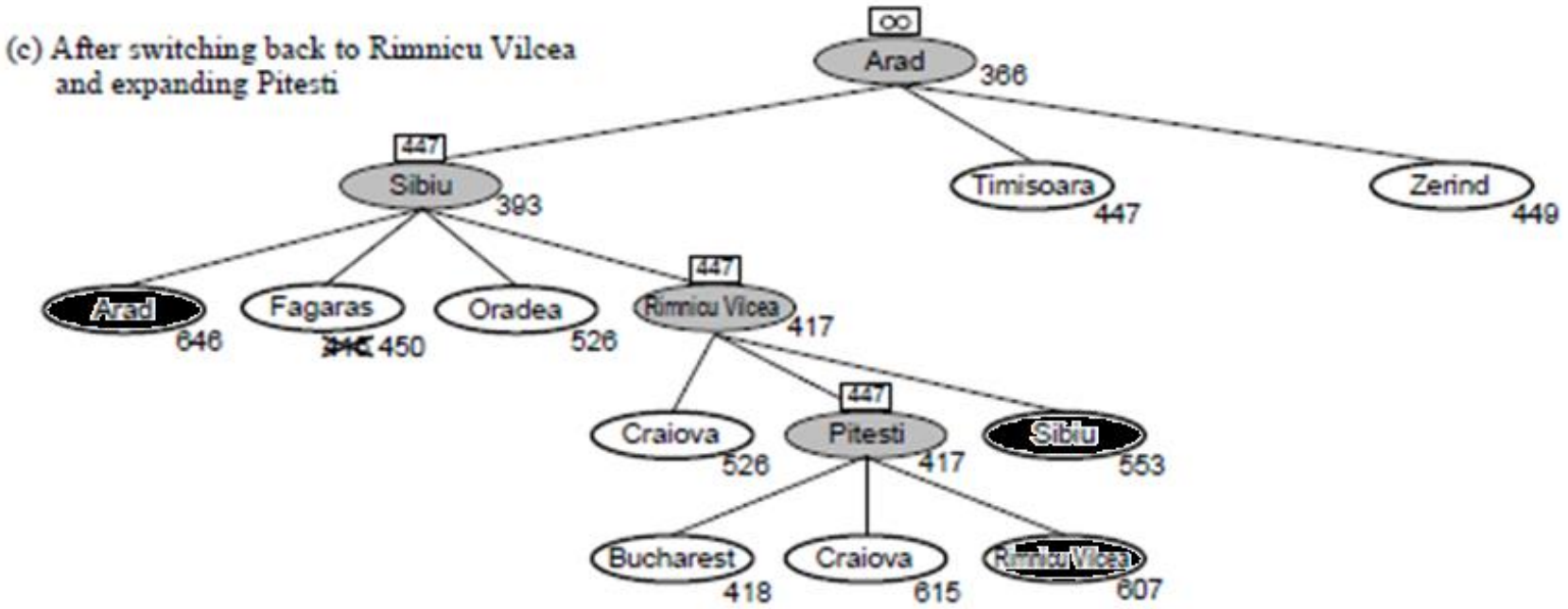
Exemplo: busca em aprofundamento por enumeração e poda



Exemplo: busca em aprofundamento por enumeração e poda



Exemplo: busca por aprofundamento por enumeração e poda



Analise do aprofundamento por enumeração e poda

- Completa: não, pelas mesmas razões que a busca em profundidade não é completa.
  - No entanto, para muitos problemas de interesse não há nenhum caminho infinito e nenhum ciclo, portanto, para muitos problemas *B & B* é completa.

- Complexidade de tempo: **$O(b^m)$**

- Complexidade de espaço: **O(bm)**
  - Enumeração e poda tem a mesma complexidade de espaço da busca em profundidade.

- Ótima: Sim

Sumário: Busca

Tipo de Busca	Seleção da fronteira	Completa	Ótima	Custo de Tempo	Custo de Espaço
Largura	Primeiro nó adicionado	S	S → custos constantes	$O(b^m)$	$O(b^m)$
Profundidade	Último nó adicionado	N S → se espaço de busca for sem ciclos e finito	N	∞ *ou* $O(b^m)$	$O(mb)$
Custo Uniforme	Nó de custo mínimo → *cost*(n)	S → custos > 0	S → custos > 0	$O(b^m)$	$O(b^m)$
Aprofundamento interativo	Último nó adicionado	N S → se espaço de busca for sem ciclos e finito	S → custos constantes	$O(b^m)$	$O(mb)$
Melhor-primeiro	Nó de custo mínimo local → *h*(n)	N	N	$O(b^m)$	$O(b^m)$
A*	Nó de custo mínimo estimado → *f*(n)	S	S	$O(b^m)$	$O(b^m)$
Aprofundamento enumeração-poda	Nó de custo mínimo estimado → *f*(n)	N S → se espaço de busca for sem ciclos e finito	S	∞ *ou* $O(b^m)$	$O(mb)$