

# Inteligência Artificial

## Ciência da Computação

**Prof. Aline Paes / [alinepaes@ic.uff.br](mailto:alinepaes@ic.uff.br)**

**Formulação Genérica de Algoritmos de busca - RN 3.3**



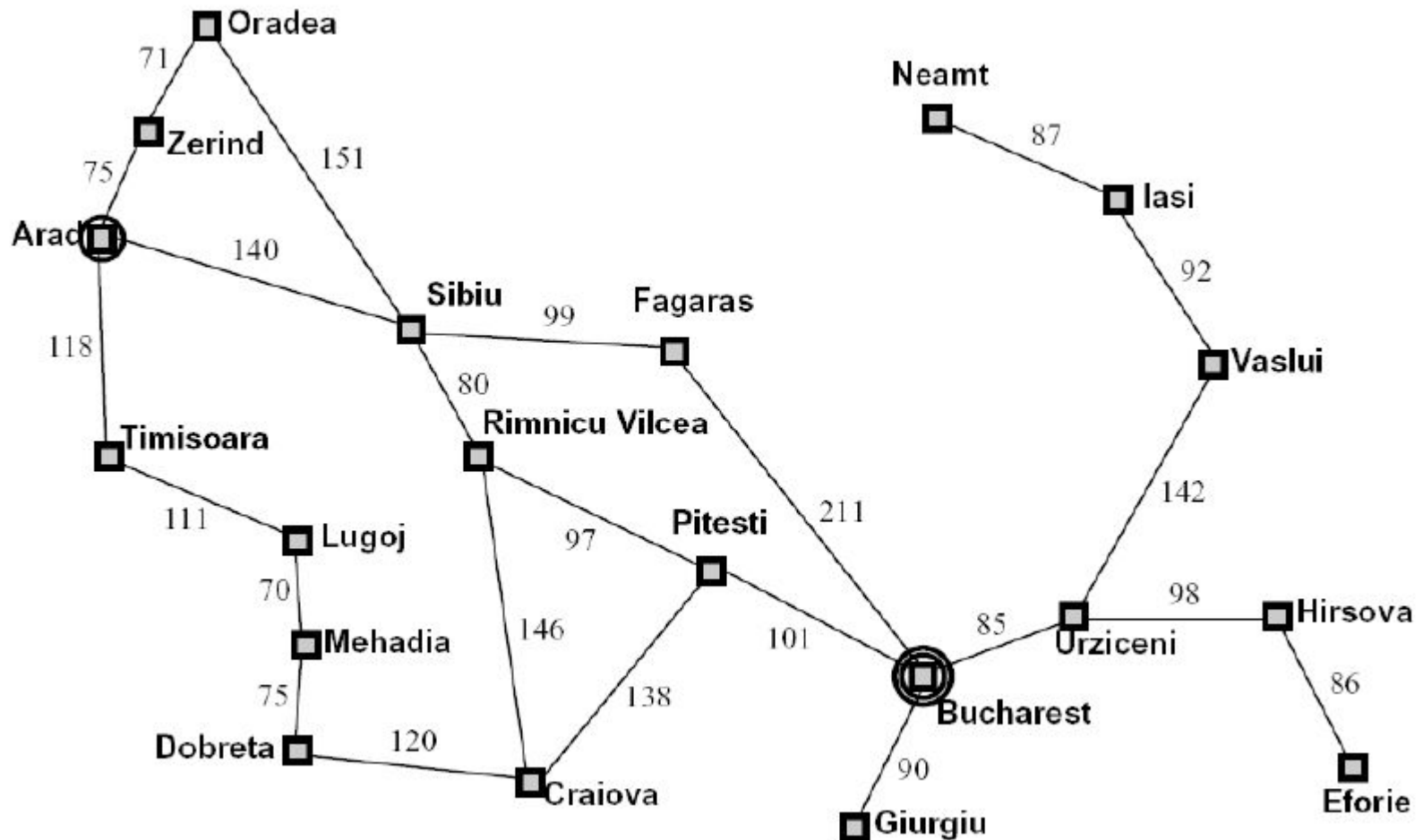
**Universidade Federal Fluminense**



# Resolução de problemas

- Um *problema* é um objetivo e um conjunto de meios para atingir o objetivo.
  - Objetivo
  - Ações
  - Estados
  - Transições

# Exemplo de problema

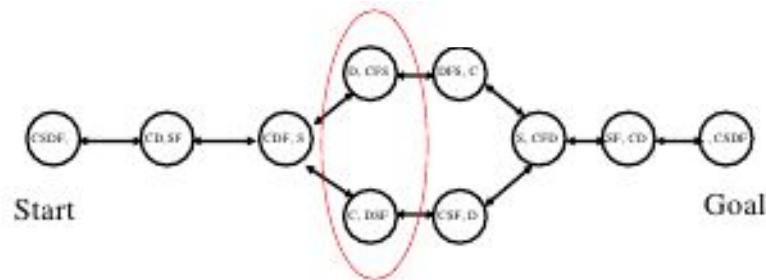


# Formulação

- Estado inicial:  $\text{in}(\text{Arad})$
- Ações:  $\text{go}(X)$ 
  - $\text{actions}(\text{in}(\text{Arad})) = \{\text{go}(\text{Sibiu}), \text{go}(\text{Timisoara}), \text{go}(\text{Zerind})\}$
- Modelo de transição:
  - $\text{result}(\text{in}(\text{Arad}), \text{go}(\text{Sibiu})) = \text{in}(\text{Sibiu})$
- Teste de meta
  - $\text{in}(\text{Bucharest})?$
- Custo do caminho:
  - soma das arestas no caminho
  - $\text{Arad} \dashrightarrow \text{Oradea} = 75 + 71 = 146$

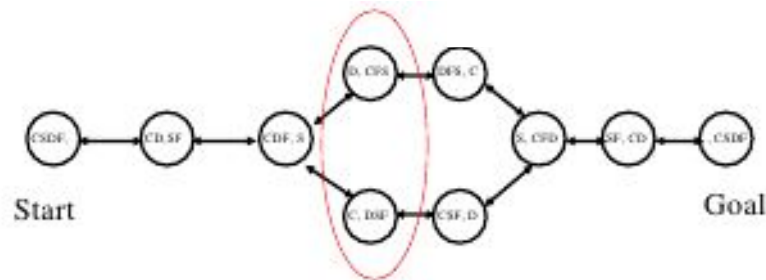
# Espaço de estados

- Modelo de transição gera um **espaço de estados**
  - **Grafo** direcionado



# Espaço de estados

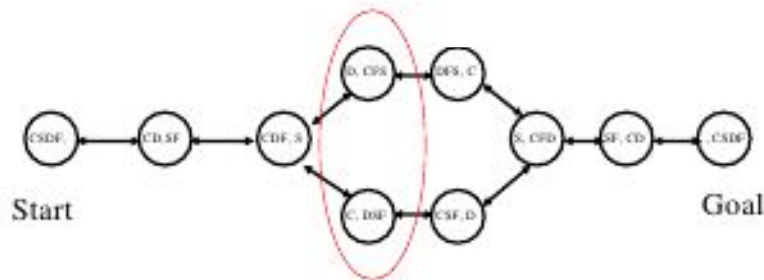
- Modelo de transição gera um **espaço de estados**
  - **Grafo** direcionado



- Existem vários estados gerados, não expandidos
  - Qual expandir primeiro?

# Espaço de estados

- Modelo de transição gera um **espaço de estados**
  - Grafo direcionado

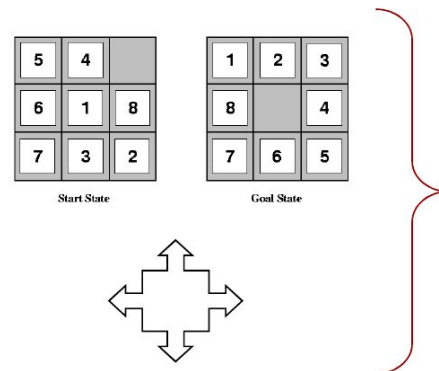


- 
- Existem vários estados gerados, não expandidos
  - Qual expandir primeiro?

Estratégia de busca

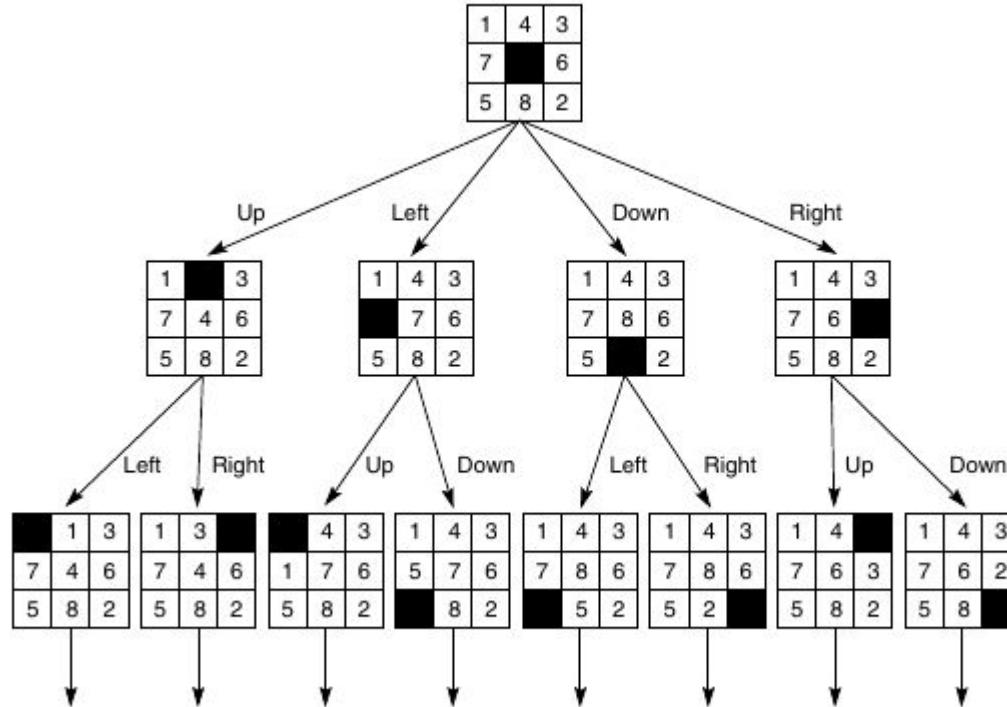
# Resolução de problemas usando busca

- *Formulação do problema*: Estado inicial, Objetivo, ações, modelo de transição, custo
- *Algoritmo de busca*: Parte da formulação do problema e busca sistematicamente por uma solução.





# Exemplo - Quebra cabeça de 8

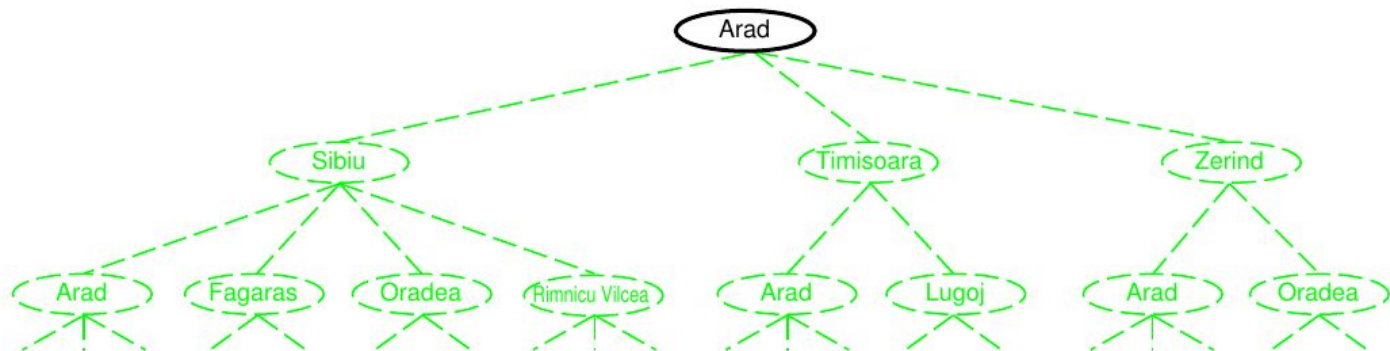


# Buscando por soluções

- **Busca:** caminho da **solução** no **espaço de estados**
  - representada por uma **árvore**
  - **Solução:** sequência de ações do estado inicial ao objetivo
- Entrada de um algoritmo de busca: um **problema** formalizado
- Saída do algoritmo: uma **solução**

# Buscando por soluções

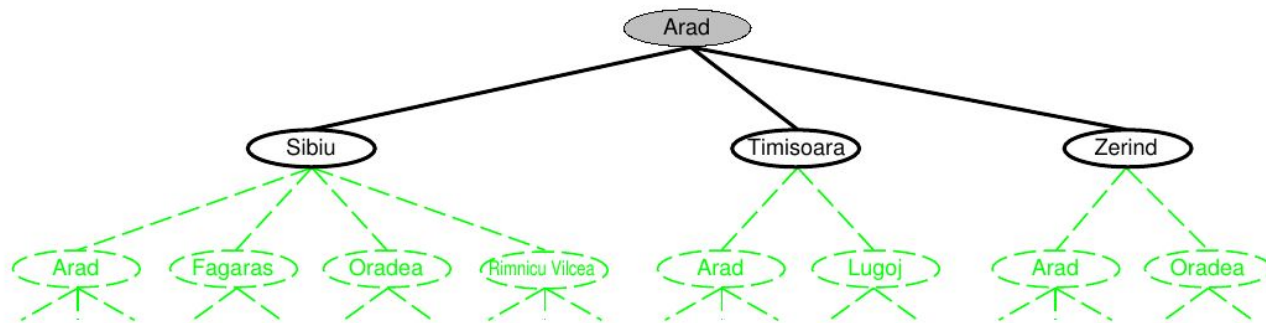
- **Árvore de busca**
  - Nós correspondem a estados
  - Estado inicial na raiz



# Buscando por soluções

- **Árvore de busca**

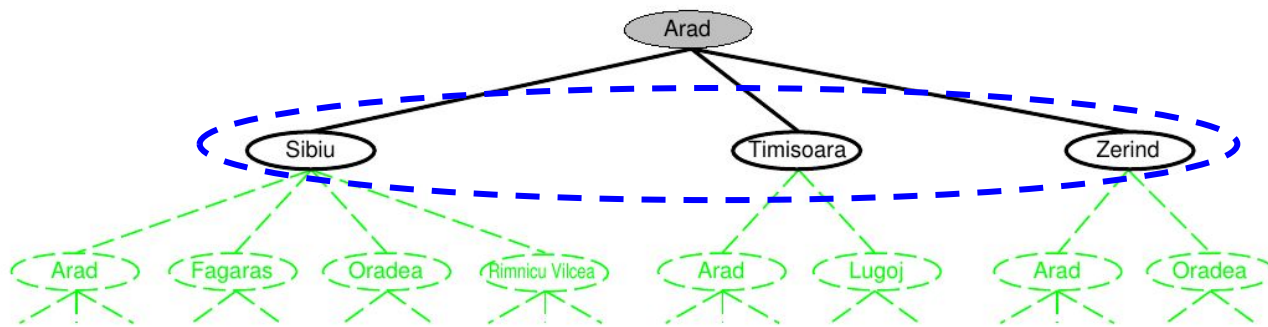
- **Expansão** do estado corrente **s** (uma folha), a partir da função **actions(s)**
- Novos estados são **gerados**, a partir da função **result(s,a)**



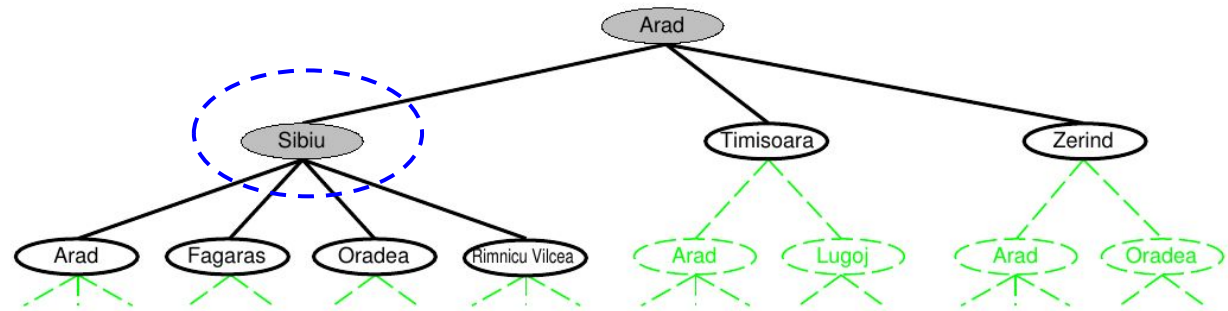
# Buscando por soluções

- **Árvore de busca**

- Estado inicial na raiz
- **Expansão** do estado corrente **s**, a partir da função **actions(s)**
- Novos estados são **gerados**, a partir da função **result(s,a)**
- **Antes** de expandir cada nó, **teste** se ele é a meta
- Nós folha disponíveis para expansão constituem a **fronteira**



# Buscando por soluções



- **Árvore de busca**

- Estado inicial na raiz
- **Expansão** do estado corrente **s**, a partir da função **actions(s)**
- Novos estados são **gerados**, a partir da função **result(s,a)**
- **Antes** de expandir cada nó, **teste** se ele é a meta
- Nós folha disponíveis para expansão constituem a **fronteira**
- Que **estado** expandir a partir da fronteira? **Estratégia de busca**
- **Busca**: Escolher um caminho a seguir, deixando o resto para depois

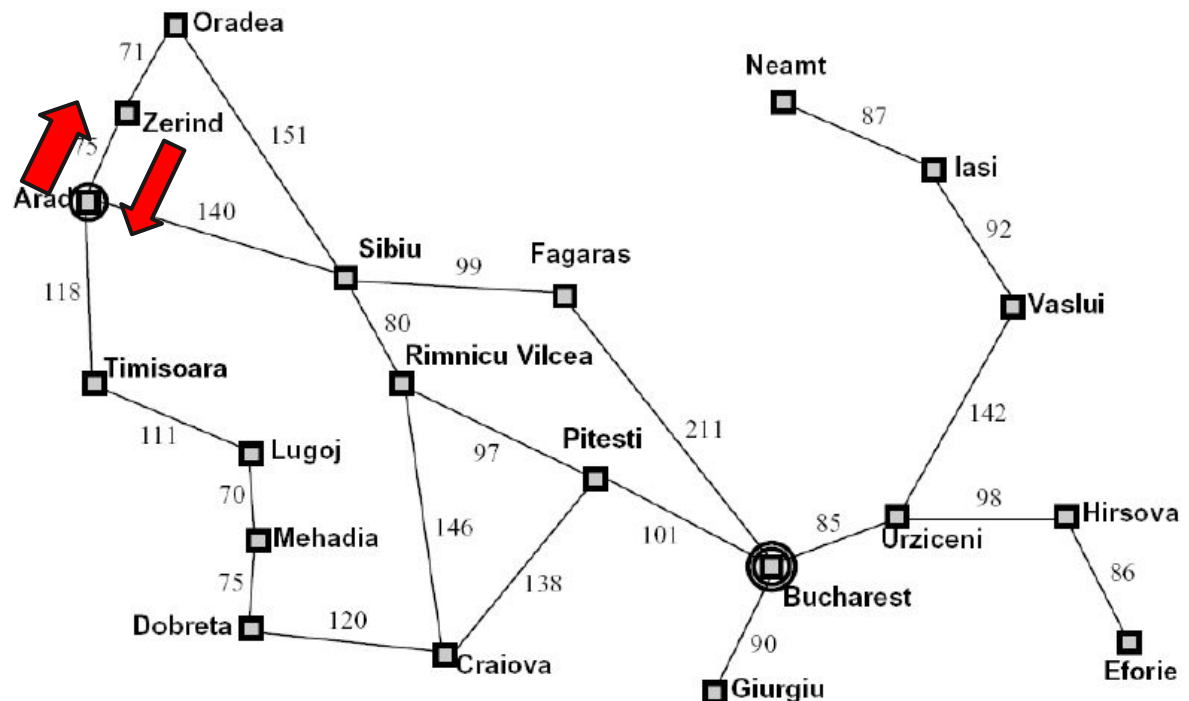
# Algoritmo de busca genérico

```
function TREE-SEARCH(problem) returns a solution, or failure
  inicializar frontier usando o estado inicial em problem
  loop do
    if frontier está vazia return failure
    escolher um node folha e removê-lo de frontier
    if node contém um estado meta return a solução correspondente
    expandir node escolhido, adicionando nodes resultantes à frontier
```

# Busca por soluções

- Evitar

- caminho em **loop**: estados repetidos no mesmo ramo

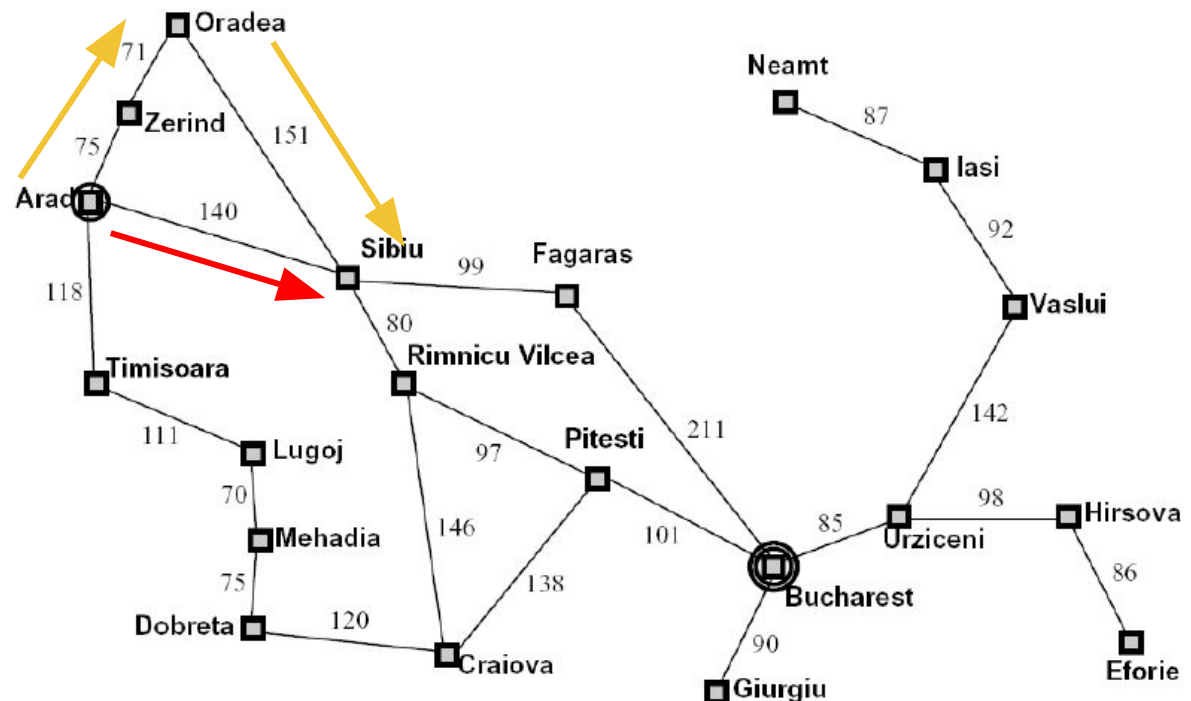




# Busca por soluções

- Evitar

- caminhos **redundantes** mais custosos

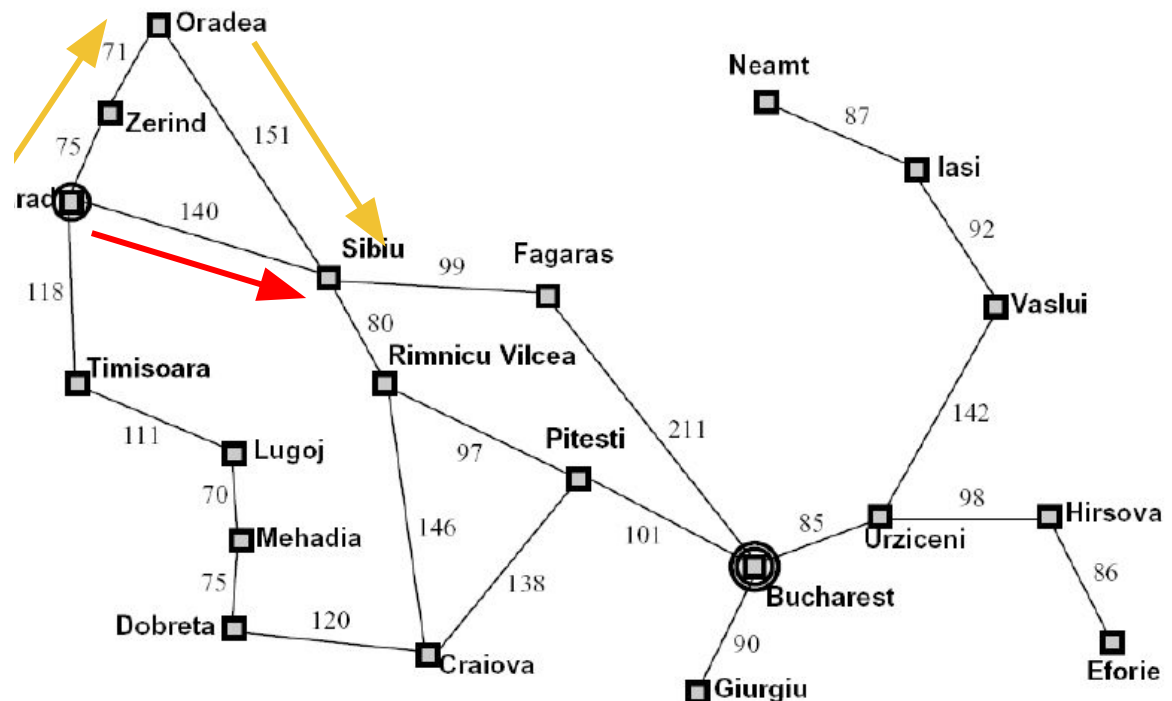


# Busca por soluções

F = [Arad]  
S = {}  
F = [Sibiu, Timi, Zerind]  
S = {Arad}  
F = [Timi, Zerind]  
S = {Arad, Sibiu}  
F = [Timi, Zerind, Fag, Orad, RV]  
S = {Arad, Sibiu, Timi}  
F = [Zerind, Fag, Orad, RV, Lugoj]  
S = {Arad, Sibiu, Timi, Zerind}  
F = [Fag, Orad, RV, Lugoj]  
S = {Arad, Sibiu, Timi, Zerind, Fag}  
F = [Orad, RV, Lugoj]  
Bucharest!

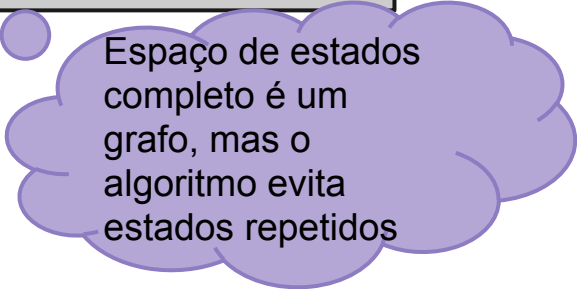
Arad-Sibiu-Fag-Bucharest

redundantes mais custosos



# Algoritmo de busca genérico

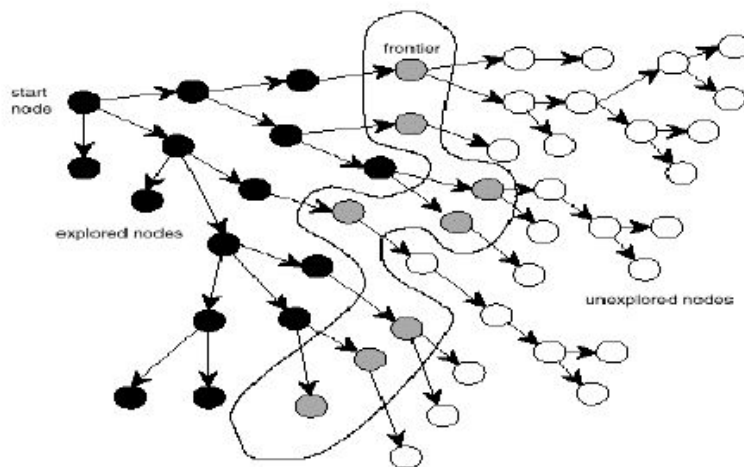
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  inicializar frontier usando o estado inicial em problem
  inicializar o exploredSet como vazio
  loop do
    if frontier está vazia return failure
    escolher um node folha e removê-lo de frontier
    if node contém um estado meta return a solução correspondente
    adicionar node em exploredSet
    expandir node escolhido, adicionando os nodes resultantes à frontier
      somente se ele não está em frontier ou em exploredSet
```



Espaço de estados completo é um grafo, mas o algoritmo evita estados repetidos

# Algoritmo de busca genérico

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  inicializar frontier usando o estado inicial em problem
  inicializar o exploredSet como vazio
  loop do
    if frontier está vazia return failure
    escolher um node folha e removê-lo de frontier
    if node contém um estado meta return a solução correspondente
    adicionar node em exploredSet
    expandir node escolhido, adicionando os nodes resultantes à frontier
      somente se ele não está em frontier ou em exploredSet
```



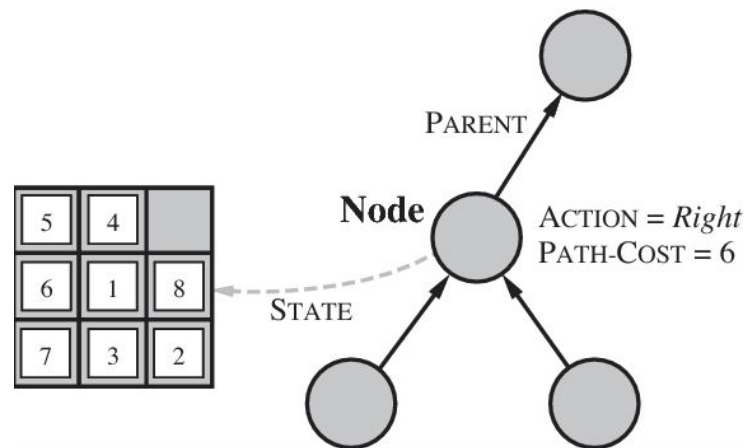
*frontier* separa  
estados explorados  
e não explorados

# Estrutura de dados para algoritmos de busca

- Cada nó  $n$  da árvore de busca é uma estrutura contendo
  - `n.state`
  - `n.parent`
  - `n.action`
  - `n.pathCost`

# Estrutura de dados para algoritmos de busca

- Cada nó  $n$  da árvore de busca é uma estrutura contendo
  - $n.state = [ [5,4,b], [6,1,8], [7,3,2] ]$
  - $n.parent$  = referencia uma outra estrutura de nó
    - $n.state$  do pai =  $[ [5,b,4], [6,1,8], [7,3,2] ]$
  - $n.action = right$
  - $n.pathCost = 6$



# Função usada na expansão

```
function CHILD-NODE(problem, parent, action) returns a node  
  return um node com  
    node.state = problem.result(parent.state, action)  
    node.parent = parent  
    node.action = action  
    node.pathCost = parent.pathCost + problem.step-cost(parent.state, action)
```

# Estratégias de busca

- variam na forma em que o nó é inserido em *frontier*
- implementada usando listas (filas/lista FIFO ou pilhas/lista LIFO ou listas de prioridade)

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  inicializar frontier usando o estado inicial em problem
  inicializar o exploredSet como vazio
  loop do
    if frontier está vazia return failure
    escolher um node folha e removê-lo de frontier
    if node contém um estado meta return a solução correspondente
    adicionar node em exploredSet
    expandir node escolhido, adicionando os nodes resultantes à frontier
      somente se ele não está em frontier ou em exploredSet
```



# Critérios de avaliação

- Completude

- *é garantido que o algoritmo retorna uma solução, quando alguma existir?*

- Otimalidade

- *a estratégia encontra a solução ótima?*

- Complexidade de tempo

- Complexidade de espaço

- ambos medidos em termos de
    - **b**: número máximo de sucessores de qualquer nó
    - **d**: número do nó meta mais profundo, desde a raiz
    - **m**: tamanho máximo de qualquer caminho

# Buscas não informadas

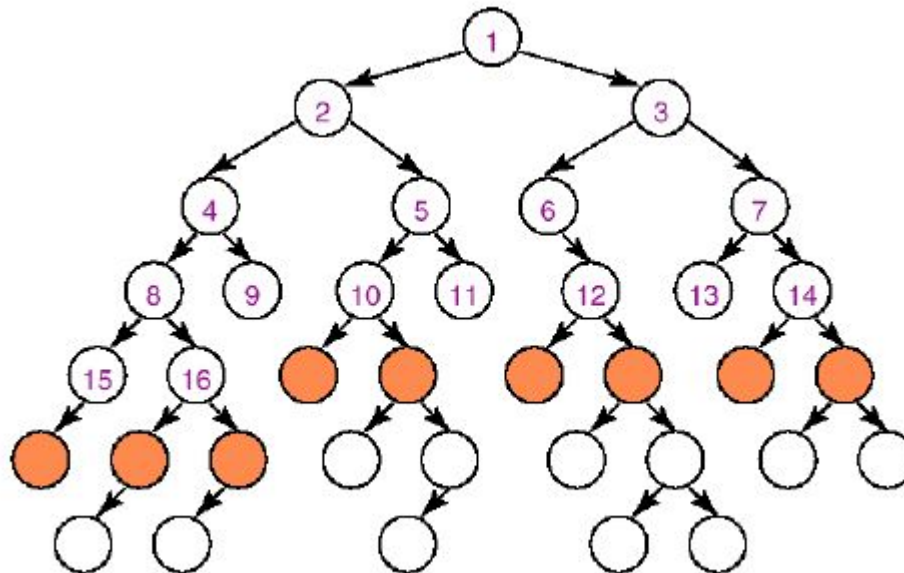
- Não consideram nenhuma informação adicional ao problema
  - Largura
  - Profundidade
  - Profundidade Limitada
  - Profundidade Iterativa
  - Bidirecional
  - Custo Uniforme

# Busca em largura

- Frontier é uma **fila (FIFO)**
  - **Remoção**
    - escolha do nó a ser expandido
      - **primeiro** nó da fila
  - **Inserção**
    - resultado da expansão do nó escolhido
      - nós são inseridos no **fim** da fila
  - nós mais **velhos** são expandidos **primeiro**
  - **Teste de meta** pode ser aplicado na **geração** ao invés de ser na expansão

# Busca em largura

- Todos os nós em um nível devem ser expandidos antes dos nós do próximo nível



# Busca em largura

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** uma solução ou falha

*node* = um nó com *state* = *problem.initialState*

*pathCost* = 0

**if** *problem.goalTest*(*node.state*) **return** *solution*(*node*)

*frontier* = lista FIFO com *node*

*explored* = emptySet

**loop do**

**if** *frontier.empty*() **return** falha

*node* = *frontier.pop*()

*explored.add*(*node.state*)

**for each** *action* in *problem.actions*(*node.state*) **do**

*child* = *child-node*(*problem*, *node*, *action*)

**if** *child.state* não está em *explored* ou em *frontier*

**if** *problem.goalTest*(*child.state*) **return** *solution*(*child*)

*frontier.add*(*child*)

retorna o caminho da raiz até node

método para inserção de elementos: acrescenta no fim da fila

remove sempre no início da lista

teste na geração do nó

# Avaliação

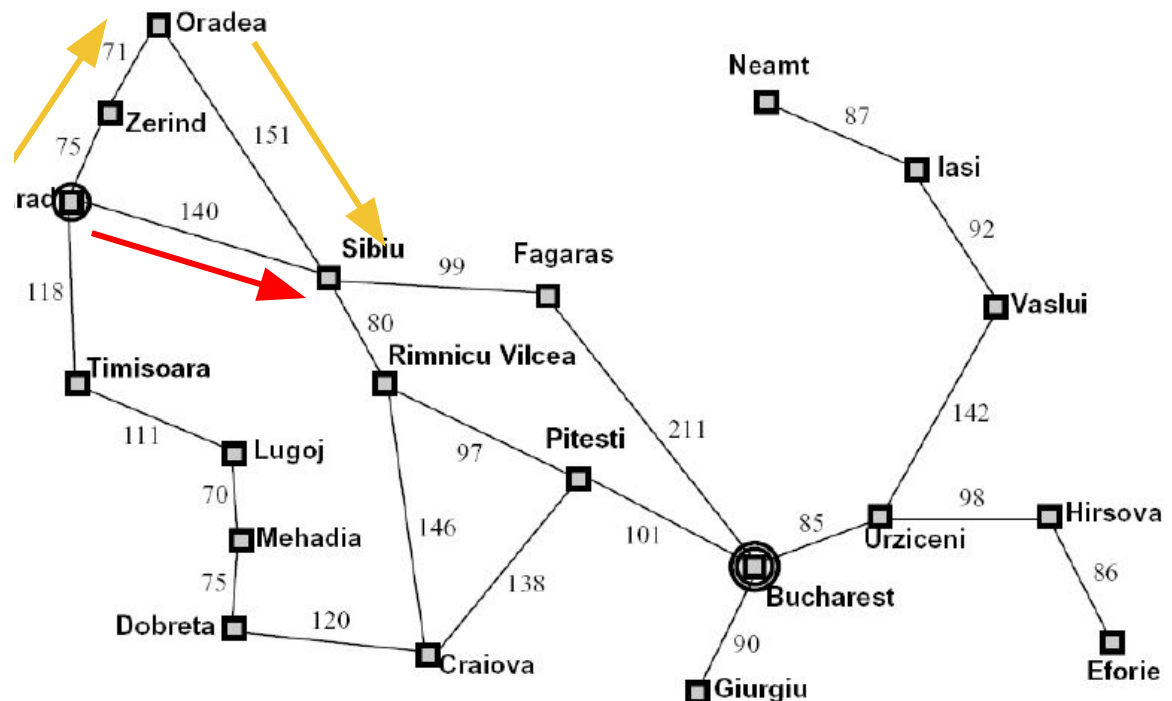
- Completa?
  - Sim (se  $b$  é finito)
- Tempo?
  - $b + b^2 + b^3 + \dots + b^d + b(b^d) = O(b^d)$
- Espaço?
  - $O(b^d)$ 
    - mantém todos os nós na memória
- Ótima?
  - Sim, se todas as ações tiverem o mesmo custo

# Busca em largura

F = [Arad]  
S = {}  
F = [Sibiu, Timi, Zerind]  
S = {Arad}  
F = [Timi, Zerind]  
S = {Arad, Sibiu}  
F = [Timi, Zerind, Fag, Orad, RV]  
S = {Arad, Sibiu, Timi}  
F = [Zerind, Fag, Orad, RV, Lugoj]  
S = {Arad, Sibiu, Timi, Zerind}  
F = [Fag, Orad, RV, Lugoj]  
S = {Arad, Sibiu, Timi, Zerind, Fag}  
F = [Orad, RV, Lugoj]  
Bucharest!

Arad-Sibiu-Fag-Bucharest

redundantes mais custosos



# Busca em Profundidade

**function** DEPTH-FIRST-SEARCH(*problem*) **returns** uma solução ou falha

*node* = um nó com *state* = *problem.initialState*

*pathCost* = 0

**if** *problem.goalTest*(*node.state*) **return** *solution*(*node*)

*frontier* = lista LIFO com *node*

método para inserção de elementos: acrescenta no topo da pilha

*explored* = emptySet

**loop do**

**if** *frontier.empty*() **return** falha

*node* = *frontier.pop*()

*explored.add*(*node.state*)

**for each** *action* in *problem.actions*(*node.state*) **do**

*child* = *child-node*(*problem*, *node*, *action*)

**if** *child.state* não está em *explored* ou em *frontier*

**if** *problem.goalTest*(*child.state*) **return** *solution*(*child*)

*frontier.add*(*child*)

teste na geração do nó

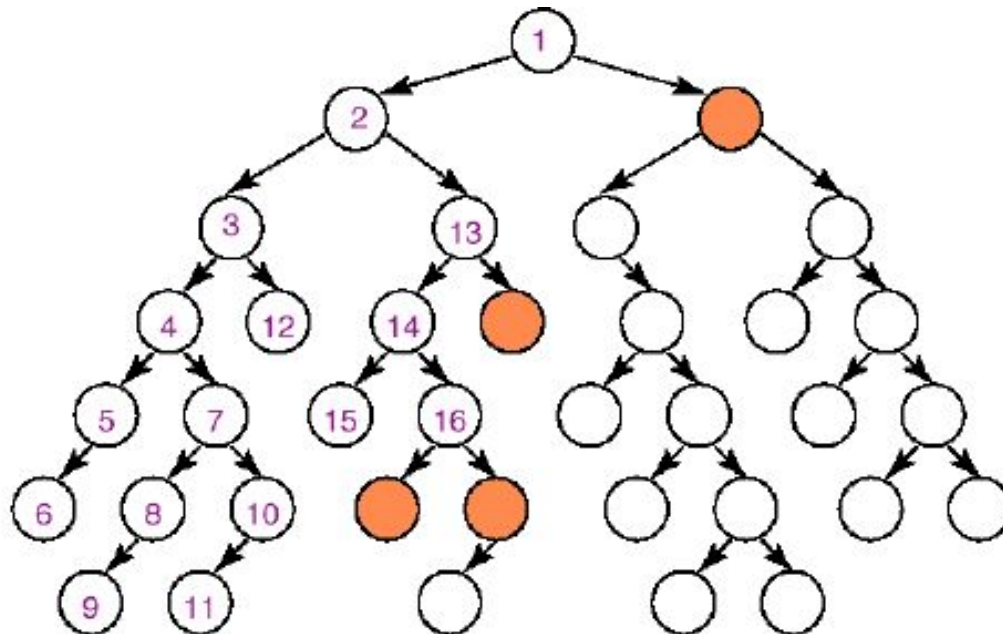


# Busca em profundidade

- Frontier é uma **pilha (LIFO)**
  - **Remoção**
    - escolha do nó a ser expandido
      - nó no **topo** da pilha
  - **Inserção**
    - resultado da expansão do nó escolhido
      - nós são inseridos no **topo** da pilha
  - nós mais **novos** são expandidos **primeiro**

# Busca em profundidade

- Expansão dos nós mais **profundos** primeiro



# Avaliação

- Completa?

- Sim, se evitar estados repetidos e espaço de estados for finito
  - caso contrário, pode ficar preso em um caminho infinito

- Tempo?

- $O(b^m)$
- Ruim se  $m \gg d$

- Espaço?

- $O(bm)$

- Ótima?

- Não

# Busca em profundidade limitada

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns uma solução ou falha/cutoff  
  return RECURSIVE-DLS(makeNode(problem.initialState), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a solution or failure/cutoff  
  if problem.goalTest(node.state) return solution(node)  
  else if limit == 0 return cutoff  
  else  
    cutoffOccurred = false  
    for each action in problem.actions(node.state) do  
      child = child-node(problem, node, action)  
      result = RECURSIVE-DLS(child, problem, limit - 1)  
      if result == cutoff  
        cutoffOccurred = true  
      else if result != failure return result  
  if cutoffOccurred return cutoff else return failure
```

# Avaliação

- Completa?
  - Não, se  $\text{limite} < d$
- Tempo?
  - $O(b^{\text{limit}})$
- Espaço?
  - $O(b \text{limit})$
- Ótima?
  - Não, se  $\text{limite} > d$

# Busca em profundidade iterativa

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns uma solução ou falha
  for depth = 0 to infinite do
    result = DEPTH-LIMITED-SEARCH(problem, depth)
    if result != cutoffOcurred return result
```

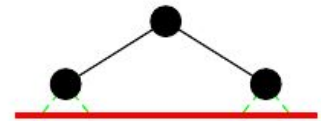
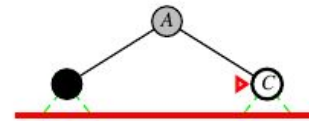
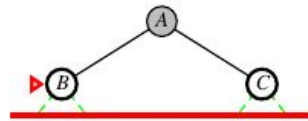
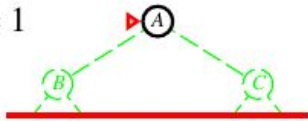
# Busca em profundidade iterativa

Limit = 0



# Busca em profundidade iterativa

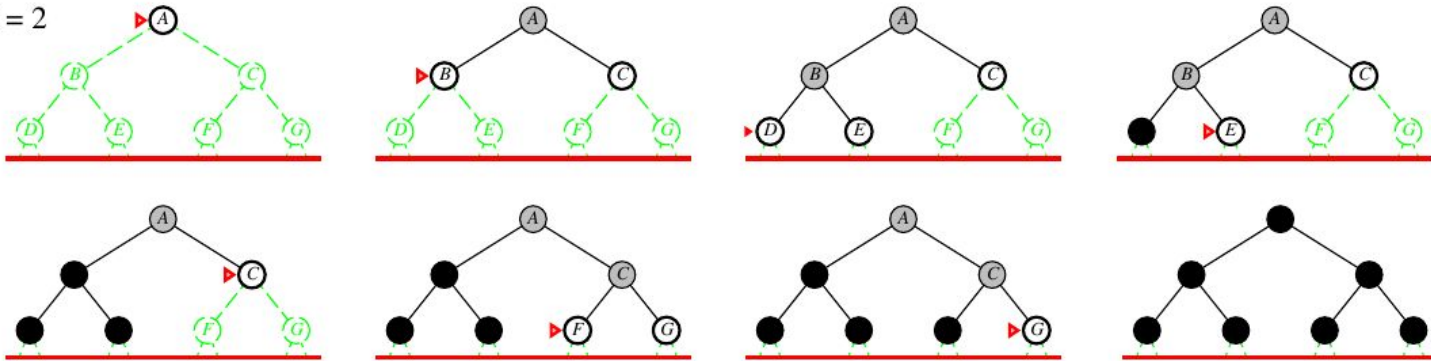
Limit = 1





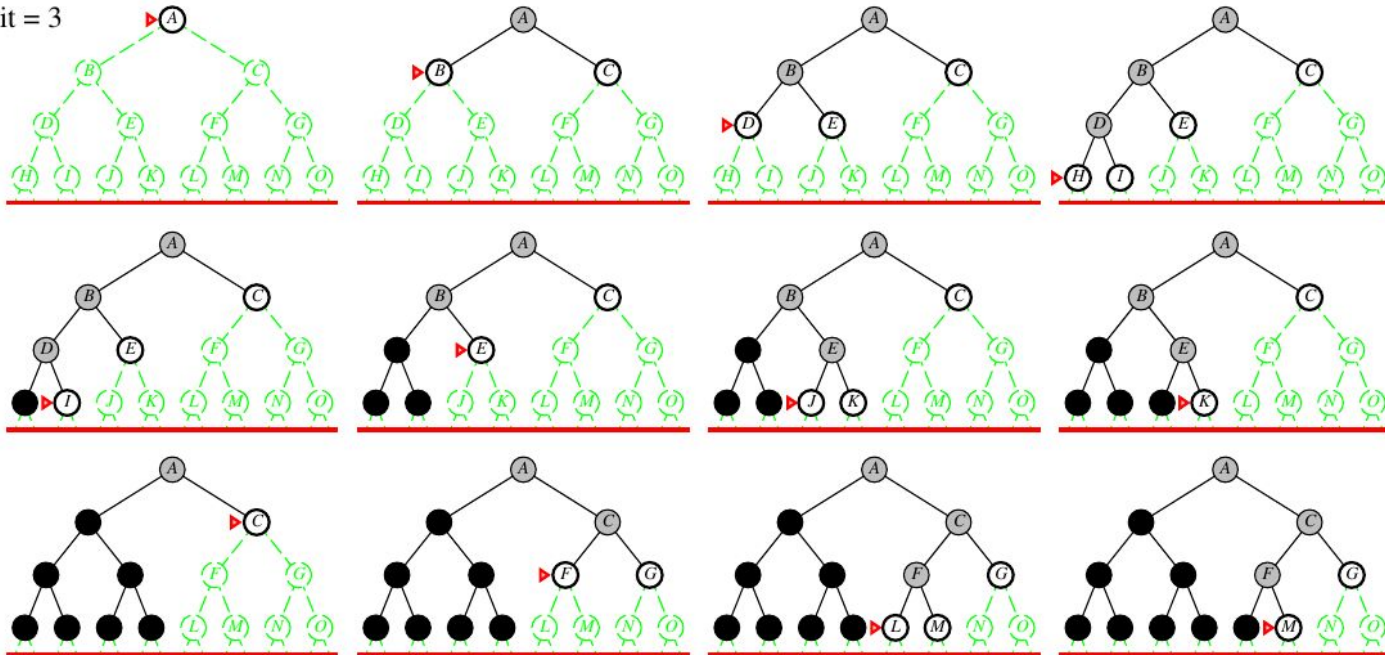
# Busca em profundidade iterativa

Limit = 2



# Busca em profundidade iterativa

Limit = 3

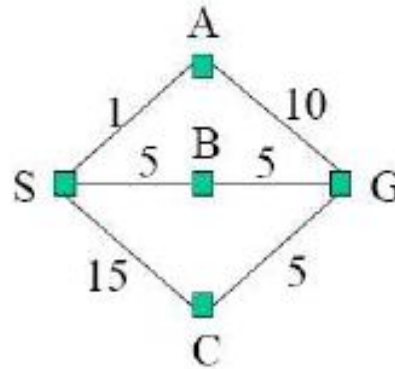


# Avaliação

- Completa?
  - Sim
- Tempo?
  - $O(b^d)$
- Espaço?
  - $O(bd)$
- Ótima?
  - Sim, se custo é uma função não decrescente da profundidade  $d$

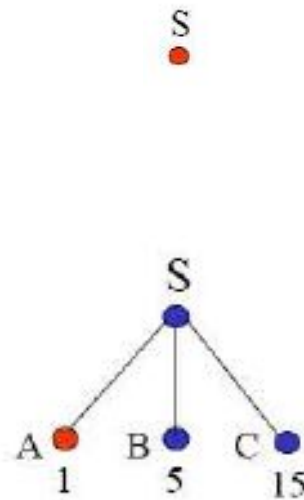
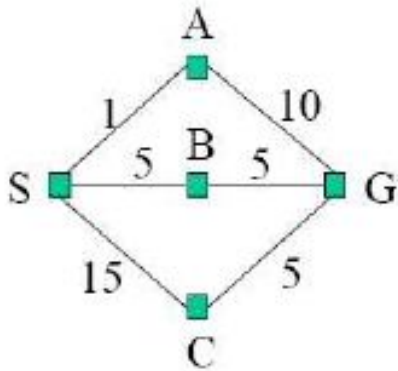
# Busca de custo uniforme

- Ir de S até G



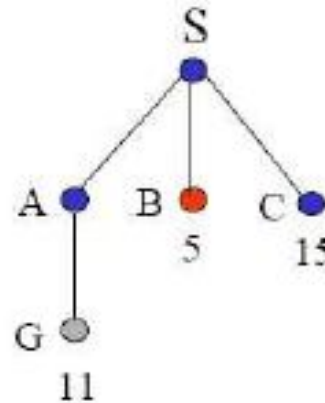
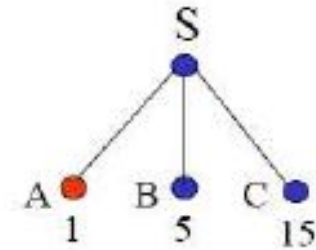
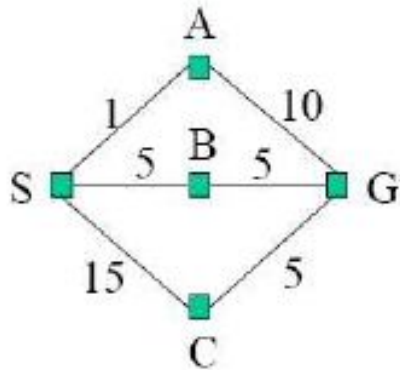
# Busca de custo uniforme

- Ir de S até G



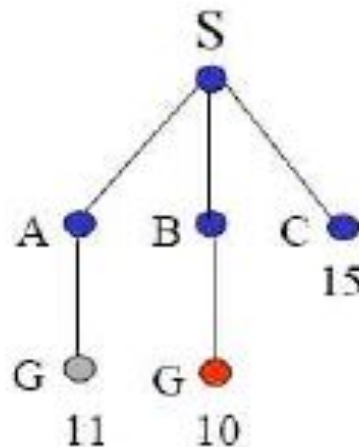
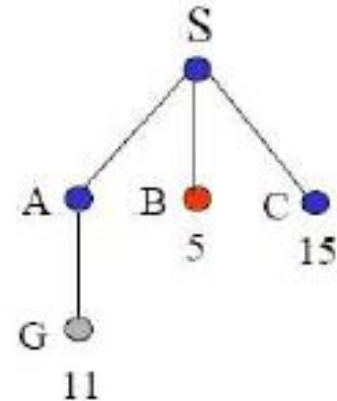
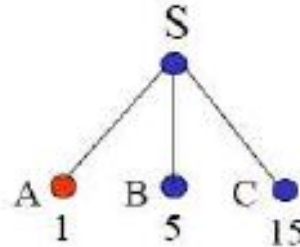
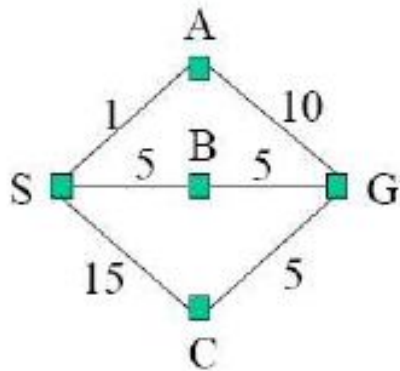
# Busca de custo uniforme

- Ir de S até G



# Busca de custo uniforme

- Ir de S até G



F = [S:0]  
F = [A:1,B:5,C:15]  
F = [B:5, G:11, C:15]  
F = [G:10, C:15]  
S - B - G

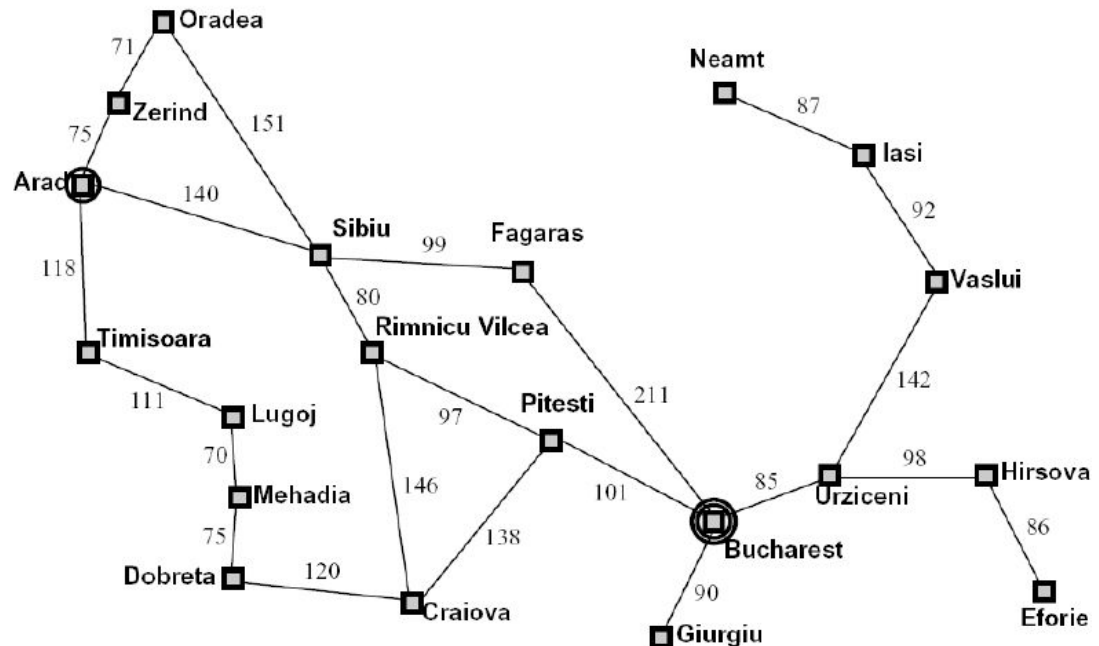
# Algoritmo

```
function UNIFORM-COST-SEARCH(problem) returns uma solução ou falha
  node = um nó com state = problem.initialState
  pathCost = 0
  frontier = lista de prioridades, ordenada por pathCost
  frontier.add(node)
  explored = emptySet
  loop do
    if frontier.empty() return falha
    node = frontier.pop()
    if problem.goalTest(node.state) return solution(node)
    explored.add(node.state)
    for each action in problem.actions(node.state) do
      child = child-node(problem, node, action) // atualiza pathCost a partir de node
      if child.state não está em exploredSet ou em frontier
        frontier.add(child)
      else if child.state está em frontier com pathCost mais alto
        replace tal node em frontier com child
```



F = [Arad:0]  
 S = {Arad}  
 F = [Zer:75, Timi:118, Sibiu:140]  
 S = {Arad, Zerind}  
 F = [Timi:118, Sibiu:140, Oradea:146]  
 S={Arad, Zerind, Timisoara}  
 F = [Sibiu:140, Oradea:146, Lugoj:229]  
 S= {Arad, Zerind, Timisoara,Sibiu}  
 F = [Oradea:146, RV: 220, Lugoj:229, Fag:239]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea}  
 F = [RV: 220, Lugoj:229, Fag:239]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV}  
 F = [Lugoj:229, Fag:239, Pitesti:317, Craiova:366]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj}  
 F = [Fag:239, Mehadia: 299, Pitesti:317, Craiova:366]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj, Fagaras}  
 F = [Mehadia: 299, Pitesti:317, Craiova:366, Buch:450]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj, Fagaras, Mehadia}  
 F = [Pitesti:317, Craiova:366, Dobreta:374, Buch:450]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj, Fagaras, Mehadia, Pitesti}  
 F = [Craiova:366, Dobreta:374, Buch:418]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj, Fagaras, Mehadia, Pitesti, Craiova}  
 F = [Dobreta:374, Buch:418]  
 S= {Arad, Zerind, Timisoara,Sibiu, Oradea, RV,Lugoj, Fagaras, Mehadia, Pitesti, Craiova, Dobreta}  
 F = [Buch:418]

# usto uniforme



# Avaliação

- Completa?
  - Sim, se o custo de cada passo  $\geq e$ ,  $e > 0$
- Tempo?
  - $O(b^{1+\lceil C^*/e \rceil})$ , onde  $C^*$  é o custo da solução ótima
- Espaço?
  - $O(b^{1+\lceil C^*/e \rceil})$
- Ótima?
  - Sim

# Busca Bidirecional

- Executar duas buscas simultâneas
  - Uma do estado inicial para a frente
  - Outra do estado objetivo para trás
- Podem ser usadas estratégias de busca diferentes
- Espera-se uma redução na complexidade
  - $b^{d/2} + b^{d/2} < b^d$

# Busca Bidirecional

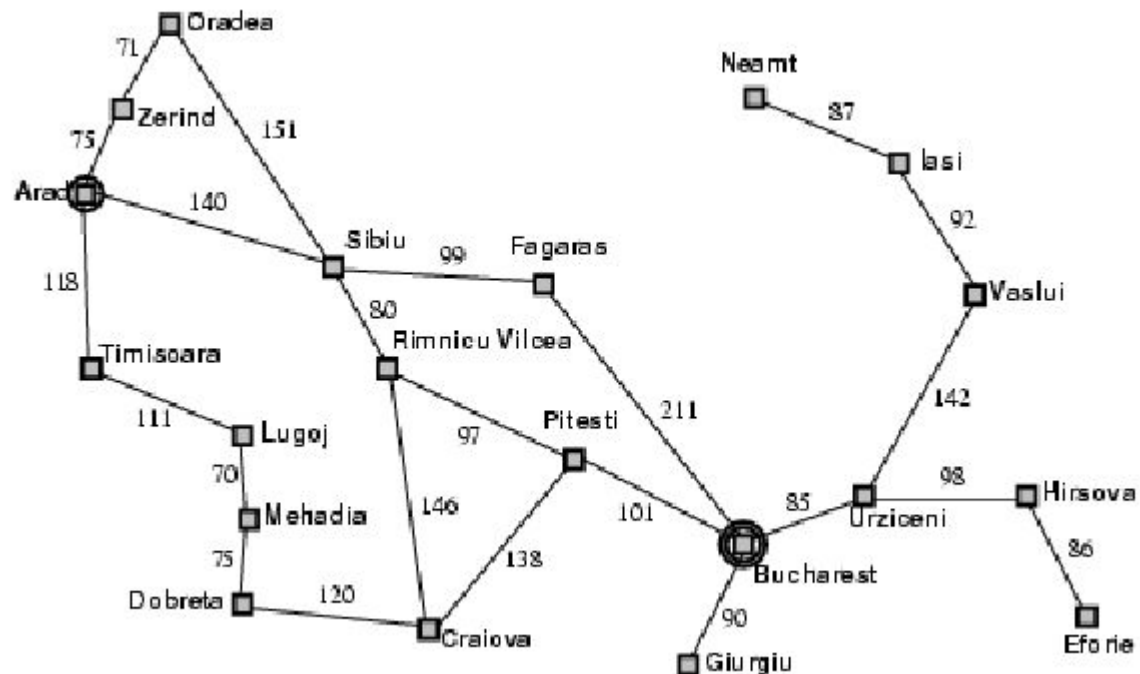
- Espera-se que as duas buscas se encontrem em algum momento
  - e é nesse momento que elas param
  - teste de meta é substituído por
    - existe interseção entre as frontiers?
  - Não garante que a primeira solução é a melhor

# Busca Bidirecional

- Como executar a busca de trás para a frente?
  - computar os predecessores de um nó, ao invés dos sucessores
    - se o modelo de transição é reversível, predecessores de um estado são seus sucessores
- Busca bidirecional no problema das  $n$  rainhas?

# Exercício

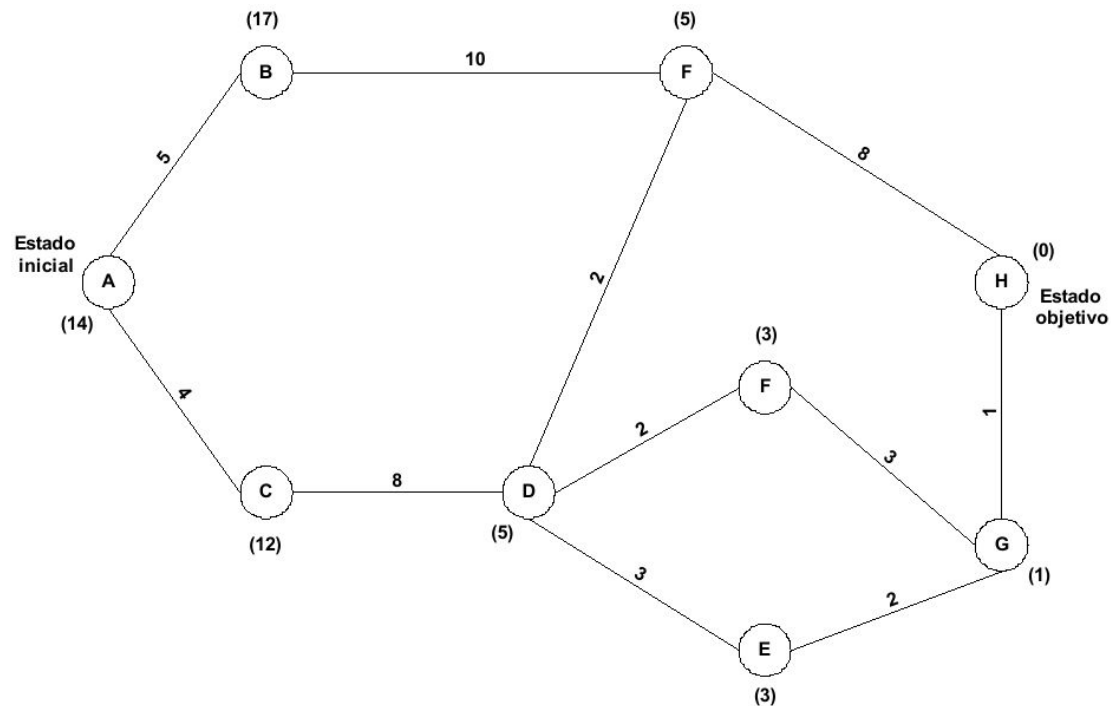
Construir a árvore de busca em largura, profundidade, profundidade limitada (limite = 4), profundidade iterativa, custo uniforme e busca bidirecional (-> custo uniforme <- largura) para:



OBS: você pode verificar sua resposta com o aplicativo disponibilizado em [aispace.org](http://aispace.org)

# Exercício

Construir as árvores de busca das estratégias em largura, profundidade, profundidade iterativa e custo uniforme, para o grafo abaixo (ignore os valores entre ( )); ignore os valores das arestas nas estratégias diferentes de custo uniforme).



# Exercício

- Encontre a solução para o problema dos Missionários e Canibais usando busca em largura e busca em profundidade.