

Resolução de Problemas Com Procura

Relação com o livro

- Capítulo 3 (3.1,3.2,3.3,3.4)

Sumário

- Agentes que resolvem problemas
- Formulação de problemas
- Exemplos de problemas
- Algoritmos de procura básicos
- Eliminação de estados repetidos
- Procura com informação parcial

Agentes que resolvem problemas

Agentes que resolvem problemas

- Ideia chave
 - Estabelecer um objectivo
 - Tentar atingi-lo
 - Procurar uma sequência de acções que satisfaça o objectivo
 - Executar essa sequência de acções

Roménia: Arad -> Bucareste

■ Férias na Roménia; actualmente em Arad



Formulação Objectivo

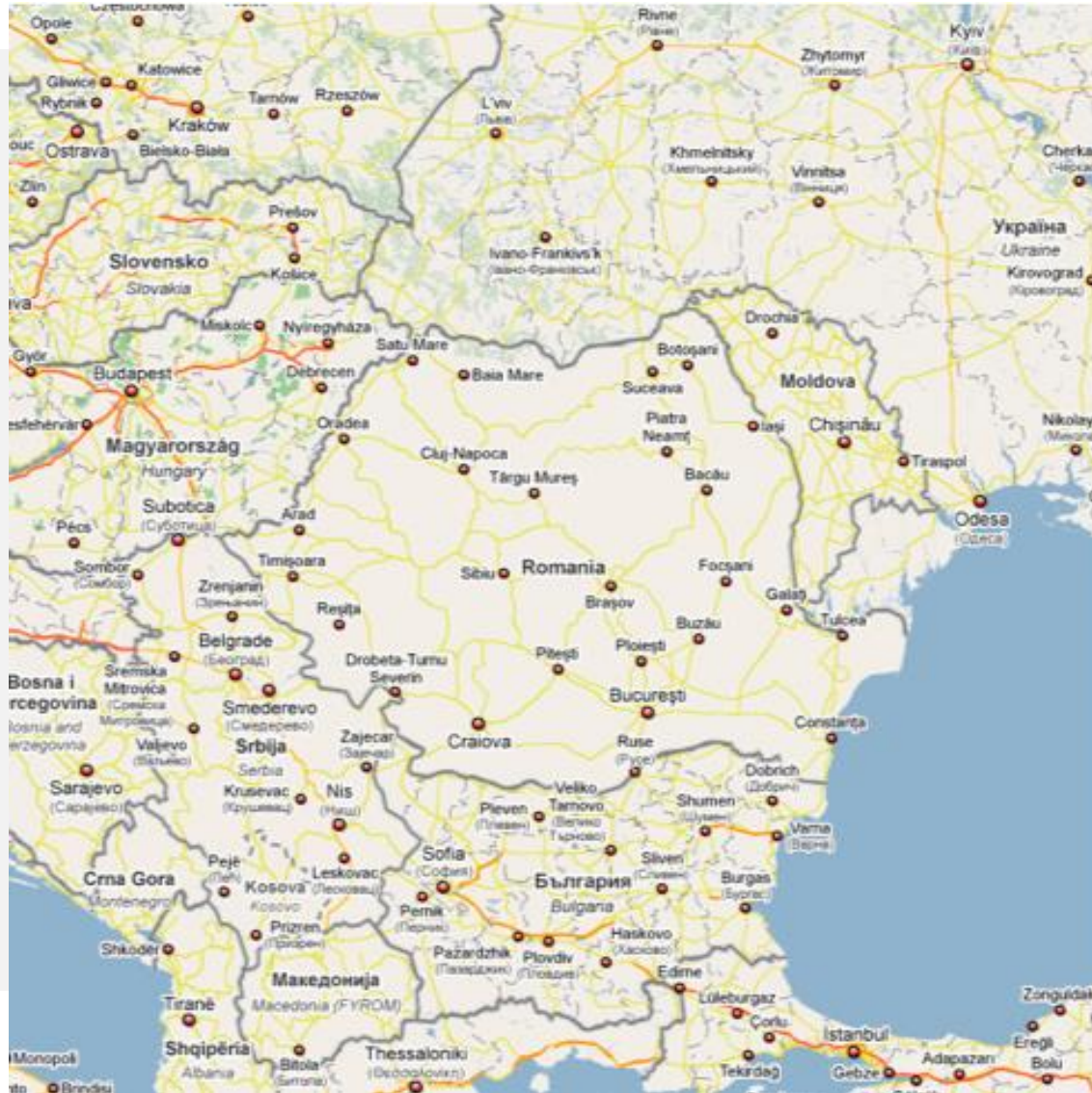
- Vôo de volta para Lisboa parte amanhã de Bucareste
- Formulação Objectivo
 - Estar em Bucarest
 - Objectivos ajudam a simplificar o processo de decisão
 - Limitam as acções a considerar



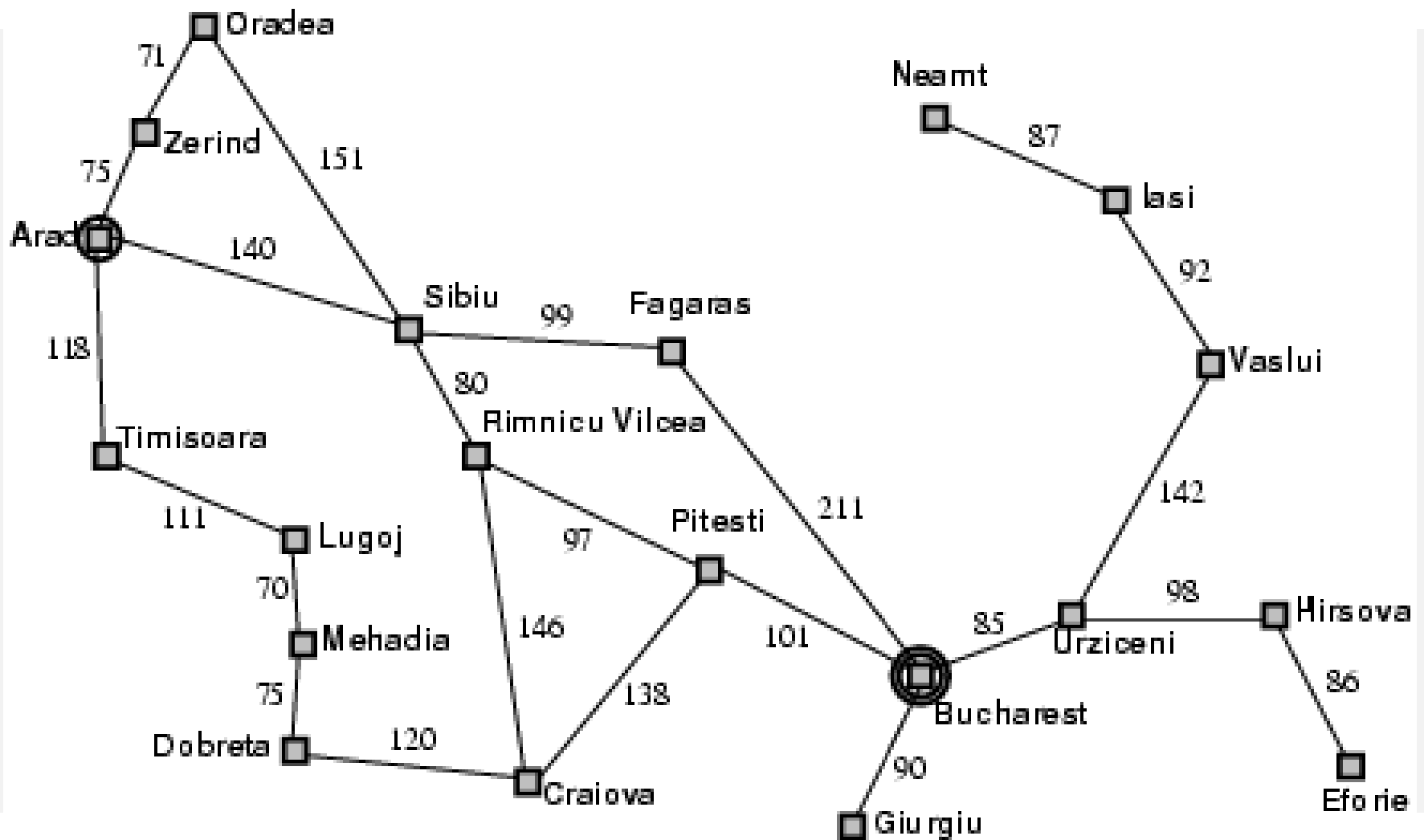
Formulação Problema

- Formulação do problema
 - Processo de decidir que acções e estados a considerar, dado um objectivo
 - Objectivo: estar em Bucareste
 - **Estados**: várias cidades
 - **Acções**: conduzir entre cidades

Exemplo: Roménia



Exemplo: Roménia



- Procura
 - Processo de procurar uma sequência de acções que atinge um objectivo
 - Para isso examina diferentes e possíveis sequências de acções
 - usado por um agente, quando confrontado com várias opções imediatas de valor desconhecido
 - Recebe um problema e devolve uma solução
 - Sequência de acções
 - E.g., Arad → Sibiu → Fagaras → Bucareste

Execução

- Execução
 - Executar a sequência de acções que permitem alcançar o objectivo
 - Pela ordem especificada na sequência
- Agente que resolve problemas
 - Formular Objectivo
 - Formular Problema
 - Procurar
 - Executar

Agentes que resolvem problemas

function SIMPLE-PROBLEM-SOLVING-AGENT (percept) **returns** an action

persistent: *seq*, an action sequence, initially empty
 state, some description of the current world state
 goal, a goal, initially null
 problem, a problem formulation

state \leftarrow UPDATE-STATE (*state*,*percept*)

if *seq* is empty **then**

goal \leftarrow FORMULATE-GOAL(*state*)

problem \leftarrow FORMULATE-PROBLEM(*state*,*goal*)

seq \leftarrow SEARCH(*problem*)

if *seq* = *failure* **then return** a null action

action \leftarrow FIRST(*seq*)

seq \leftarrow REST(*seq*)

return *action*

Tipo de ambiente

- Estático (vs. dinâmico)
 - Ambiente não é alterado enquanto agente efectua formulação e resolução do problema
- Observável (vs. parcialmente observável)
 - Sensores dão acesso ao estado completo do ambiente
- Discreto (vs. contínuo)
 - Número limitado de percepções e acções distintas claramente definidas
- Determinístico (vs. estocástico)
 - Estado seguinte é determinado em função do estado actual e da acção executada pelo agente; fase de execução independente das percepções!

Formulação do Problema

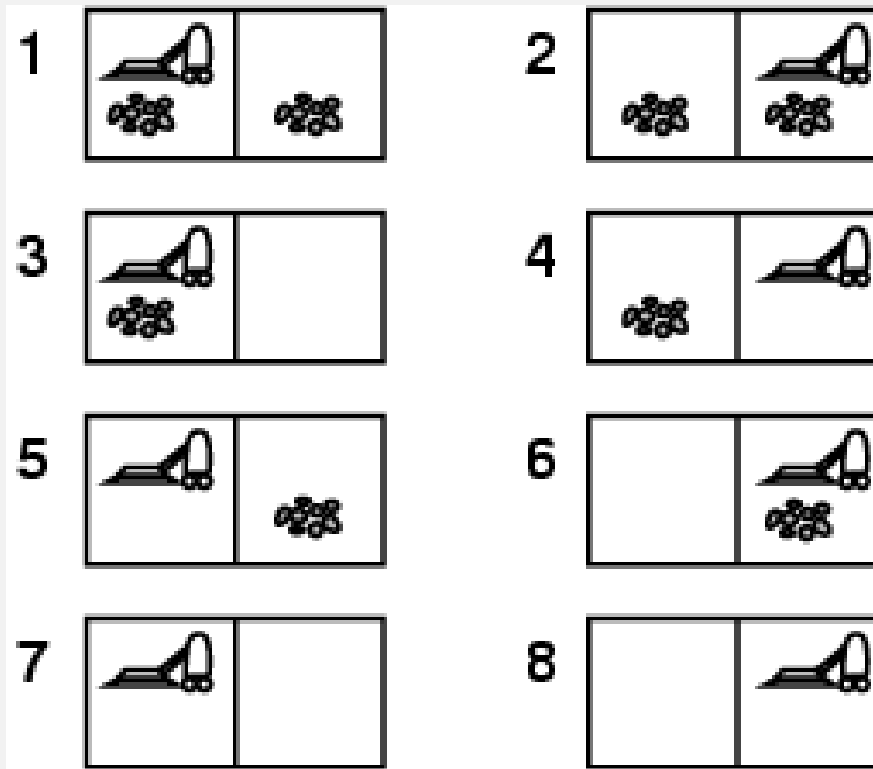
Formulação do Problema

- Problema definido por 5 componentes
 - **Estado inicial**
 - onde se encontra o agente
 - **Acções**
 - função que dado um estado e , retorna o conjunto de acções que podem ser executadas em e
 - **Modelo de transição/Resultado**
 - função que dada uma acção a e um estado e , retorna o estado e' que resulta de executar a em e
 - **Teste objectivo**
 - função que dado um estado e , retorna *Verdade* se o estado e for um estado objectivo, e *Falso* c.c.
 - **Custo caminho**
 - função que atribui um custo numérico a cada caminho (sequência de acções a partir do estado inicial)
 - Escolhida em função do desempenho pretendido para o agente (rapidez \rightarrow km)

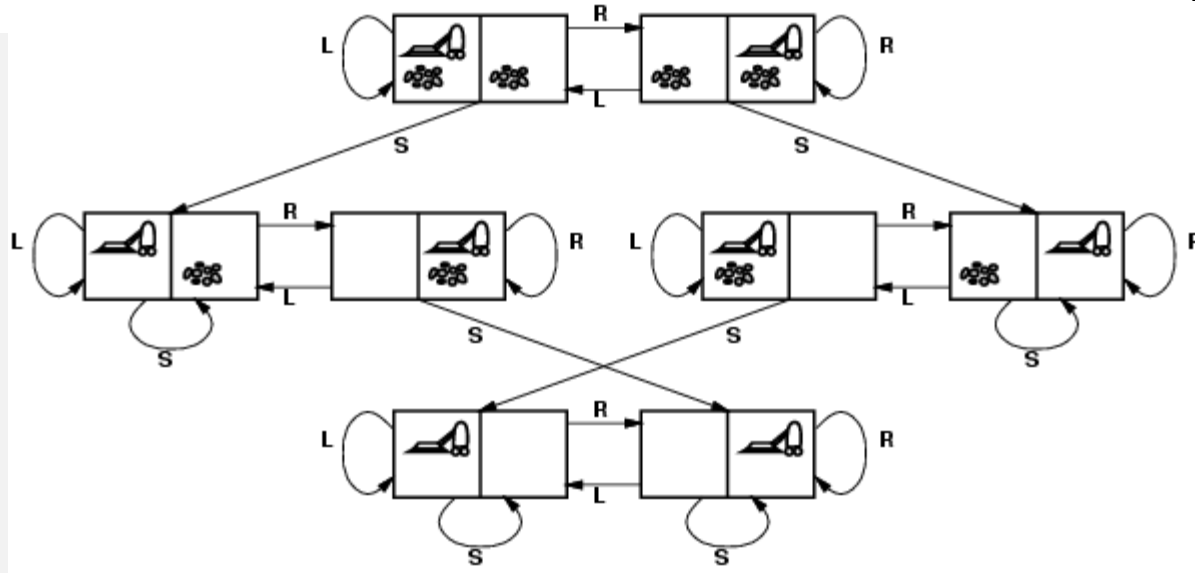
Formulação do Problema

- Problema ir de Arad – Bucareste
 - Estado inicial
 - $Em(Arad)$
 - Acções
 - Caminhos entre cidades
 - E.g. considerando o estado $e = Em(Arad)$
 - $accoes(e) = \{Ir(Sibiu), Ir(Timisoara), Ir(Zerind)\}$
 - Resultado
 - $Resultado(Em(Arad), Ir(Zerind)) = Em(Zerind)$
 - Teste objectivo
 - $Objectivo(e) = (e == Em(Bucareste))$
 - Custo caminho
 - Distância percorrida (em Km) desde o estado inicial

Mundo do aspirador: estados



Espaço de Estados: grafo



- estados? sujidade e localização do robot
- estado inicial? qualquer estado pode ser o inicial
- acções? esquerda, direita, aspirar
- resultado? ver imagem acima
- teste objectivo? não haver sujidade em nenhuma posição
- custo de caminho? 1 por cada acção no caminho

Exemplo: 8-puzzle

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

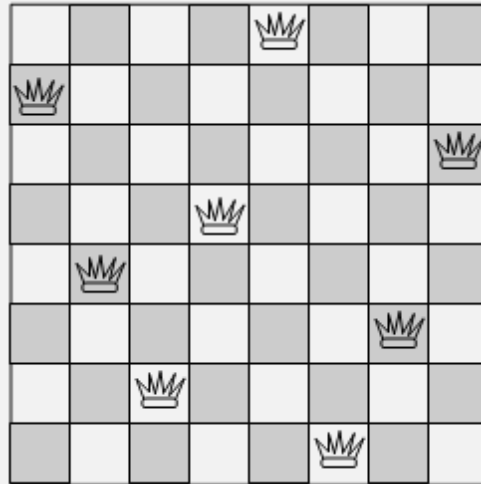
Start State

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Goal State

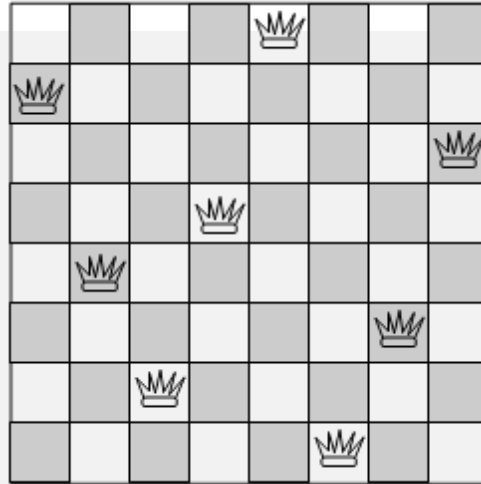
- estados? localização das peças
 - estado inicial?
 - acções? mover espaço esq, dir, cima, baixo
 - resultado? troca de peças resultante do movimento
 - teste objectivo? = estado objectivo (dado)
 - custo de caminho? 1 por movimento no caminho
- [Nota: solução óptima para a família n -Puzzle é NP-difícil]

Exercício: 8-rainhas



- Colocar 8 rainhas num tabuleiro 8x8 tal que nenhuma rainha ataca outra rainha (uma rainha ataca outra rainha se estiver na mesma linha, na mesma coluna ou na mesma diagonal)

Solução: 8-rainhas

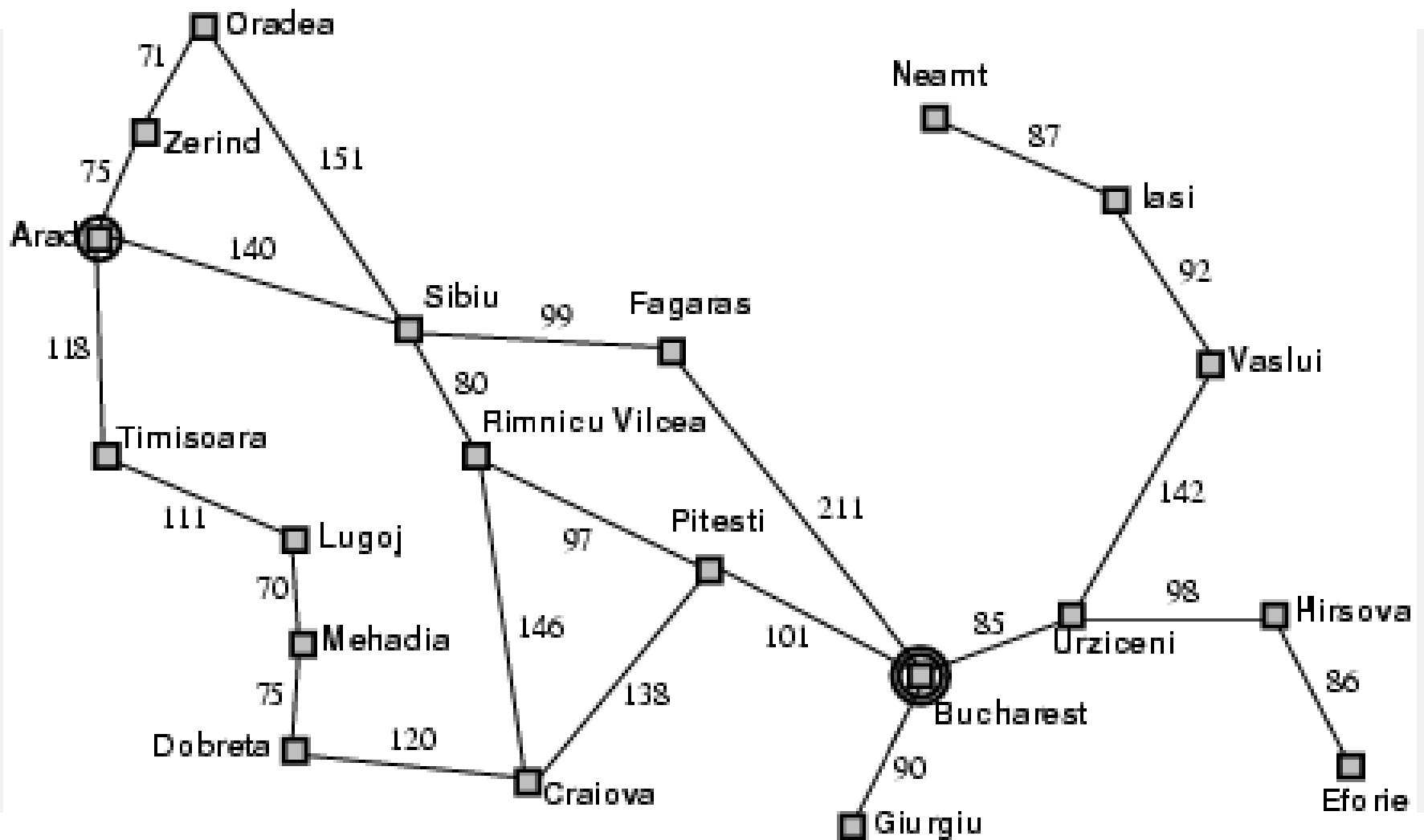


- estados? qualquer tabuleiro de 8x8 com n rainhas ($0 \leq n \leq 8$), com 1 rainha por coluna, e colocadas nas n colunas mais à esquerda
- estado inicial? tabuleiro sem rainhas
- acções? adicionar uma rainha na coluna mais à esquerda não preenchida (de modo a que não seja atacada por nenhuma outra rainha)
- resultado? tabuleiro com a rainha adicionada
- teste objectivo? 8 rainhas no tabuleiro, nenhuma atacada
- custo caminho? 1 por movimento

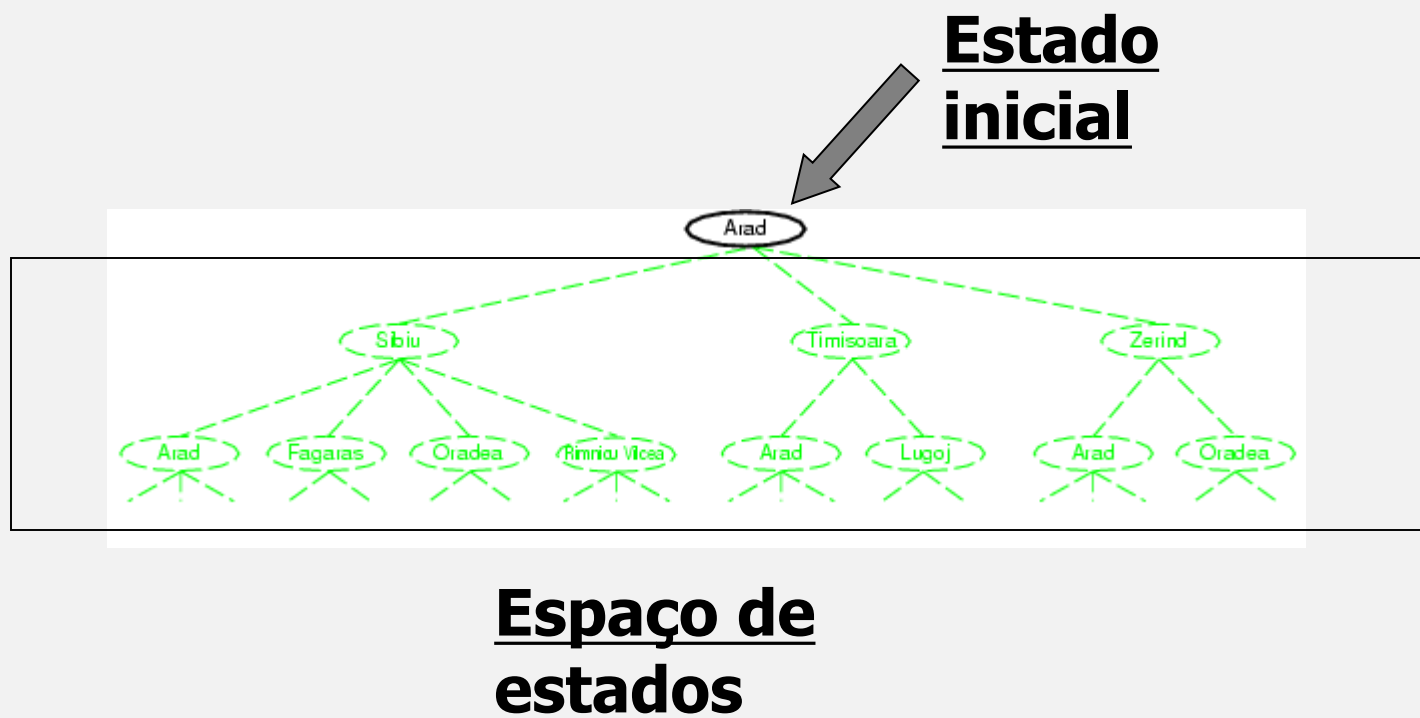
Procura

- Uma vez definido o problema:
 - Precisamos de procurar uma solução
 - Procurando no espaço de estados do problema
- Algoritmos de procura consideram as diferentes sequências de acções possíveis
 - Começando a partir do estado inicial
 - A isto chama-se uma Árvore de Procura

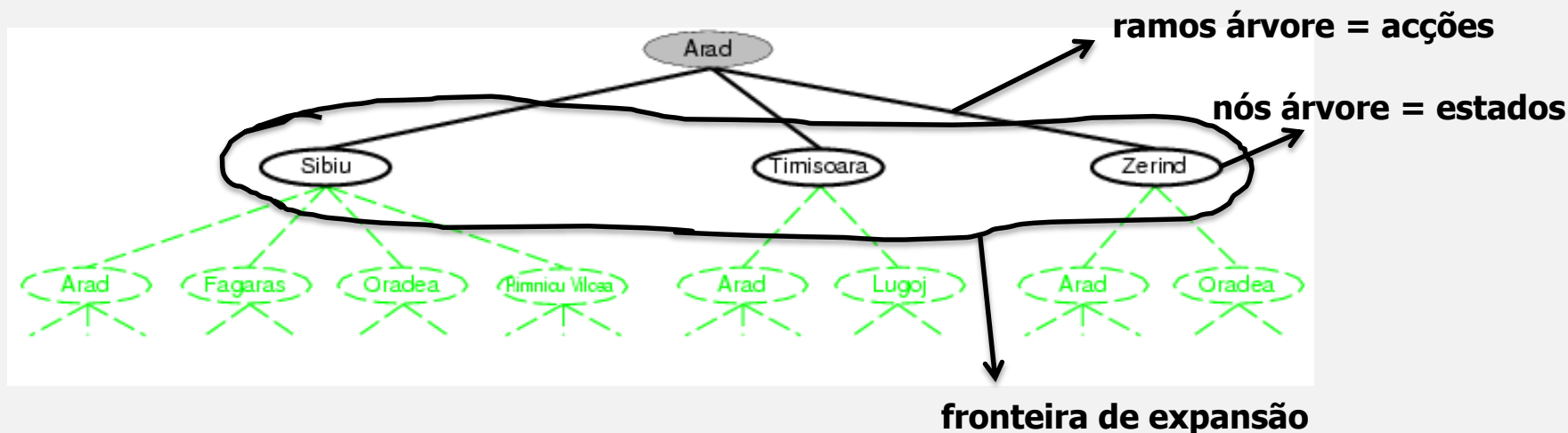
Exemplo: Roménia



Árvore de Procura: exemplo



Árvore de Procura: exemplo



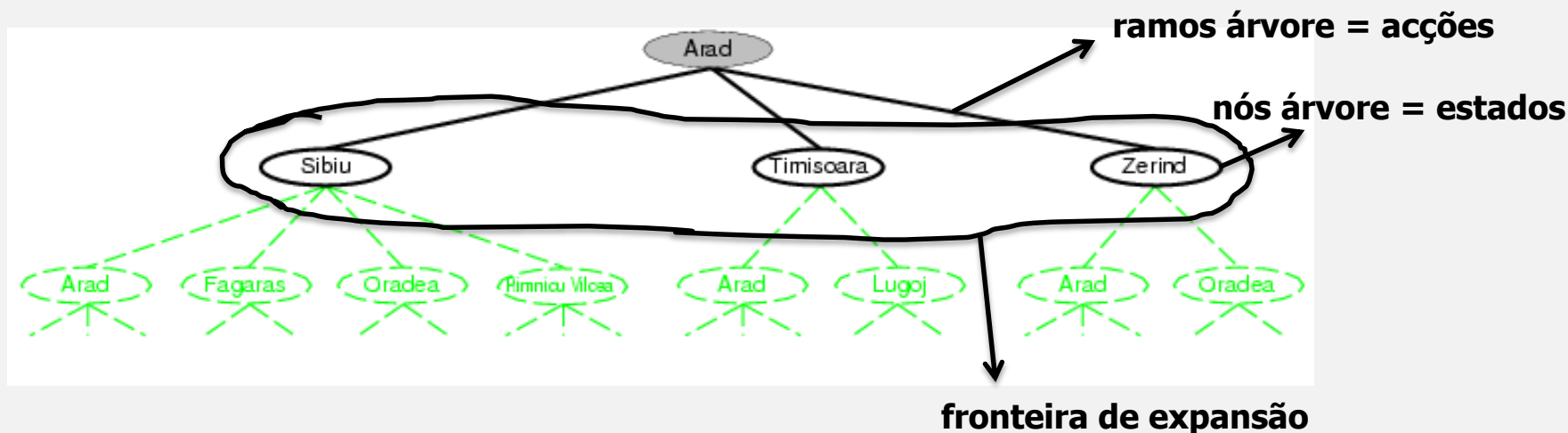
- Existe um único nó folha que pode ser expandido: Arad
- Expandir um nó:
processo de aplicar todas as acções possíveis ao estado, gerando um novo conjunto de estados
- Nós resultantes da expansão do nó Arad: Sibiu, Timisoara, Zerind
- **Fronteira de expansão:** nós gerados que ainda não foram expandidos.
também chamada de **lista de nós abertos**

Árvore de Procura (Algoritmo)

```
function TREE-SEARCH (problem) returns a solution, or failure
  initialize the frontier using the initial state of the problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

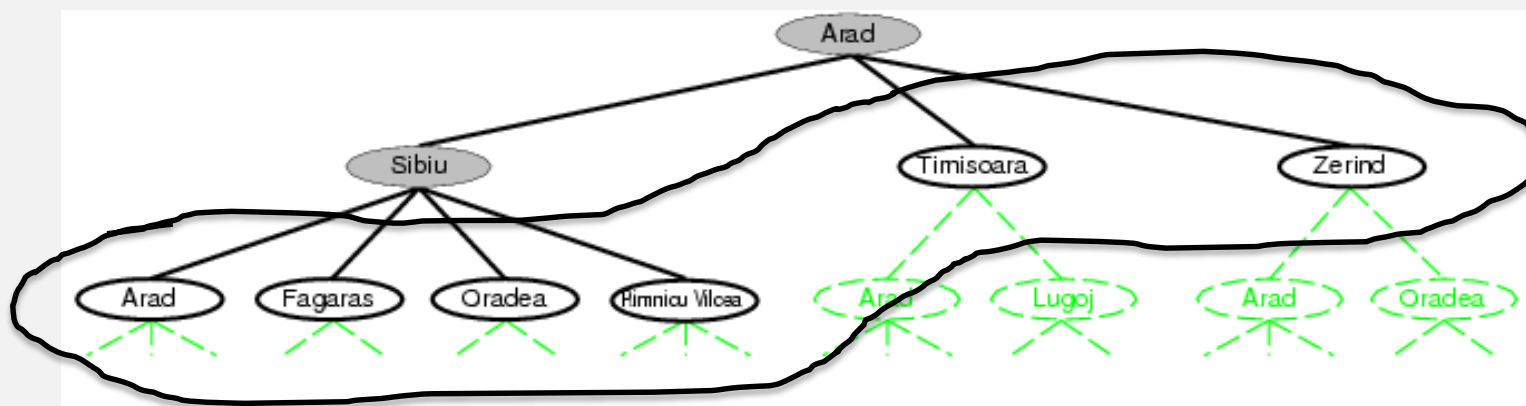
Russel & Norvig, 3rd Ed.

Árvore de Procura: exemplo



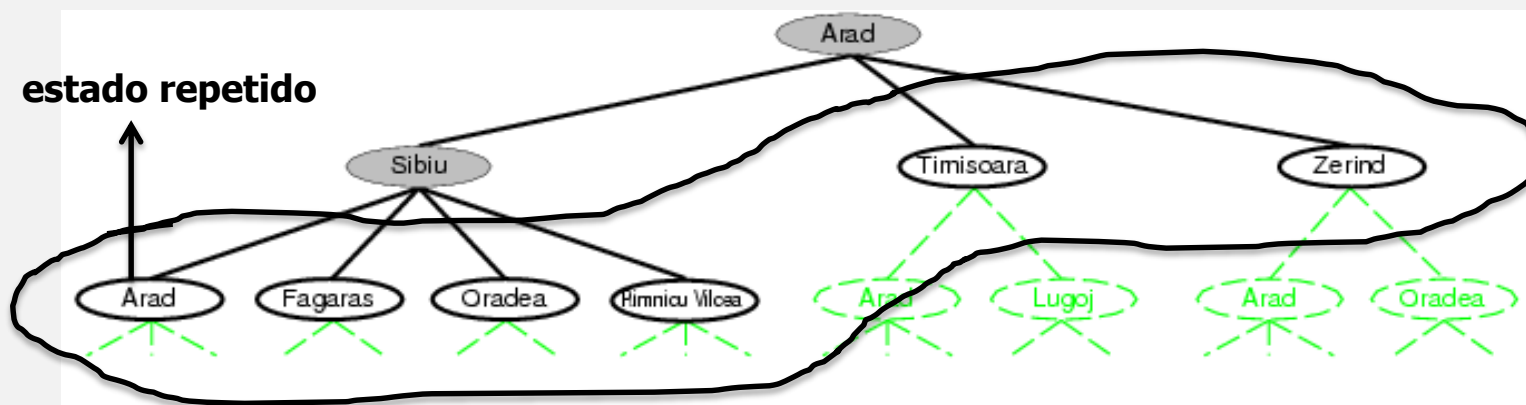
- Existem três nós na fronteira que podem ser expandidos: *Sibiu*, *Timisoara*, *Zerind*
- Suponha que o nó *Sibiu* é escolhido pela **estratégia** de procura como o próximo nó a ser expandido

Árvore de Procura: exemplo



- Nó *Sibiu* é removido da fronteira (já foi expandido)
- Nós resultantes da expansão do nó *Sibiu* são adicionados à fronteira

Árvore de Procura: problema



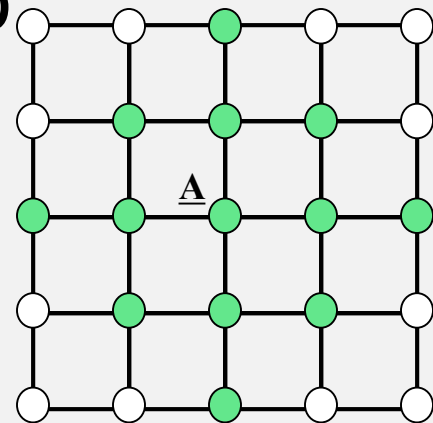
- **Potencial problema do algoritmo Árvore de Procura**
- **estado Arad repetido**
- **existência de caminhos cíclicos dá origem a árvores de procura infinita**

Problema: Estados Repetidos

- No entanto detectar estados repetidos pode ser importante por outra razão

- Problema em Grelha

- Cada estado tem 4 sucessores
- Árvore profundidade d tem 4^d folhas
- Mas só há cerca de $2d^2$ estados distintos



- Para $d = 20$, um bilião de nós face a 800 estados distintos
- Neste exemplo, detectar estados repetidos reduz brutalmente o espaço de estados

Evitar estados repetidos

- Algoritmos que esquecem a sua história estão destinados a repeti-la
- Algoritmo Árvore de Procura extendido com informação acerca dos nós já expandidos
 - Conjunto de explorados
 - Também designado de lista de nós fechados
 - Nós gerados equivalentes a nós gerados anteriormente são descartados em vez de adicionados à fronteira

Evitar estados repetidos

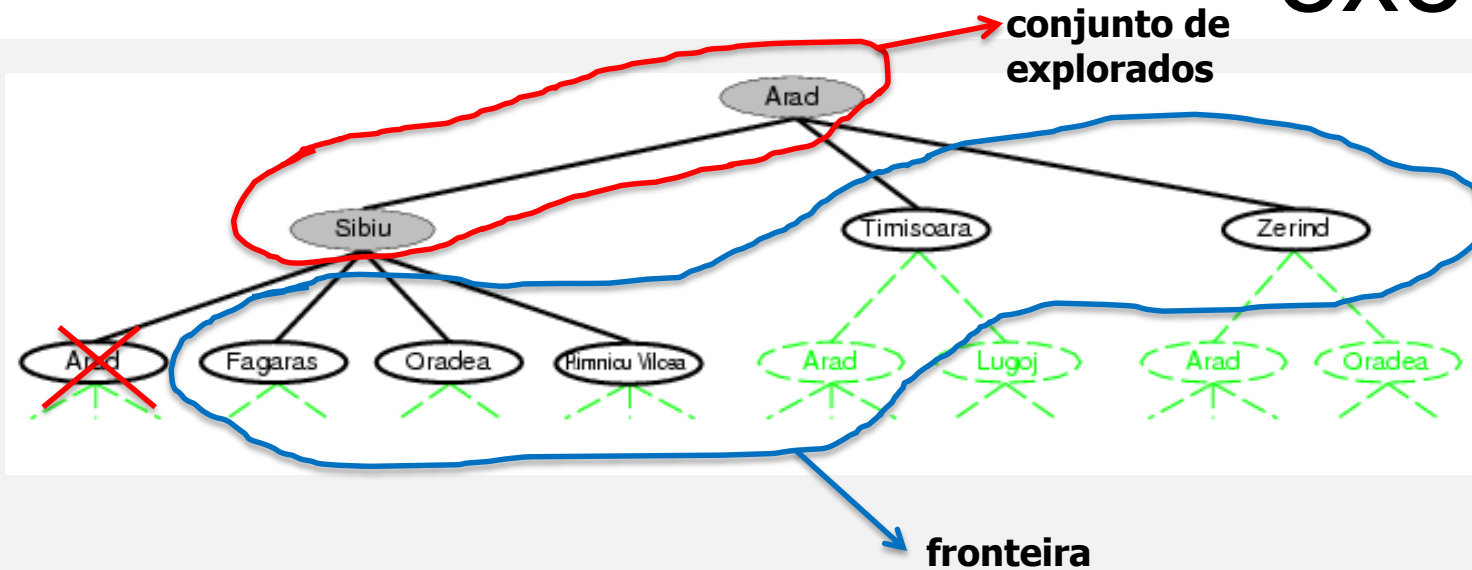
- Todos os nós gerados têm que ser mantidos em memória
 - Complexidade espacial exponencial
 - Complexidade temporal linear se se usar uma “Hash-Table”
 - É preciso comparar
 - Custo de guardar e comparar
 - Custo de re-expandir os nós

Grafo de Procura (Algoritmo)

```
function GRAPH-SEARCH (problem) returns a solution, or failure
  initialize the frontier using the initial state of the problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    but only for resulting nodes not in the frontier or explored set
```

Russel & Norvig, 3rd Ed.

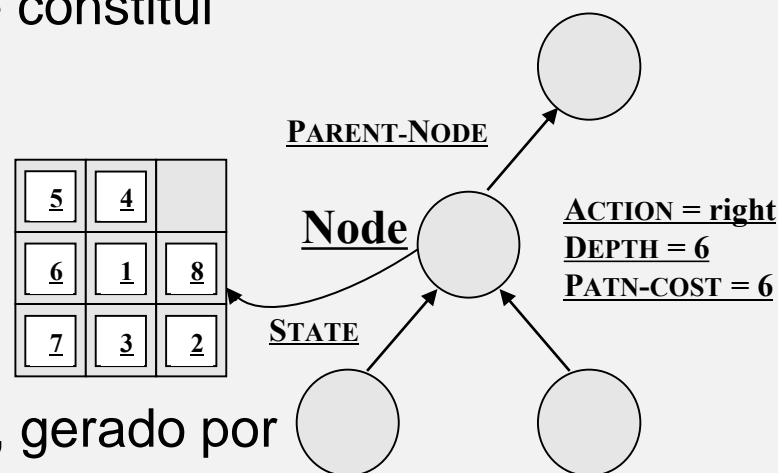
Grafo de Procura: exemplo



Estruturas usadas em Procura

• Nós vs Estados

- Um **estado** é a representação de uma configuração física
- Um **nó** é uma estrutura de dados que constitui parte de uma árvore de procura
 - n.State
 - n.Parent
 - n.Action
 - n.Path-Cost
 - n.Depth
- Dois nós podem ter o mesmo estado, gerado por sequências de acções diferentes



Como gerar um nó

```
function CHILD-NODE (problem, parent, action) returns a node
  return a node with
    STATE = problem.Result(parent.State,action)
    PARENT = parent
    ACTION = action
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.State,action)
    DEPTH = parent.DEPTH + 1
```

Russel & Norvig, 3rd Ed.

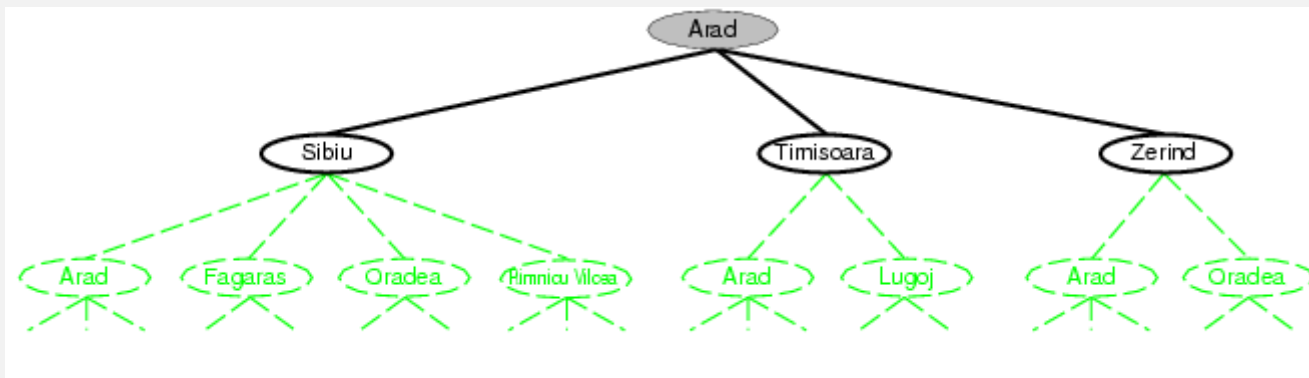
Estruturas usadas em Procura

- Representação Fronteira
 - Conjunto de nós
 - Representação mais intuitiva
 - A estratégia de procura é uma função que selecciona o próximo nó a expandir
 - Fila de nós
 - Representação mais eficiente
 - Fila encontra-se sempre ordenada por ordem de expansão pela estratégia de procura
 - O primeiro nó da fila é sempre seleccionado para expansão
- Representação Conjunto de explorados
 - Utilização de hash table mais eficiente

Estratégias de Procura

Estratégias de Procura

- Uma estratégia de procura é caracterizada por escolher a **ordem de expansão dos nós**
 - Ou em alternativa a ordem de inserção dos nós na fronteira



Estratégias de Procura

- As estratégias são avaliadas de acordo com 4 aspectos:
 - **Compleitude**: encontra sempre uma solução caso exista (se não existir diz que não há solução)
 - **Complexidade temporal**: número de nós gerados
 - **Complexidade espacial**: número máximo de nós em memória
 - **Optimalidade**: encontra a solução de menor custo
- Complexidade temporal e espacial são medidas em termos de:
 - **b** : máximo factor de ramificação da árvore de procura (branching factor)
 - **d** : profundidade da solução de menor custo (nó com estado inicial tem profundidade 0)
 - **m** : máxima profundidade do espaço de estados (pode ser ∞)

Estratégias de Procura

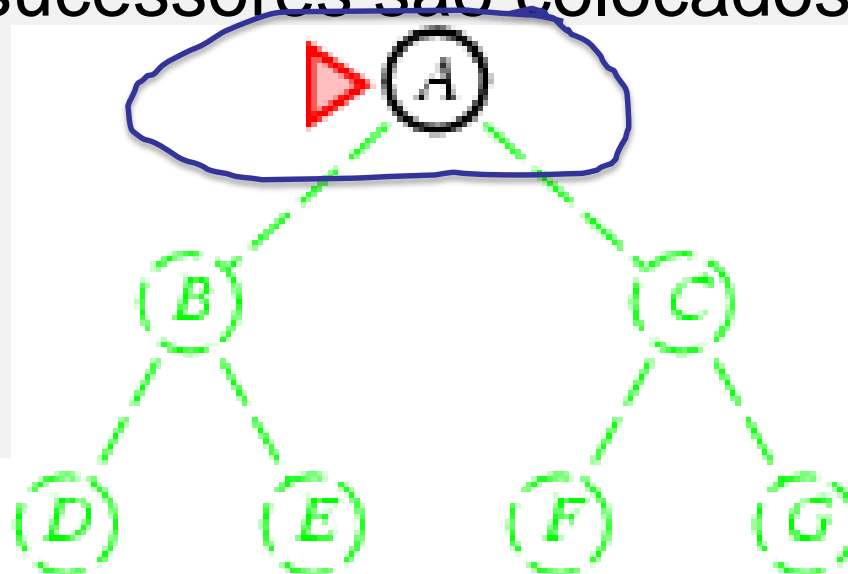
- Complexidade temporal
 - Função do número de **nós gerados** durante a procura
 - Não de nós expandidos.
 - Tempo para expandir um nó cresce com o número de nós por ele gerados.
- Complexidade espacial
 - Função do número de **nós guardados** em memória

Procura Não Informada

- Estratégias de procura **não informada** usam apenas a informação disponível na definição do problema
 - Largura Primeiro
 - Custo Uniforme
 - Profundidade Primeiro
 - Profundidade Limitada
 - Profundidade Iterativa
 - Bi-Direccional
- Também chamadas estratégias de **procura cega**

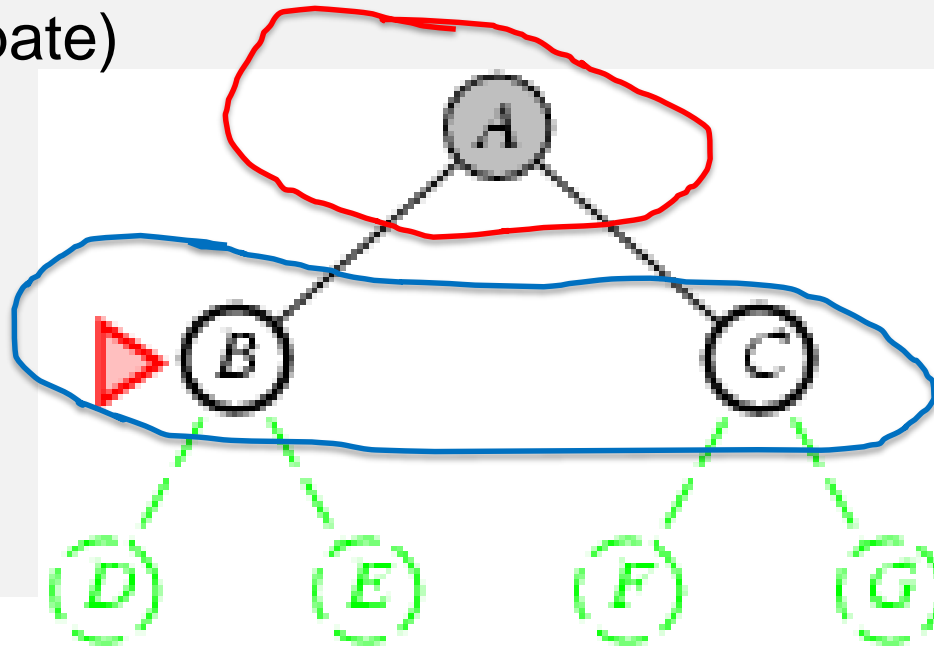
Procura Largura Primeiro

- Expande nó de menor profundidade na fronteira
- Implementação:
 - *nós gerados* colocados numa fila (FIFO), i.e., novos sucessores são colocados no fim da lista



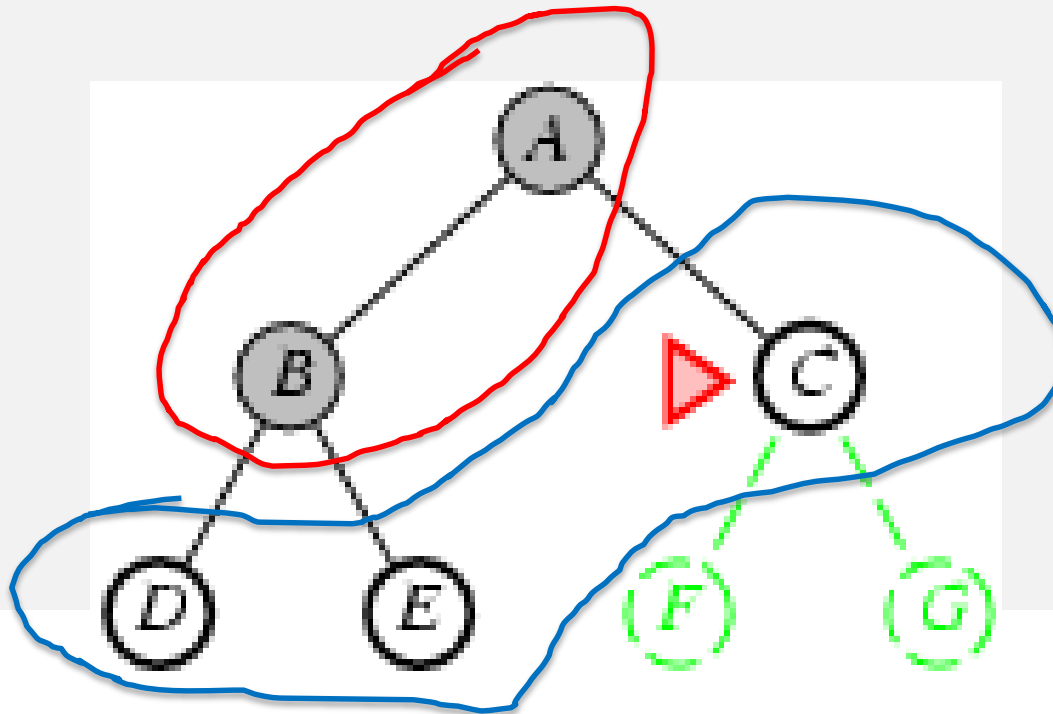
Procura Largura Primeiro

- Nó A é expandido: novos nós B e C
- B está no início da fila: próximo nó a expandir
 - A ordem é irrelevante para nós com a mesma profundidade (pode usar-se ordem alfabética para desempate)



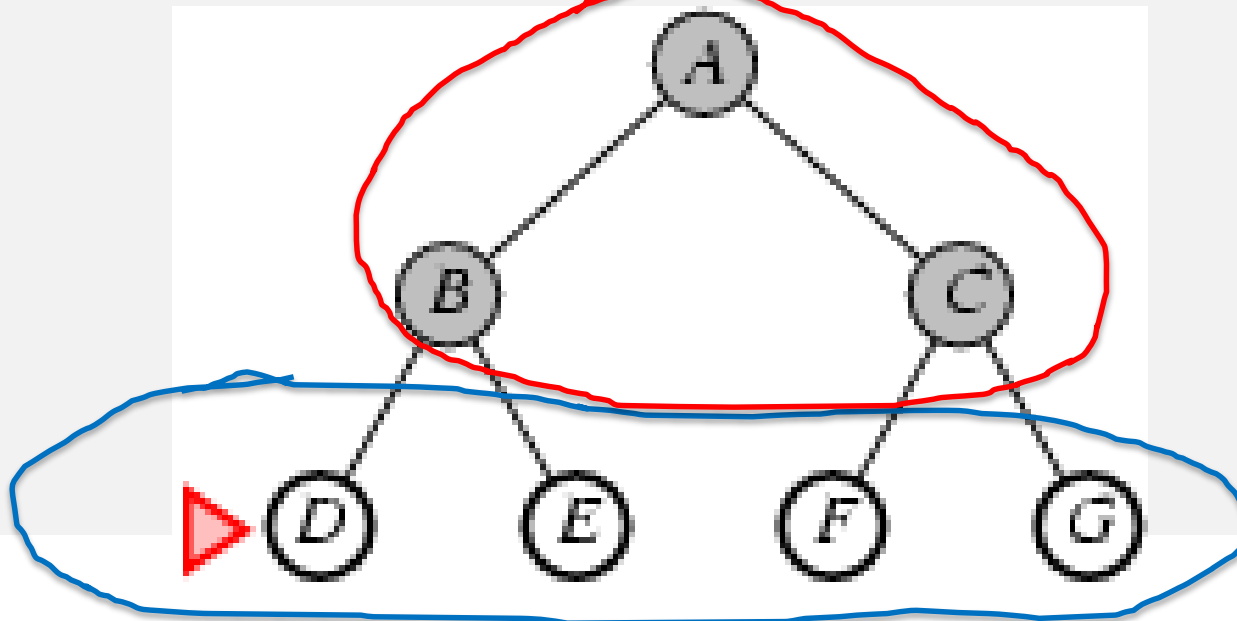
Procura Largura Primeiro

- Nó B é expandido: novos nós D e E
- C está no início da fila; os outros nós na fronteira (D e E) têm maior profundidade



Procura Largura Primeiro

- Estado actual
 - Fronteira: D, E, F, G
 - Nós gerados mas ainda não expandidos
 - Conjunto Explorados: A, B, C



PLP (Algoritmo Árvore)

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

node \leftarrow a node with State = *problem.Initial-State*, Path-Cost = 0

if *problem.Goal-Test*(*node.State*) **then return** Solution(*node*)

frontier \leftarrow a FIFO queue with *node* as the only element

loop do

if EMPTY?(*frontier*) **then return failure**

node \leftarrow POP(*frontier*)

for each *action* **in** *problem.Actions*(*node.State*) **do**

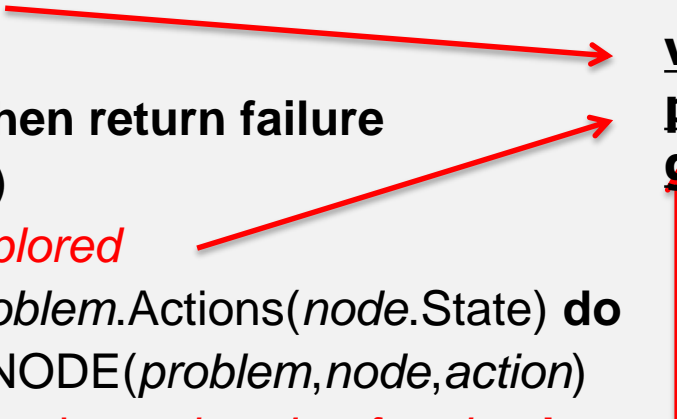
child \leftarrow CHILD-NODE(*problem,node,action*)

if *problem.Goal-Test*(*child.State*) **then return** Solution(*child*)

frontier \leftarrow Insert(*child,frontier*)

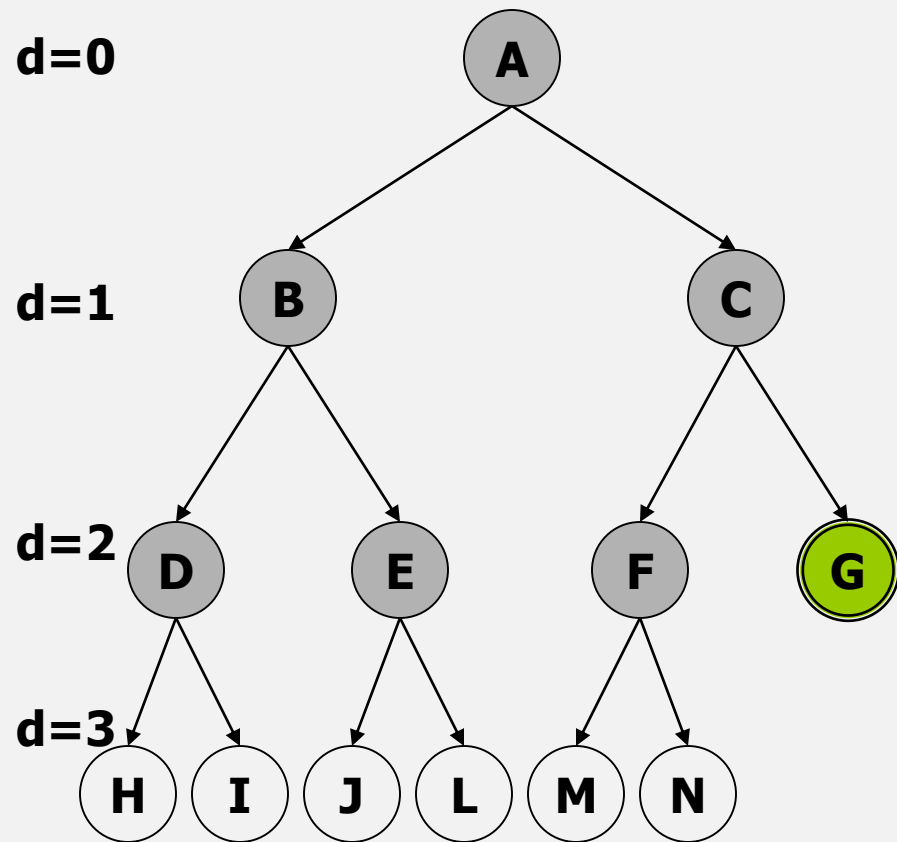
PLP (Algoritmo Grafo)

```
function BREADTH-FIRST-SEARCH (problem) returns a solution, or failure
  node  $\leftarrow$  a node with State = problem.Initial-State, Path-Cost = 0
  if problem.Goal-Test(node.State) then return Solution(node)
  frontier  $\leftarrow$  a FIFO queue with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier)
    add node.State to explored
    for each action in problem.Actions(node.State) do
      child  $\leftarrow$  CHILD-NODE(problem,node,action)
      if child.State is not in explored or frontier then
        if problem.Goal-Test(child.State) then return Solution(child)
        frontier  $\leftarrow$  Insert(child,frontier)
```



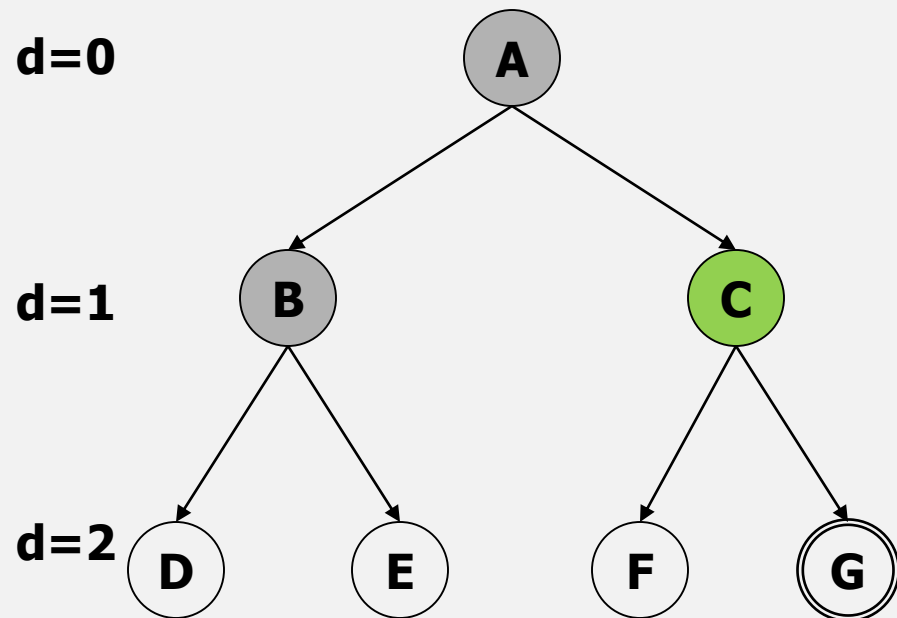
versão
procura em
grafo

PLP com teste na expansão



- Estado objectivo **G**
- Factor de ramificação $b=2$
- Profundidade da solução $d=2$
- Tempo
 - $2^1 + 2^2 + (2^3 - 2)$
 - $O(2^3)$, i.e, $O(b^{d+1})$
- Espaço
 - $\sim 2^3 - 2$
 - $O(2^3)$ i.e, $O(b^{d+1})$

PLP com teste na geração



- Estado objectivo **G**
- Factor de ramificação $b=2$
- Profundidade da solução $d=2$
- Tempo
 - $2^1 + 2^2$
 - $O(2^2)$, i.e, $O(b^d)$
- Espaço
 - $\sim 2^2$
 - $O(2^2)$ i.e, $O(b^d)$

PLP: propriedades

- Completa? Sim (se b é finito)
- Tempo? $b+b^2+b^3+\dots+b^d = O(b^d)$, i.e. exponencial em d
- Espaço? $O(b^d)$ (todos os nós por expandir em memória)
- Óptima?
 - Sim, se custo de caminho for uma função não-decrescente da profundidade (e.g. se custo = 1 por acção)
 - i.e. a solução óptima é a que está mais acima na árvore
 - logo não é óptima no caso geral

b : máximo factor de ramificação da árvore de procura

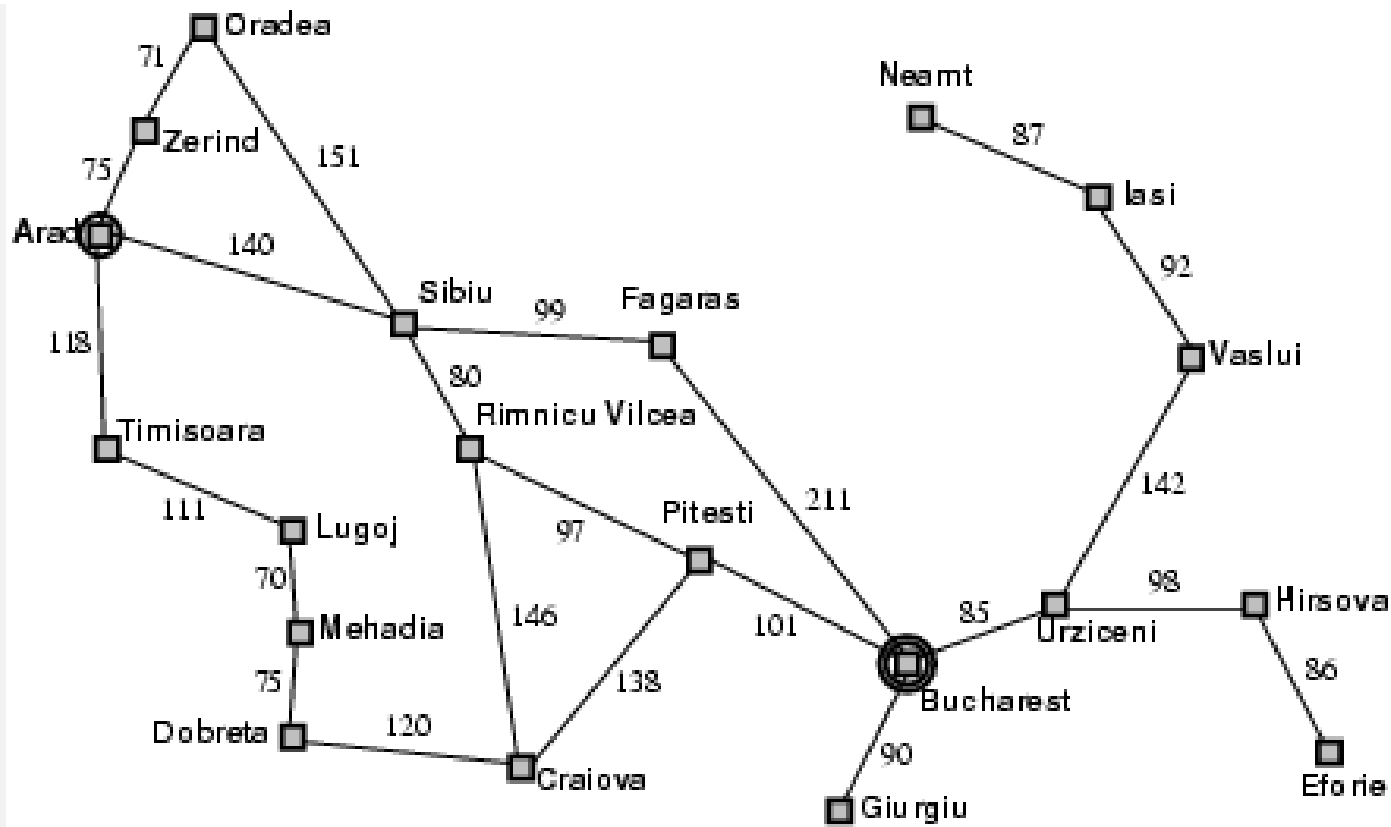
d : profundidade da solução de menor custo

Problema PLP

- **Espaço** é o maior problema (mais do que tempo)
 - Assumindo $b=10$
 - 1M nós por segundo
 - 1000 bytes/nó

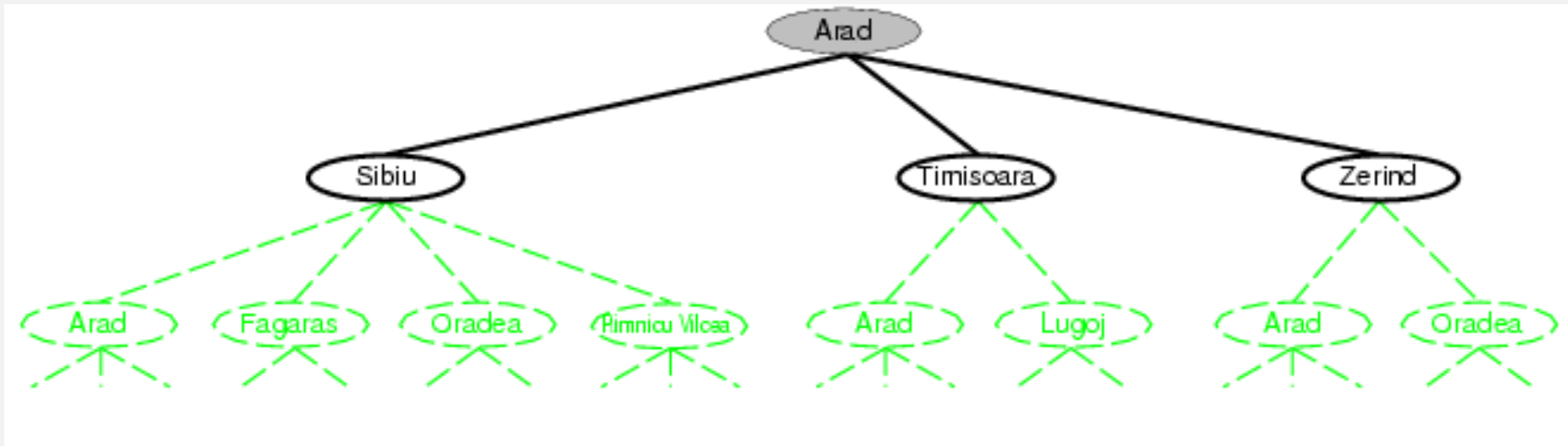
| <u>Profundidade</u> | <u>Nós</u> | <u>Tempo</u> | <u>Memória</u> |
|---------------------|-----------------------------|----------------|---------------------|
| <u>2</u> | <u>110</u> | <u>0.11 ms</u> | <u>107 KB</u> |
| <u>4</u> | <u>11 110</u> | <u>11 ms</u> | <u>10.6 MB</u> |
| <u>6</u> | <u>10^6</u> | <u>1.1 s</u> | <u>1 GB</u> |
| <u>8</u> | <u>10^8</u> | <u>2 min</u> | <u>103 GB</u> |
| <u>10</u> | <u>10^{10}</u> | <u>3 h</u> | <u>10 terabytes</u> |
| <u>12</u> | <u>10^{12}</u> | <u>13 dias</u> | <u>1 petabyte</u> |

Exemplo: Roménia



- Profundidade da Solução?
- N° mínimo de nós expandidos até solução?
- Obs: teste objectivo é feito antes da expansão do nó!

Exemplo: solução

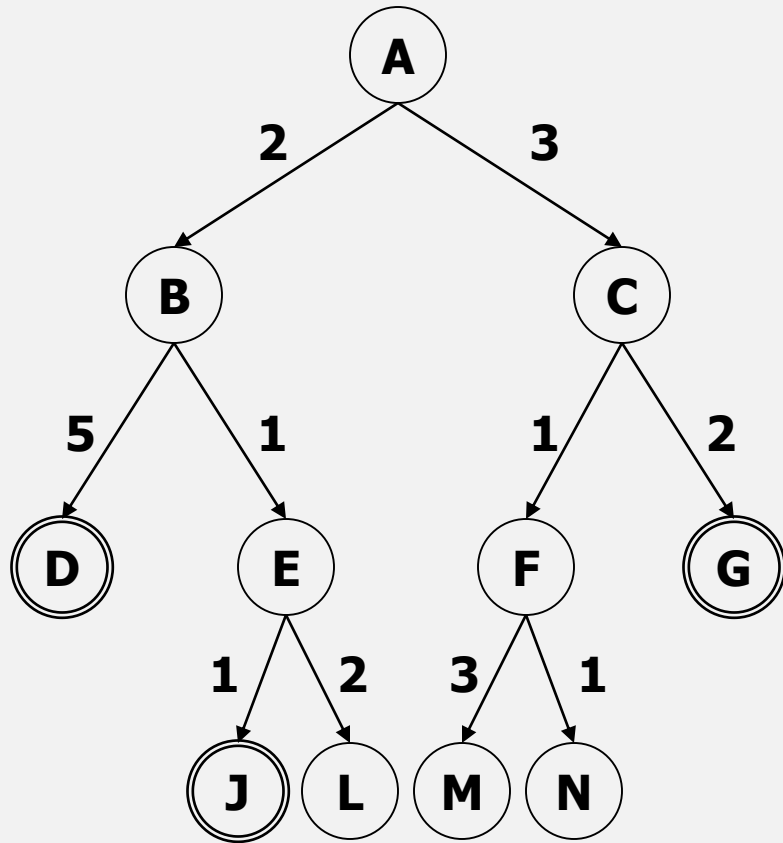


- Profundidade da solução: 3

Procura Custo Uniforme

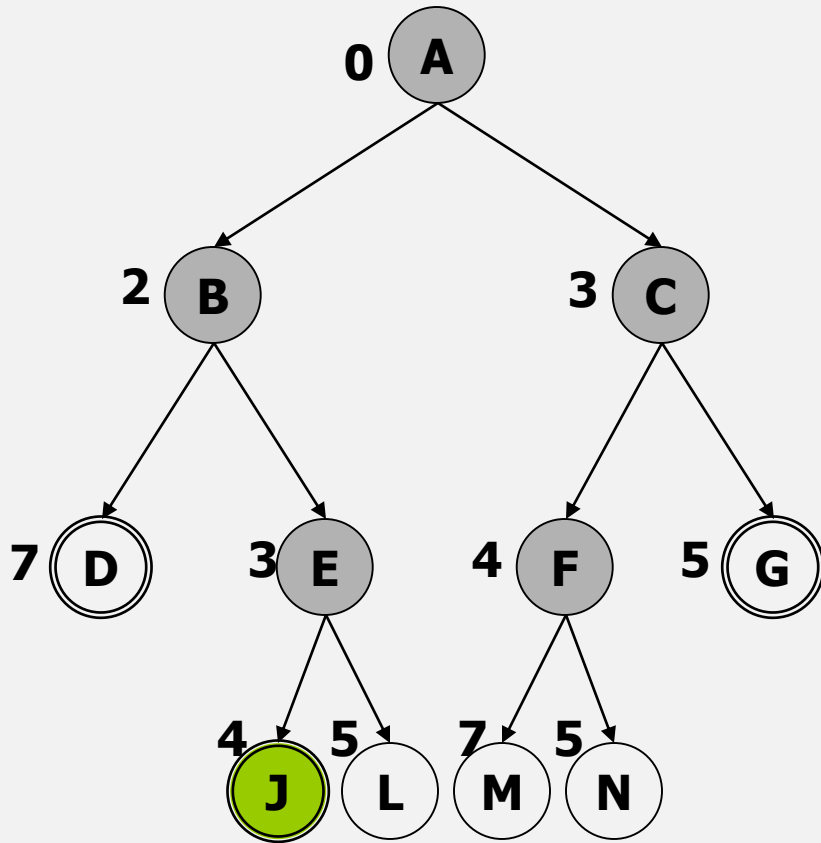
- Expandir nó n na fronteira que tem menor custo $g(n)$
- Implementação:
 - *fronteira* = fila ordenada por custo do caminho
- Equivalente à procura por largura primeiro se todos os ramos tiverem o mesmo custo

Procura Custo Uniforme



- Custo associado a cada ramo
- Ordem de expansão dos nós?
 - Desempate: ordem alfabética
- Solução encontrada?

Procura Custo Uniforme



- Ordem de expansão dos nós?
 - A(0), B(2), C(3), E(3), F(4),
- Solução encontrada?
 - J (custo 4)

P. Custo Uniforme (Árvore)

```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
  node  $\leftarrow$  a node with State = problem.Initial-State, Path-Cost = 0
  frontier  $\leftarrow$  a priority queue ordered by Path-Cost, with node as the only
  element

  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier)
    if problem.Goal-Test(node.State) then return Solution(node)

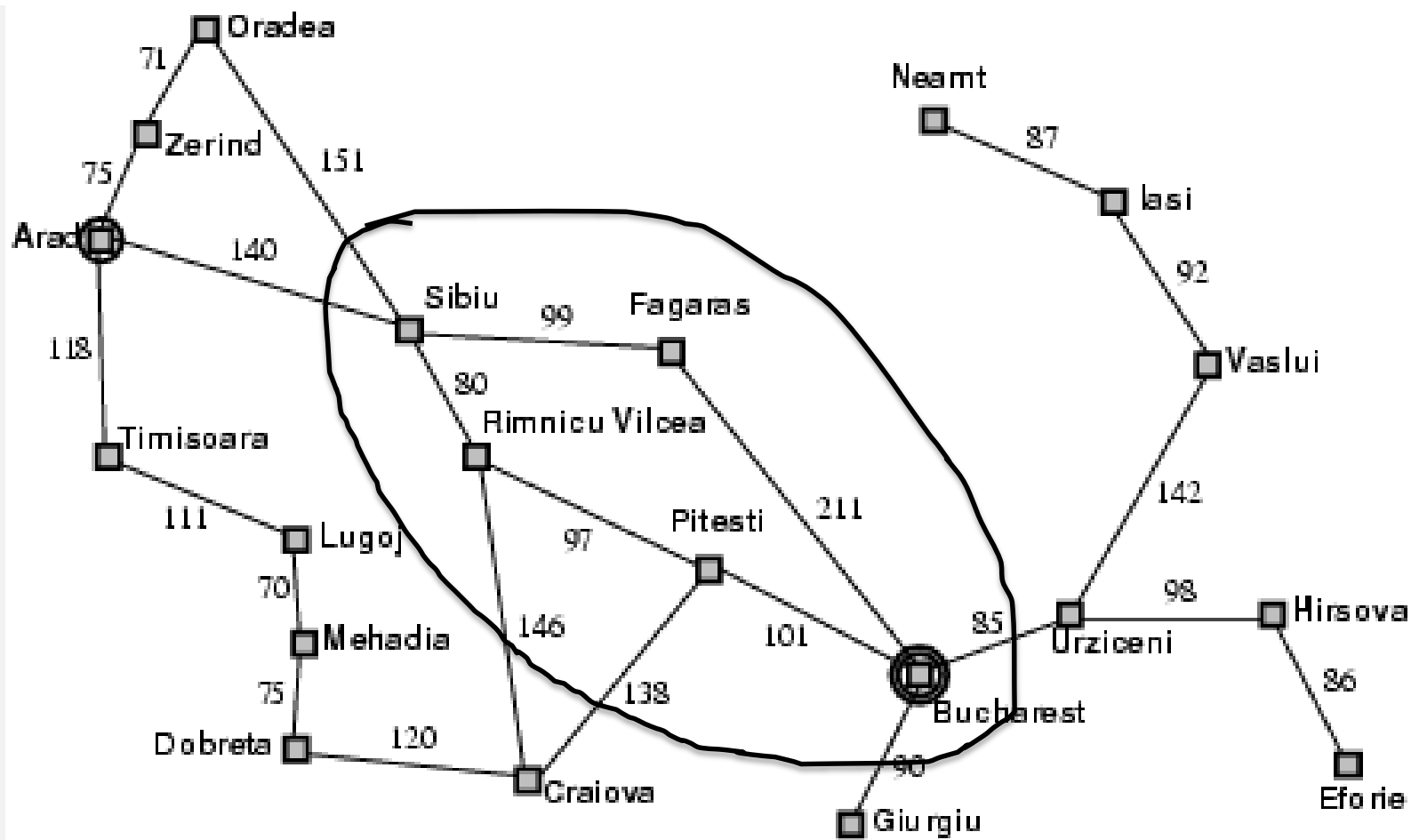
    for each action in problem.Actions(node.State) do
      child  $\leftarrow$  CHILD-NODE(problem,node,action)

      frontier  $\leftarrow$  Insert(child,frontier)
```


P. Custo Uniforme (Grafo)

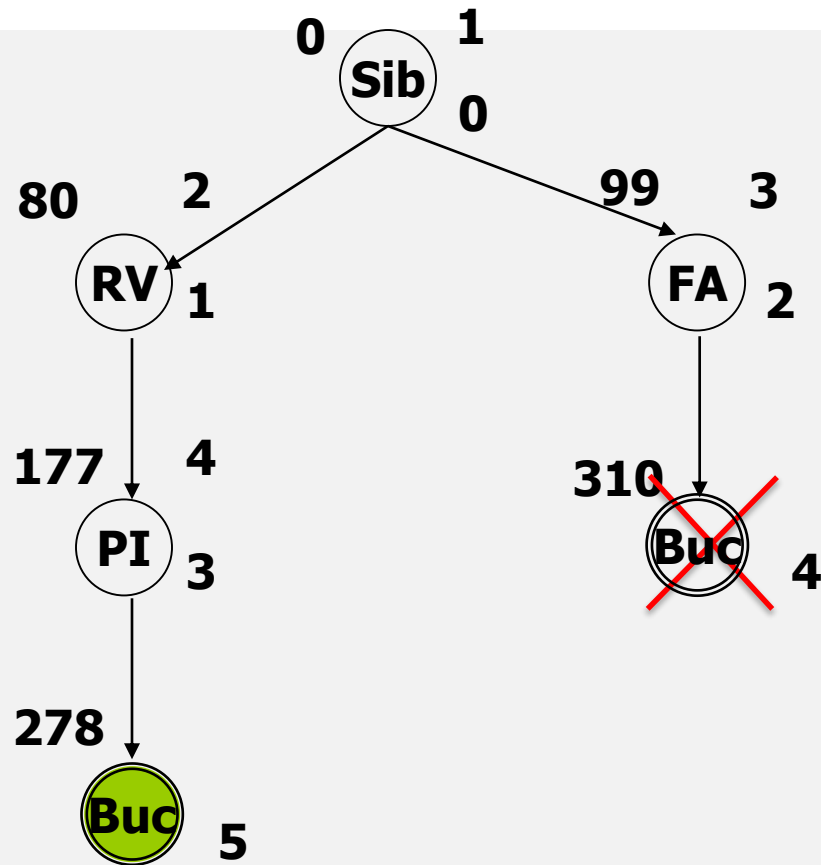
```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
  node  $\leftarrow$  a node with State = problem.Initial-State, Path-Cost = 0
  frontier  $\leftarrow$  a priority queue ordered by Path-Cost, with node as the only
  element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(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  $\leftarrow$  CHILD-NODE(problem,node,action)
      if child.State is not in explored or frontier then
        frontier  $\leftarrow$  Insert(child,frontier)
      else if child.State is in frontier with higher Path-Cost then
        replace that frontier node with child
```

Exemplo: Sibiu → Bucharest





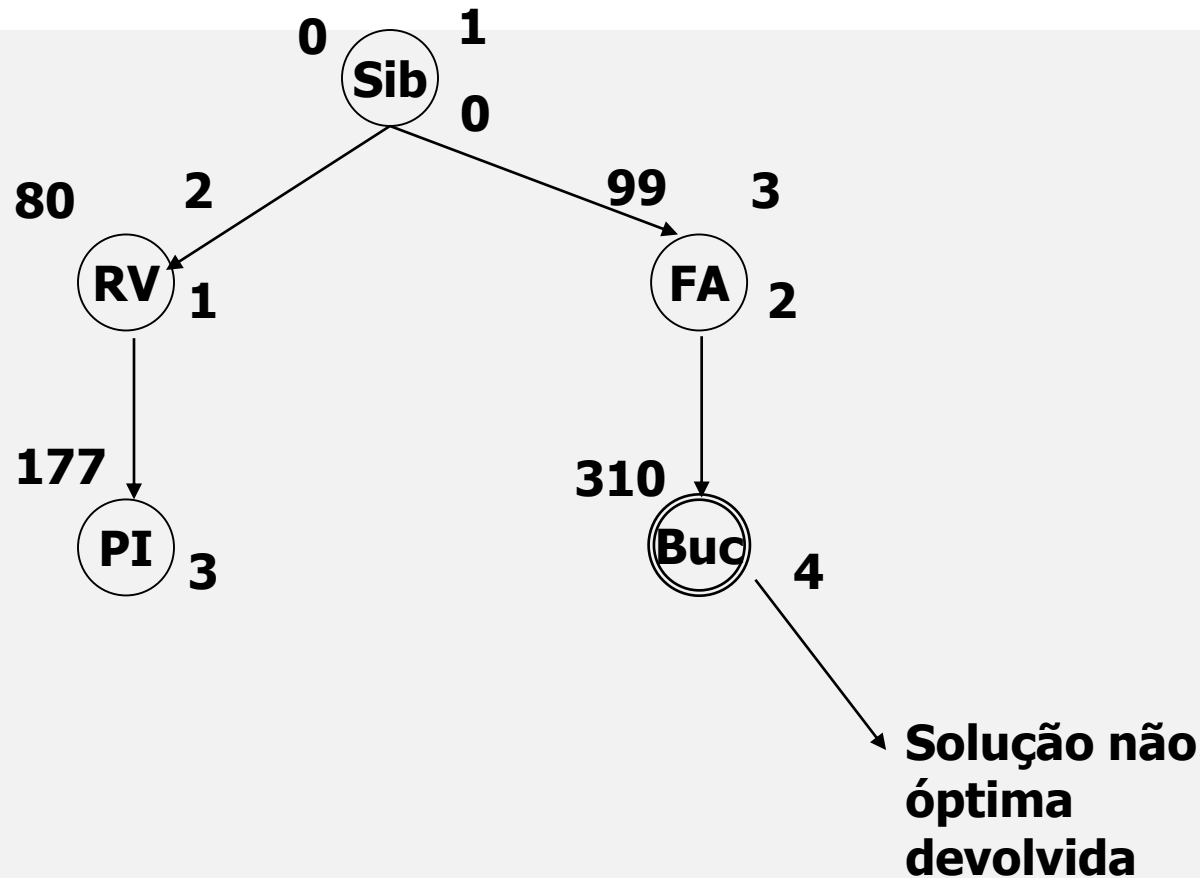
Exemplo: Sibiu → Bucharest



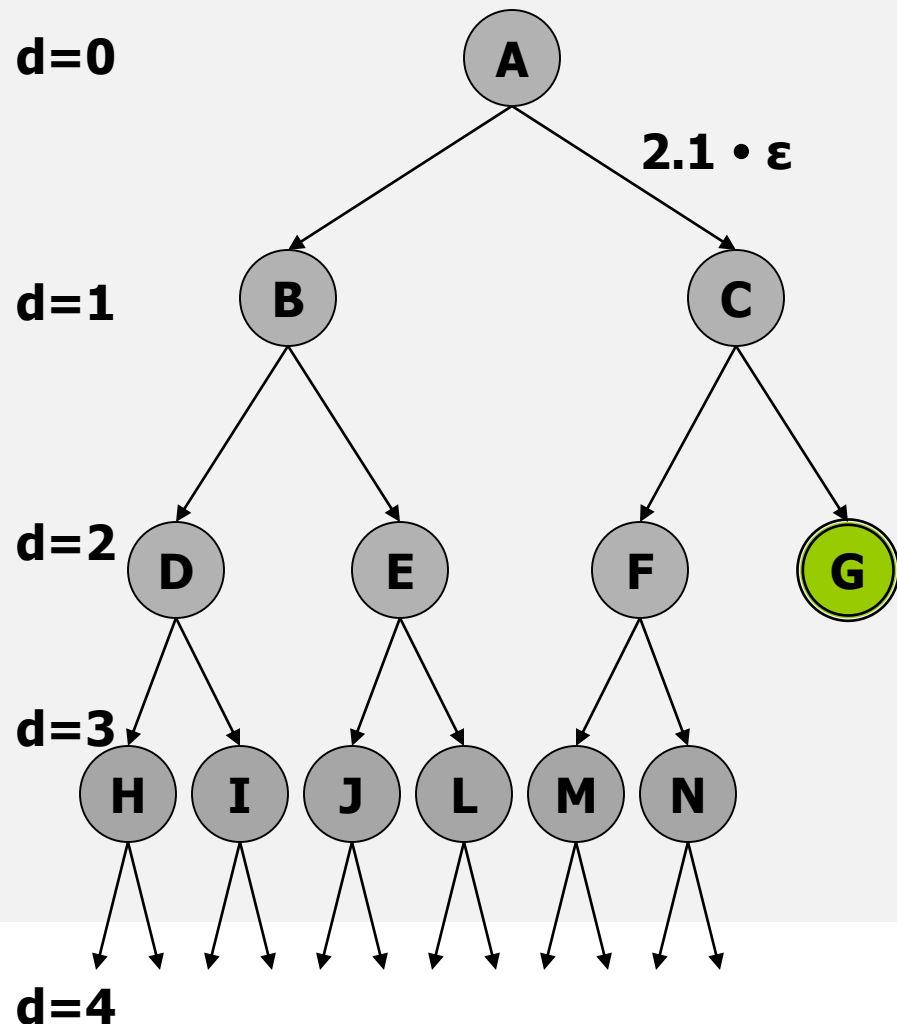
Teste Objectivo PCU

- Porque é que o teste objectivo é feito na expansão e não na geração?
 - Perdemos optimabilidade

PCU com teste geração



Complexidade: exemplo



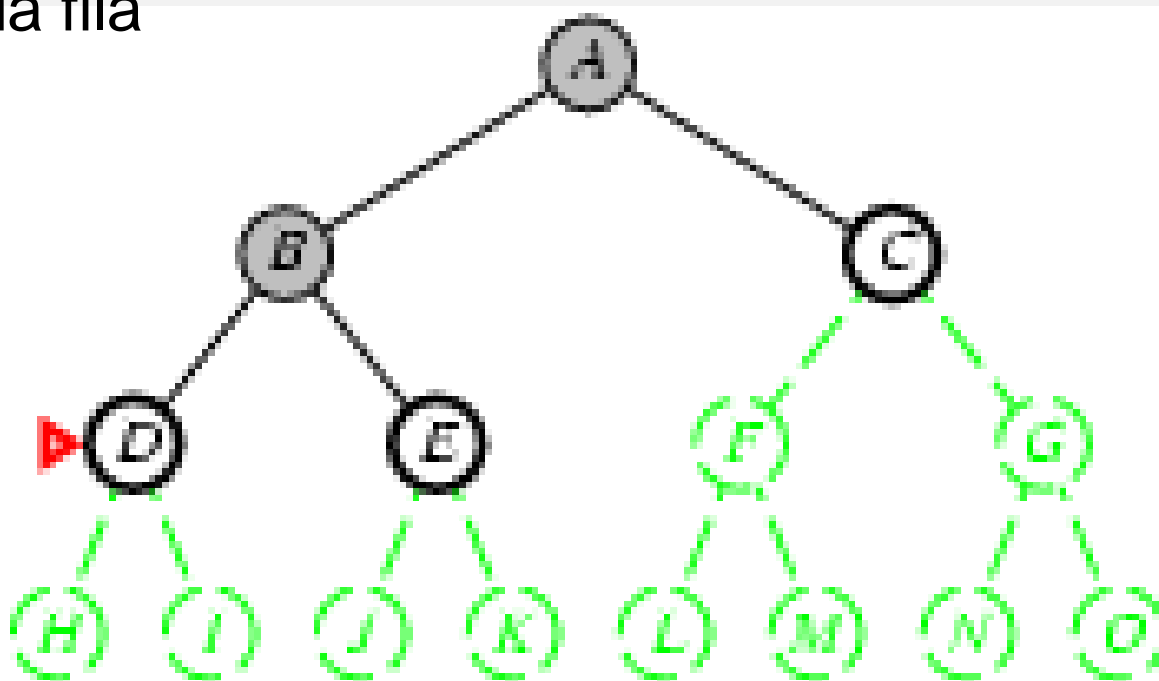
- Todos os ramos com custo ϵ , com excepção do ramo assinalado
- Objectivo G ($C^* = 3.1 \cdot \epsilon$)
- Factor de ramificação $p=2$
- Tempo e Espaço
 - $1 + 2^1 + 2^2 + (2^3 - 2) + (2^4 - 4)$
 - $O(2^4)$, i.e, $O(b^{1 + \lfloor C^*/\epsilon \rfloor})$

Complexidade PCU

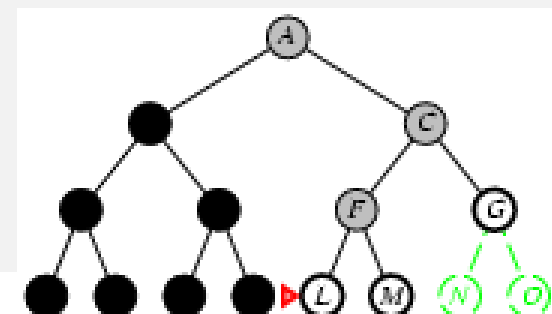
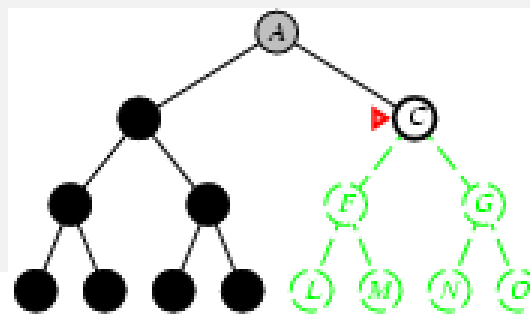
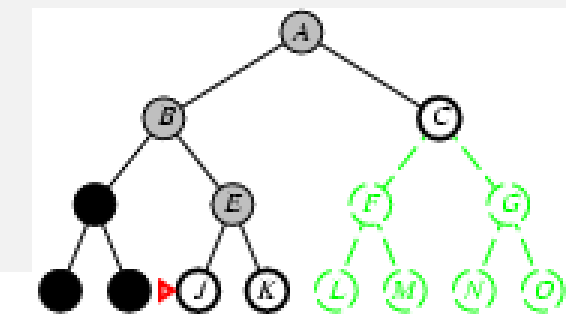
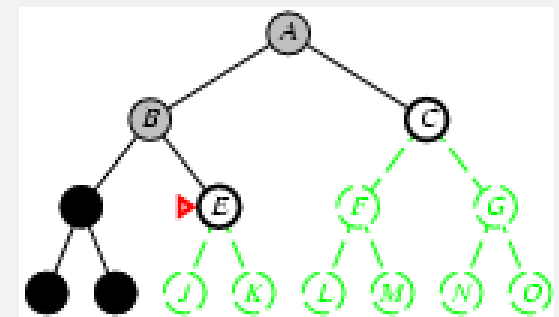
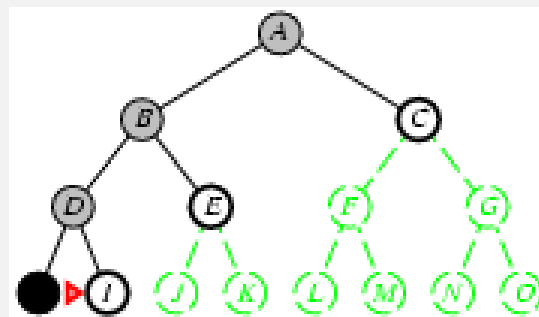
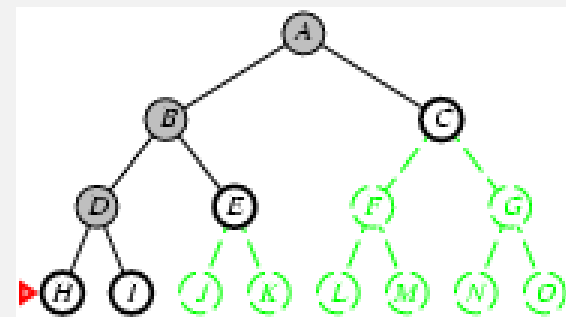
- Completa? Sim, se custo do ramo $\geq \varepsilon$
 - ε é uma constante > 0 , para evitar ciclos em ramos com custo 0
 - Custo do caminho aumenta sempre com a profundidade
- Tempo? # de nós com $g \leq$ custo da solução óptima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ onde C^* é o custo da solução óptima
 - Todos os ramos com o mesmo custo $\rightarrow O(b^{1+\lfloor C^*/\varepsilon \rfloor}) = O(b^{d+1})$
- Espaço? # de nós com $g \leq$ custo da solução óptima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- Óptima? Sim – nós expandidos por ordem crescente de g

P. Profundidade Primeiro

- Expandir nó na fronteira com a maior profundidade
- Implementação:
 - *fronteira* = fila LIFO (pilha), i.e., sucessores colocados no início da fila



P. Profundidade Primeiro



Profundidade Primeiro: propriedades

- Completa? Não: não encontra a solução em espaços de estados com profundidade infinita/com ciclos
 - Modificação para evitar estados repetidos ao longo do caminho → completa em espaços finitos
- Tempo? $O(b^m)$: problemático se máxima profundidade do espaço de estados m é muito maior do que profundidade da solução de menor custo d
- Espaço? $O(b \cdot m)$ - espaço linear (só um caminho)
- Óptima? Não

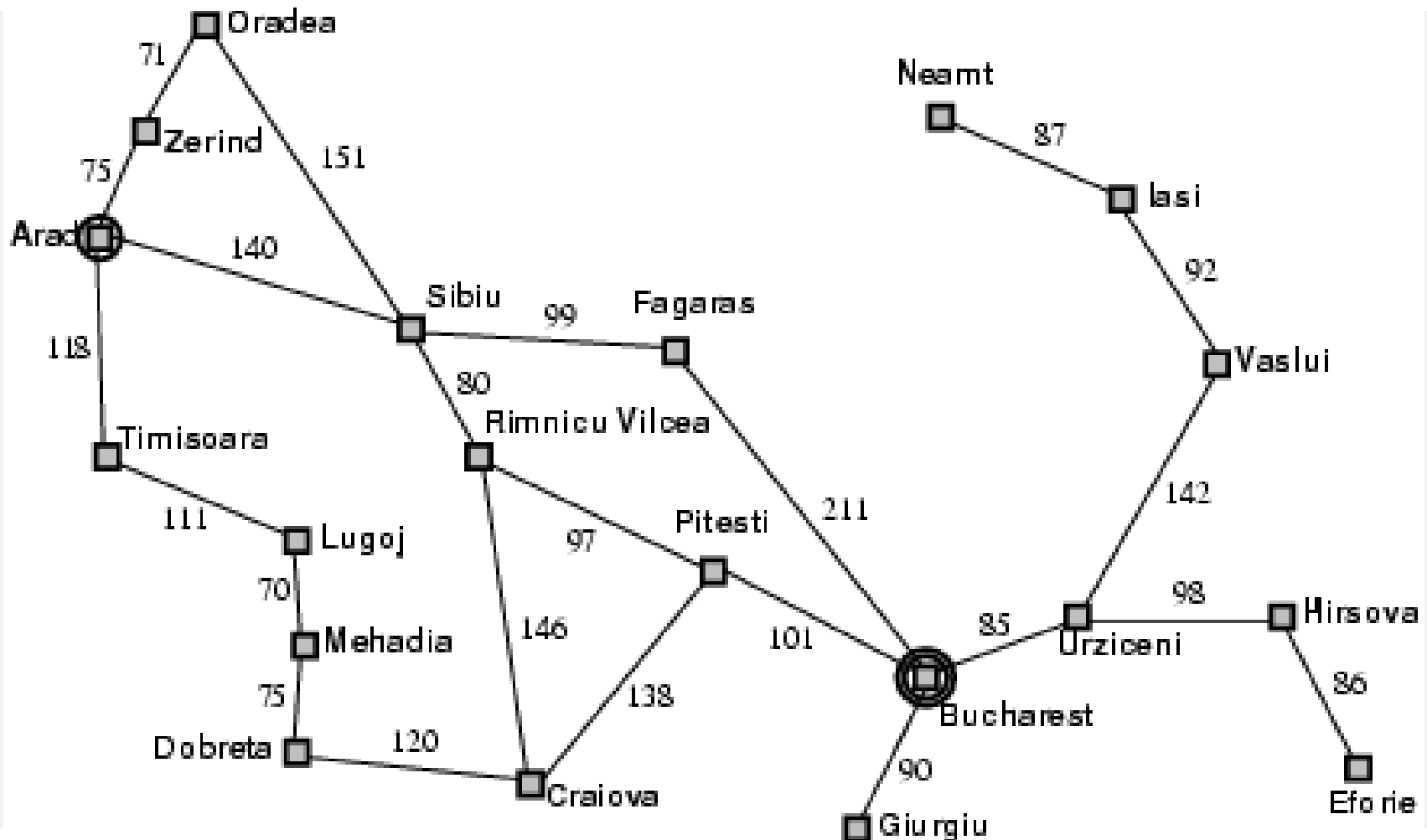
P. Profundidade Primeiro

- **Implementação:** habitualmente recursiva
- Nós deixam de ser guardados em memória quando todos os seus sucessores são gerados
- Variante: procura por retrocesso
 - Usa ainda menos memória: $O(m)$ vs. $O(b*m)$
 - Só é gerado um sucessor de cada vez

P. Profundidade Limitada

- Profundidade primeiro com limite de profundidade l , i.e., nós com profundidade l não têm sucessores
- Resolve problema da profundidade infinita
 - Limite pode ser determinado em função do tipo de problema
 - **Diâmetro** do espaço de estados define máxima profundidade da solução
- Se $d > l$ não é encontrada solução
- Complexidade temporal $O(b^l)$
- Complexidade espacial $O(b^l)$

Exemplo: diâmetro? 9



Distância máxima de 9 troços entre quaisquer duas cidades

Implementação Recursiva

function DEPTH-LIMITED-SEARCH (*problem, limit*) **returns** a solution, or failure/cutoff

return RECURSIVE-DLS(Make-Node(*problem.Initial-State*), *problem, limit*)

function RECURSIVE-DLS (*node, problem, limit*) **returns** a solution, or failure/cutoff

if *problem.Goal-Test*(*node.State*) **then return** Solution(*node*)

else if *limit* = 0 **then return** cutoff

cutoff_occurred? \leftarrow false

for each *action* **in** *problem.Actions*(*node.State*) **do**

child \leftarrow Child-Node(*problem, node, action*)

result \leftarrow RECURSIVE-DLS(*child, problem, limit - 1*)

if *result* = cutoff **then** *cutoff_occurred?* \leftarrow true

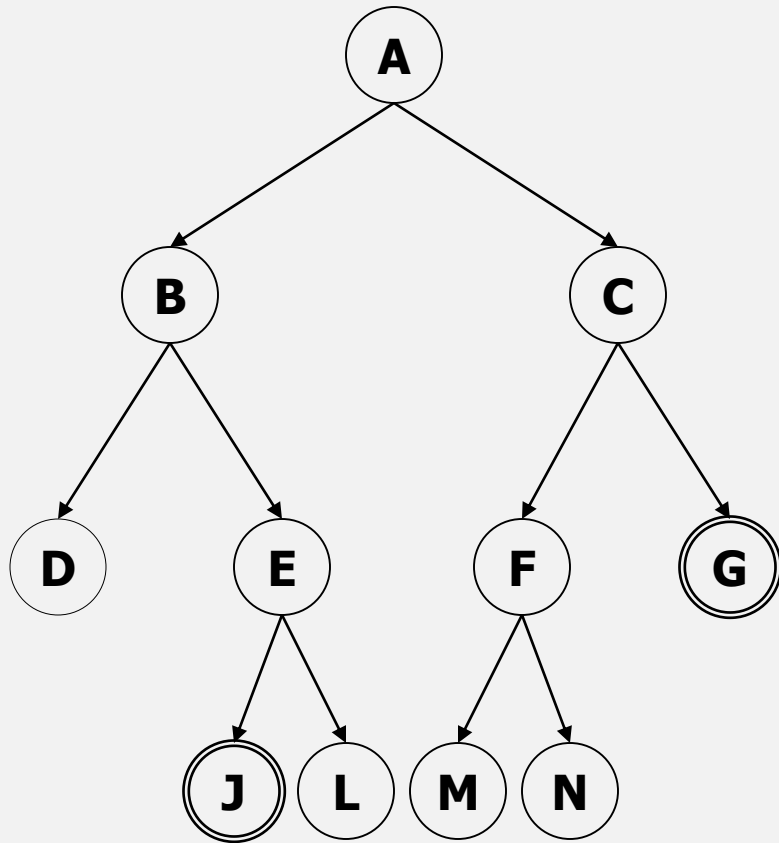
else if *result* \neq failure **then return** *result*

if *cutoff_occurred?* **then return** *cutoff* **else return** failure

Implementação Recursiva

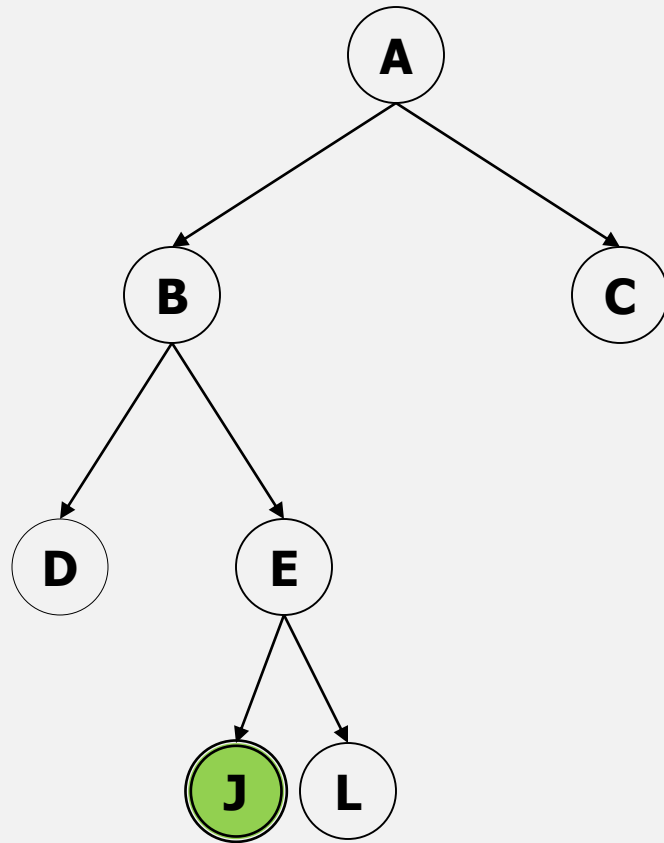
- Algoritmo *Deep-Limited-Search* tem 3 outputs possíveis:
 - Solução: se encontra solução
 - Corta: se não encontra solução mas não chegou a expandir toda a árvore devido ao limite de profundidade
 - Não há solução: se não encontrou solução e expandiu toda a árvore

Profundidade Limitada: exemplo



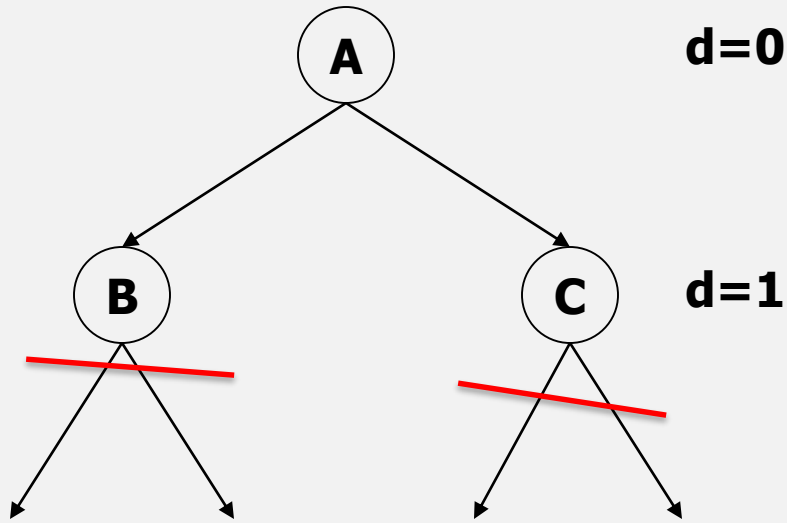
- Solução encontrada?
 - Profundidade Primeiro
 - Profundidade Limitada
 - $l = 1$
 - $l = 2$

Profundidade Limitada: exemplo



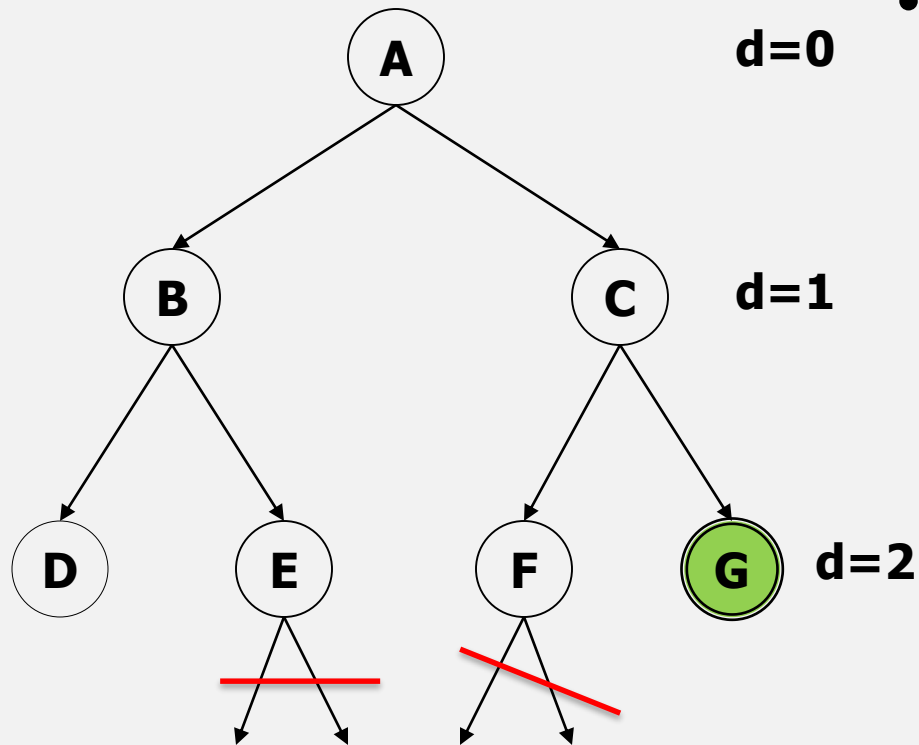
- Solução encontrada?
 - Profundidade Primeiro

Profundidade Limitada: exemplo



- Solução encontrada?
 - Profundidade Limitada
 - $l=1$

Profundidade Limitada: exemplo



- Solução encontrada?
 - Profundidade Limitada
 - $l=2$

P. Profundidade Iterativa

- Profundidade limitada com limite incremental: $l=0$, $l=1$, $l=2$, $l=3$, ..., $l=d$
- Combina vantagens da largura primeiro e da profundidade primeiro

```
function ITERATIVE-DEEPENING-SEARCH (prob)  
  returns a solution, or failure  
  for depth  $\leftarrow 0$  to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITE-SEARCH(prob,limit)  
    if result  $\neq$  cutoff then return result
```

Profundidade Iterativa

$l=0$

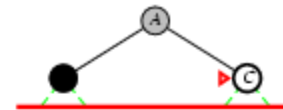
Limit = 0



Profundidade Iterativa

$l=1$

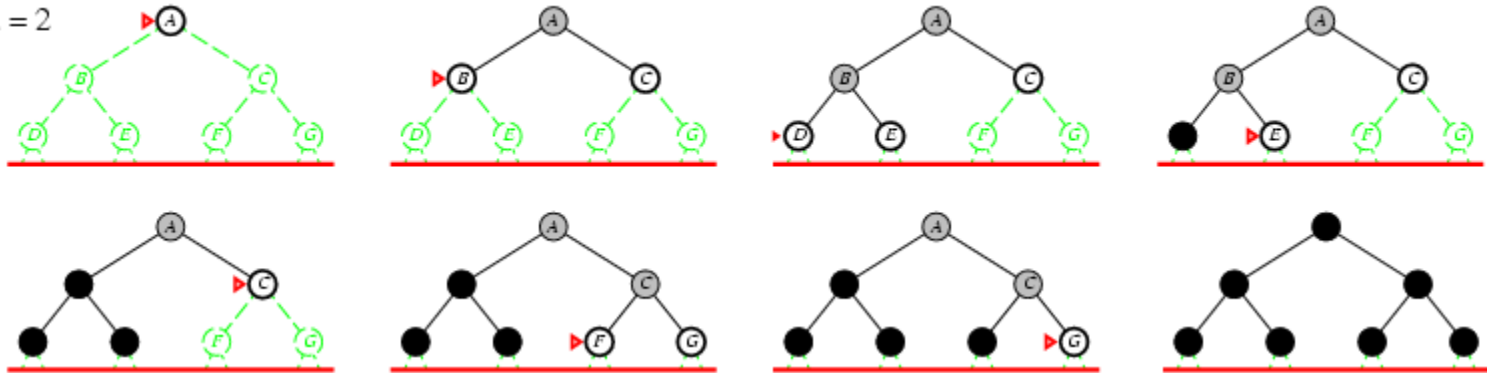
Limit = 1



Profundidade Iterativa

$l=2$

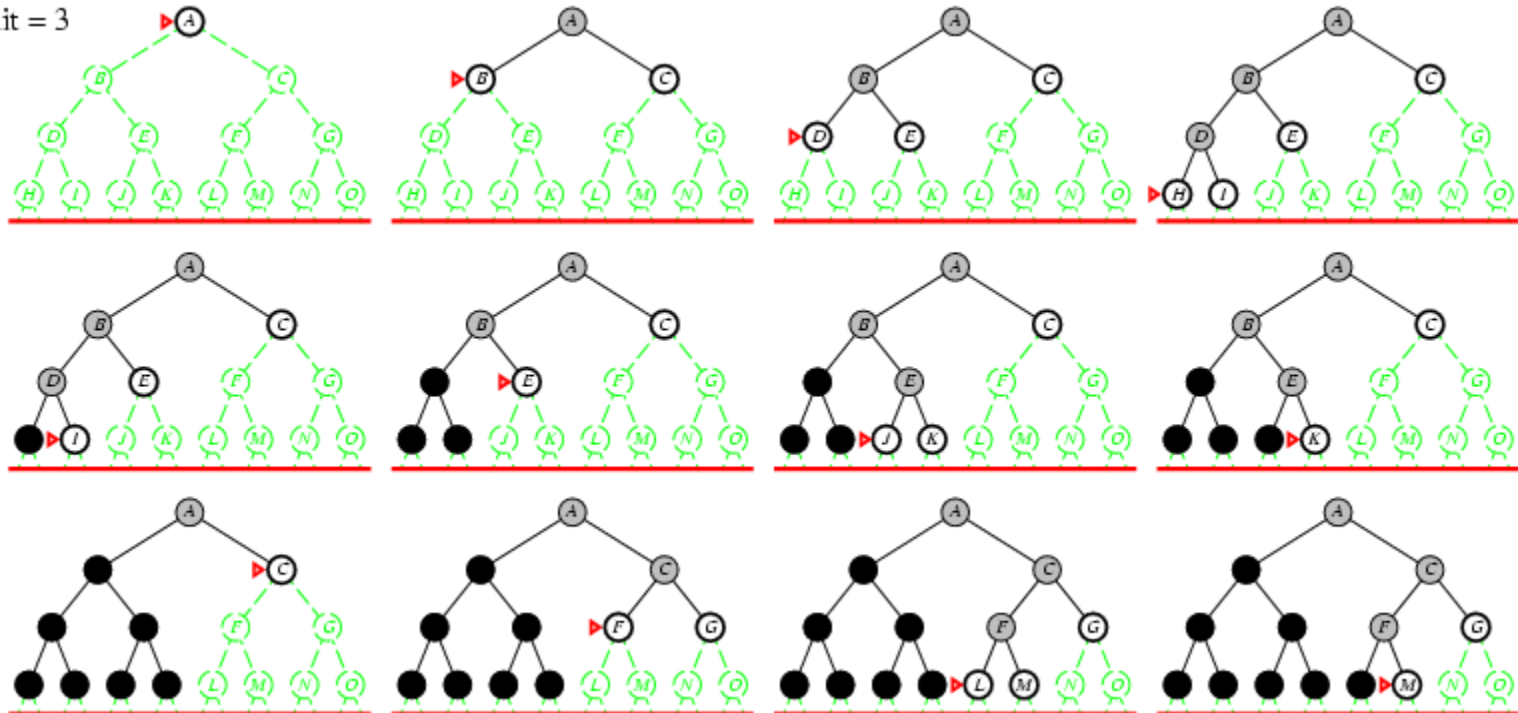
Limit = 2



Profundidade Iterativa

$l=3$

Limit = 3



Profundidade Iterativa

- Número de nós gerados na procura em profundidade limitada com profundidade d e factor de ramificação b :

$$N_{PPL} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

- Número de nós gerados na procura em profundidade iterativa com profundidade d e factor de ramificação b :

$$N_{PPI} = (d+1)b^0 + db^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- Para $b = 10$, $d = 5$,

- $N_{PPL} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$

- $N_{PPI} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

- Esforço adicional = $(123,450 - 111,110)/111,110 = 11\%$

Profundidade Iterativa:

propriedades

- Completa? Sim
- Tempo? $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Espaço? $O(b^*d)$
- Óptima? Sim, se custo de cada ramo = 1

Procura Bi-Direccional

- Executar duas procuras em largura em simultâneo
 - Uma a partir do estado inicial (*forward*, para a frente)
 - Outra a partir do estado final (*backward*, para trás)
- Procura termina quando as duas procuras se encontram (têm um estado em comum)
- Motivação: $b^{d/2} + b^{d/2} \ll b^d$
- Necessidade de calcular eficientemente os **predecessores** de um nó
- Problemática quando estados objectivos são descritos implicitamente (por ex^o, checkmate)

Procura Bi-Direccional:

propriedades

- Completa? Sim, se p é finito e se executa procura em largura primeiro em ambas as direcções
- Tempo? $O(b^{d/2})$
- Espaço? $O(b^{d/2})$
- Óptima? Sim, se custo de cada ramo = 1 e se executa procura em largura primeiro em ambas as direcções

Resumo dos algoritmos

| | Largura Primeiro | Custo Uniforme | Profund. Primeiro | Profund. Limitada | Profund. Iterativa | Bi-direc- cional |
|-----------|---------------------|-----------------------------------------|----------------------|----------------------|-----------------------|---------------------|
| Completa? | Sim ^a | Sim ^{a,b} | Não | Não | Sim ^a | Sim ^{a,d} |
| Tempo | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Espaço | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b^l)$ | $O(bd)$ | $O(b^{d/2})$ |
| Óptima? | Sim ^c | Sim | Não | Não | Sim ^c | Sim ^{c,d} |

- **a completa se b é finito**
- **b completa se custo de cada ramo > 0**
- **c óptima se todos os ramos têm o mesmo custo**
- **d se ambas as direcções executam procura em largura primeiro**