Resolução de Problemas Com Procura

Capítulo 3

Sumário

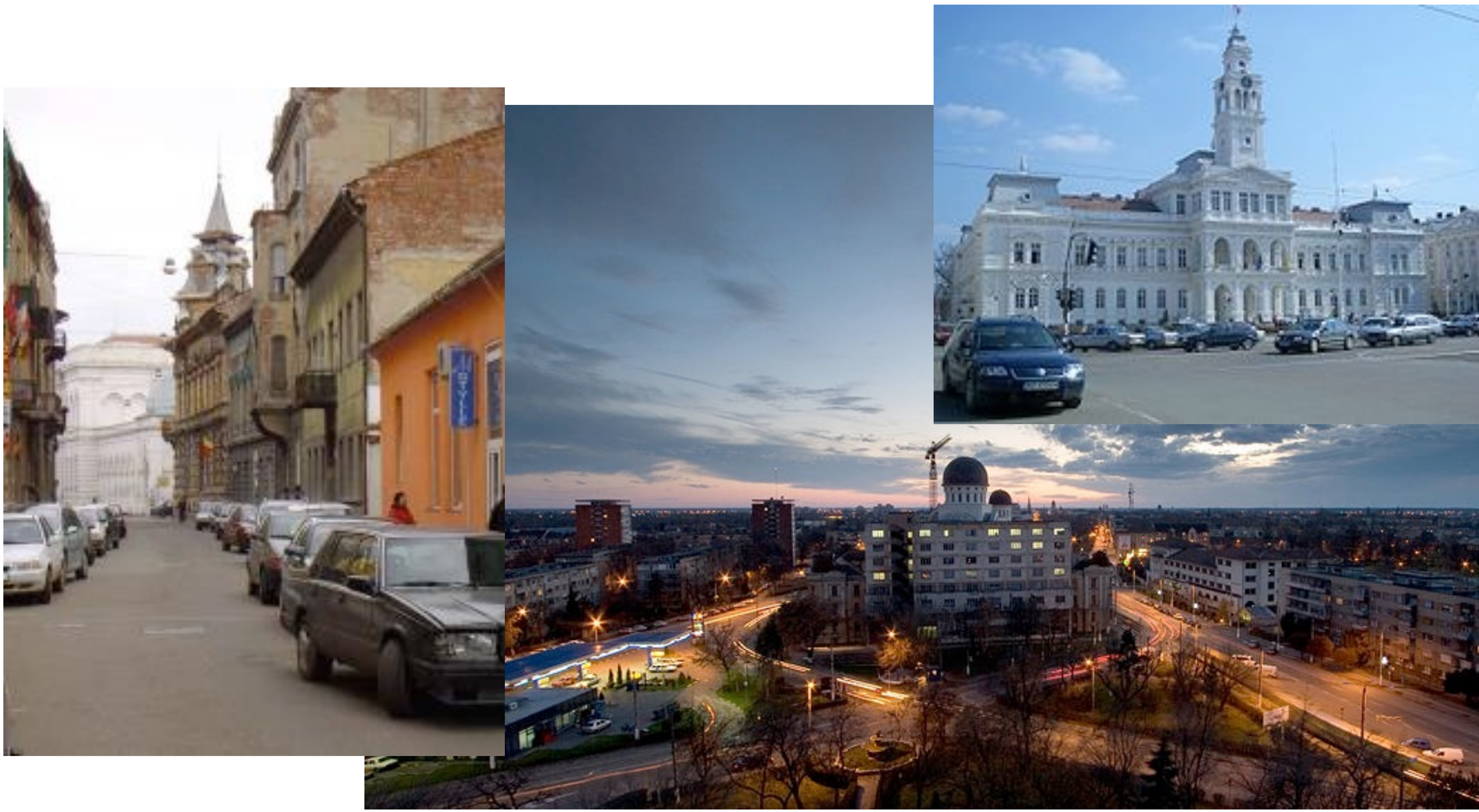
- Agentes que resolvem problemas
- Tipos de problemas
- Formulação de problemas
- Exemplos de problemas
- Algoritmos de procura básicos
- Eliminação de estados repetidos
- Procura com informação parcial

Agentes que resolvem problemas

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

Roménia: Arad -> Bucareste

- Férias na Roménia; atualmente em Arad



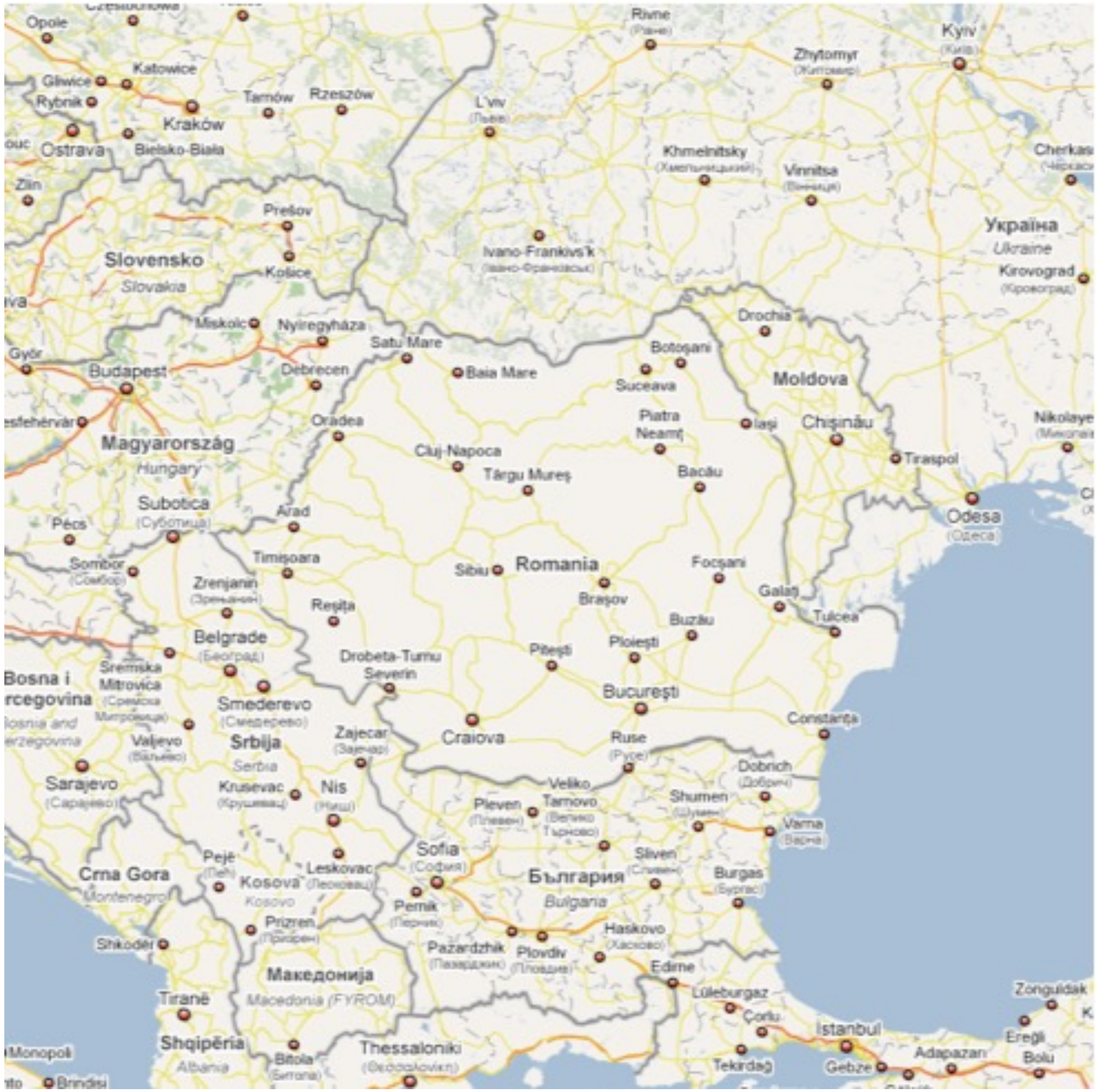
Formulação objetivo

- Voo de volta para Lisboa parte amanhã de Bucareste
- Formulação objetivo
  - Estar em Bucareste
  - Objetivos ajudam a simplificar o processo de decisão
    - Limitam as ações a considerar

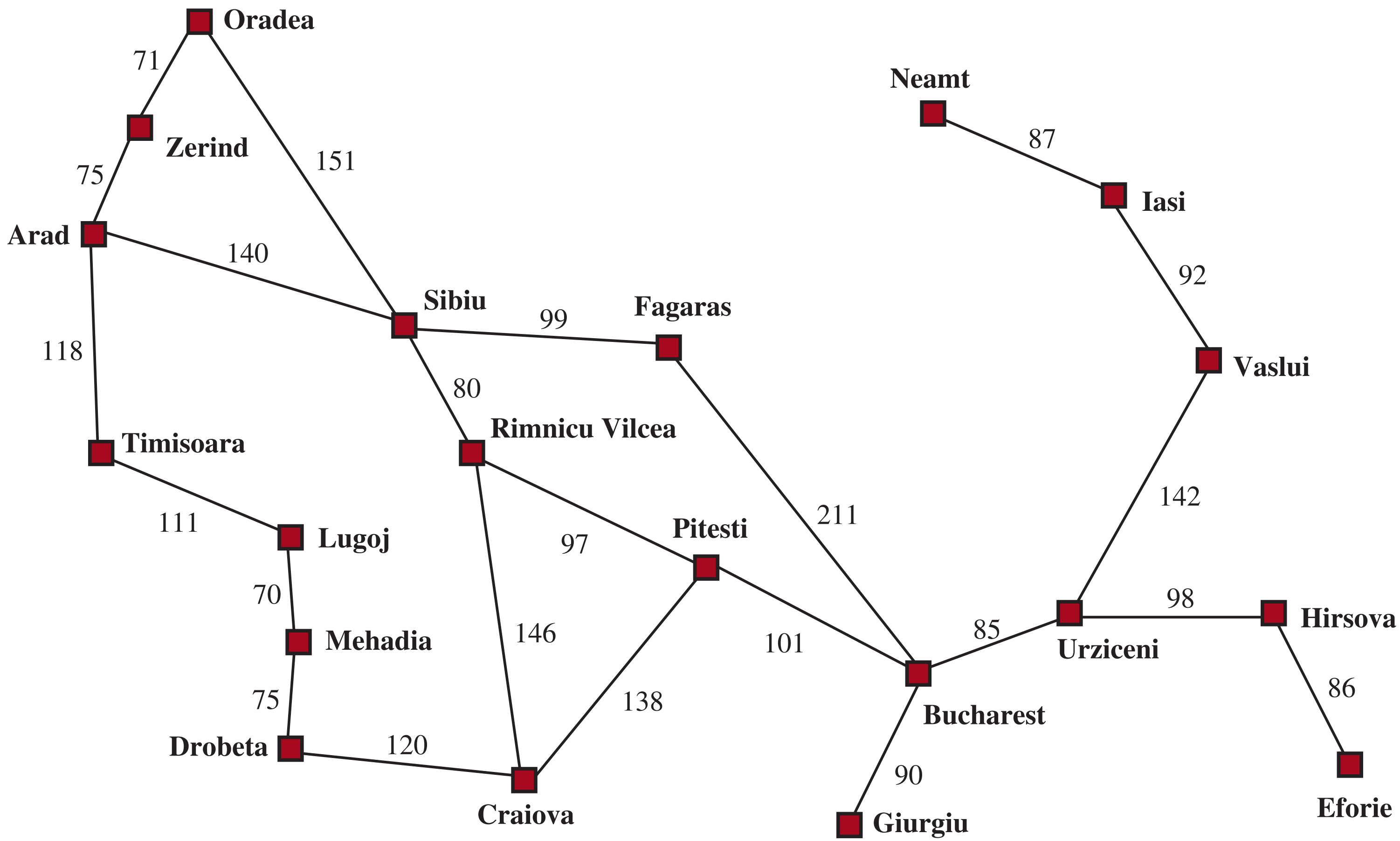
Formulação Problema

- *Férias na Roménia; atualmente em Arad*
- *Voo parte amanhã de Bucareste*
- *Formulação do objetivo:*
  - *Estar em Bucareste*
- Formulação do problema:
  - Estados: várias cidades
  - ações: conduzir entre cidades
- *Encontrar solução:*
  - *Sequência de cidades*
  - *E.g., Arad, Sibiu, Fagaras, Bucareste*

Exemplo: Roménia



Exemplo: Roménia



Exemplo: Roménia

- Inexistência de mapa
  - Escolhas aleatórias…
- Existência de mapa
  - Possibilidade de considerar sequência de estados
  - Uma vez encontrada uma solução, basta executar uma sequência de ações para atingir o objetivo
- Procura = encontrar uma solução
- Execução = sequência de ações que permitem alcançar o objetivo
- Agente = Formular + Procurar + Executar

Procura

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

Execução

- Execução
  - Executar a sequência de ações que permitem alcançar o objetivo
  - Pela ordem especificada na sequência

- Agente que resolve problemas
  - Formular objetivo
  - Formular Problema
  - Procurar
  - Executar

Tipo de ambiente/problema

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

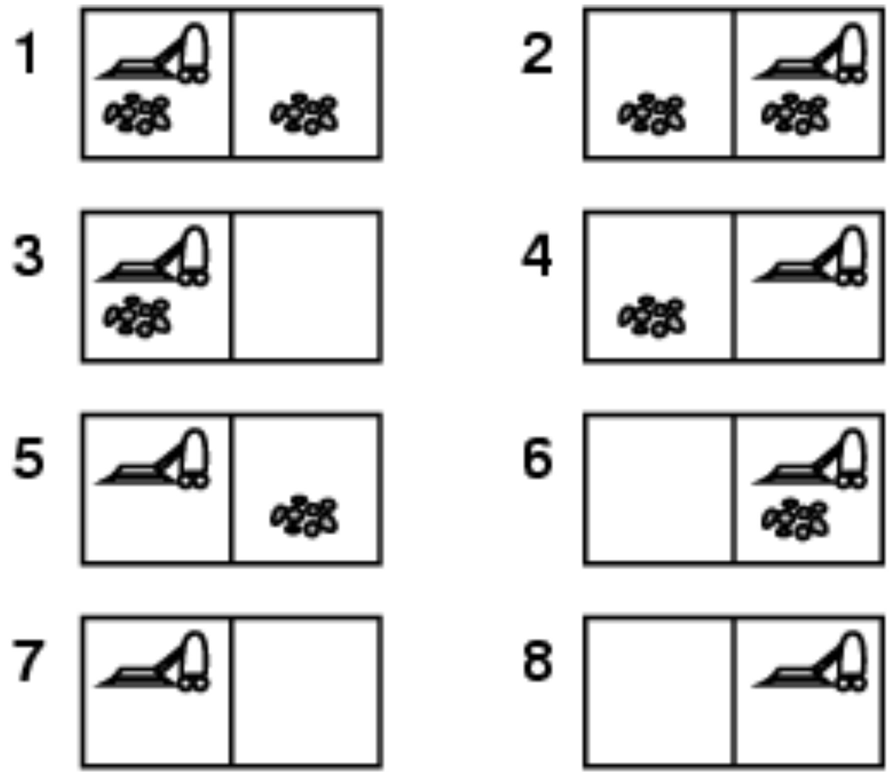
Formulação do Problema (1/2)

- Problema definido por 5 componentes
  - Estado inicial
    - onde se encontra o agente
  - Ações(e)
    - função que dado um estado ***e***, retorna o conjunto de ações que podem ser executadas em ***e***
  - Modelo de transição/Resultado(a,e)
    - função que dada uma ação ***a*** e um estado ***e***, retorna o estado ***e'*** que resulta de executar ***a*** em ***e***
  - Teste objetivo
    - função que dado um estado ***e***, retorna *Verdade* se o estado ***e*** for um estado objetivo, e *Falso* caso contrário
  - Custo caminho
    - função que atribui um custo numérico a cada caminho (sequência de ações a partir do estado inicial)
    - Escolhida em função do desempenho pretendido para o agente (rapidez → km)

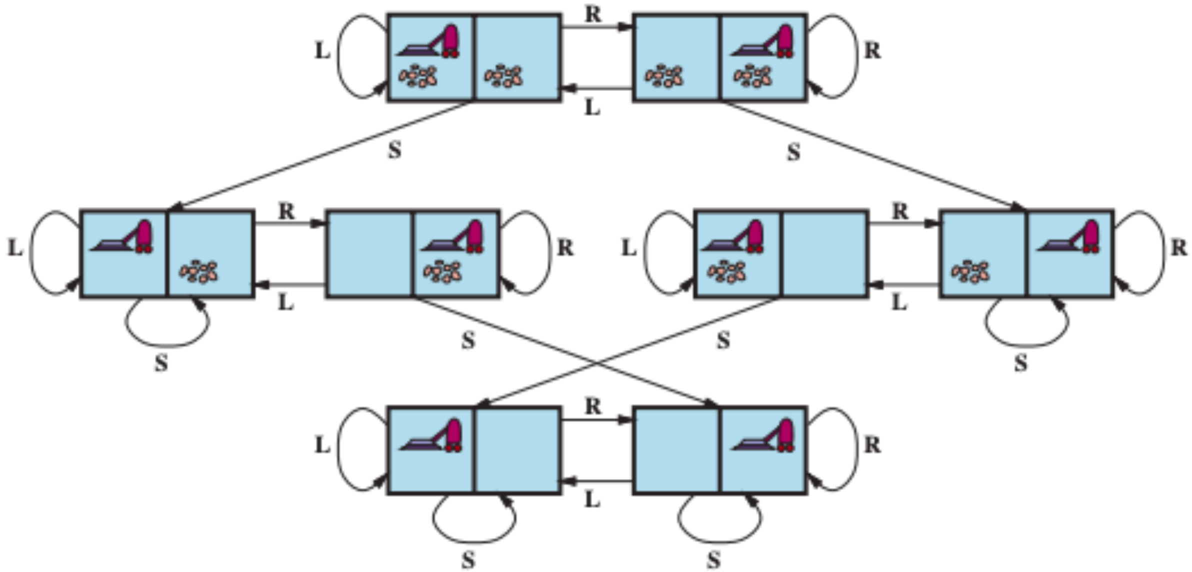
Formulação do Problema (2/2)

- Problema ir de Arad – Bucareste
  - Estado inicial
    - Em(Arad)
  - Ações
    - Caminhos entre cidades
    - E.g. considerando o estado ***e*** = Em(Arad)
      - acoes(e) = {Ir(Sibiu),Ir(Timisoara),Ir(Zerind)}
  - Resultado
    - Resultado(Em(Arad),Ir(Zerind)) = Em(Zerind)
  - Teste objetivo
    - objetivo(e) = (e == Em(Bucareste))
  - Custo caminho
    - Distância percorrida (em Km) desde o estado inicial

Mundo do aspirador: estados

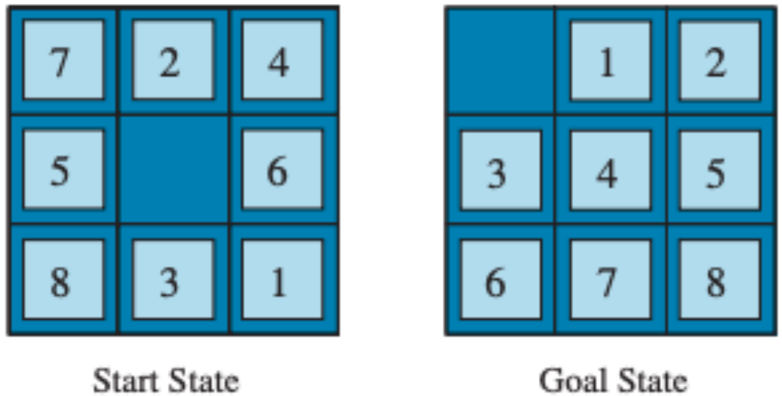


Espaço de Estados: grafo



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

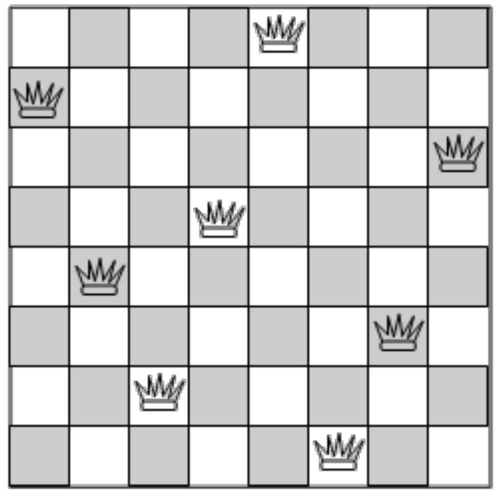
Exemplo: 8-puzzle



- estados? localização das peças
- estado inicial? ver imagem acima
- ações? mover espaço esq, dir, cima, baixo
- resultado? troca de peças resultante do movimento
- teste objetivo? = estado objetivo (dado)
- custo de caminho? 1 por movimento no caminho

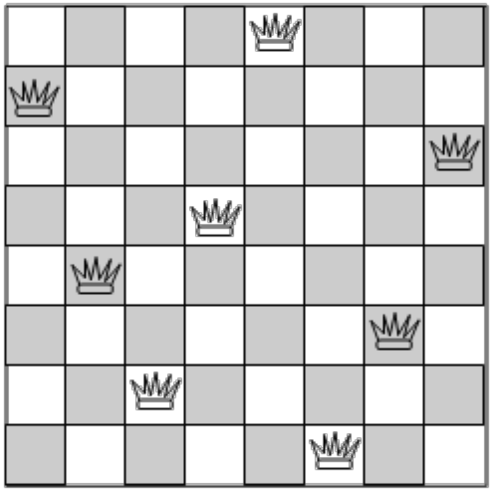
[Nota: solução ótima para a família *n*-Puzzle é NP-difícil]

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

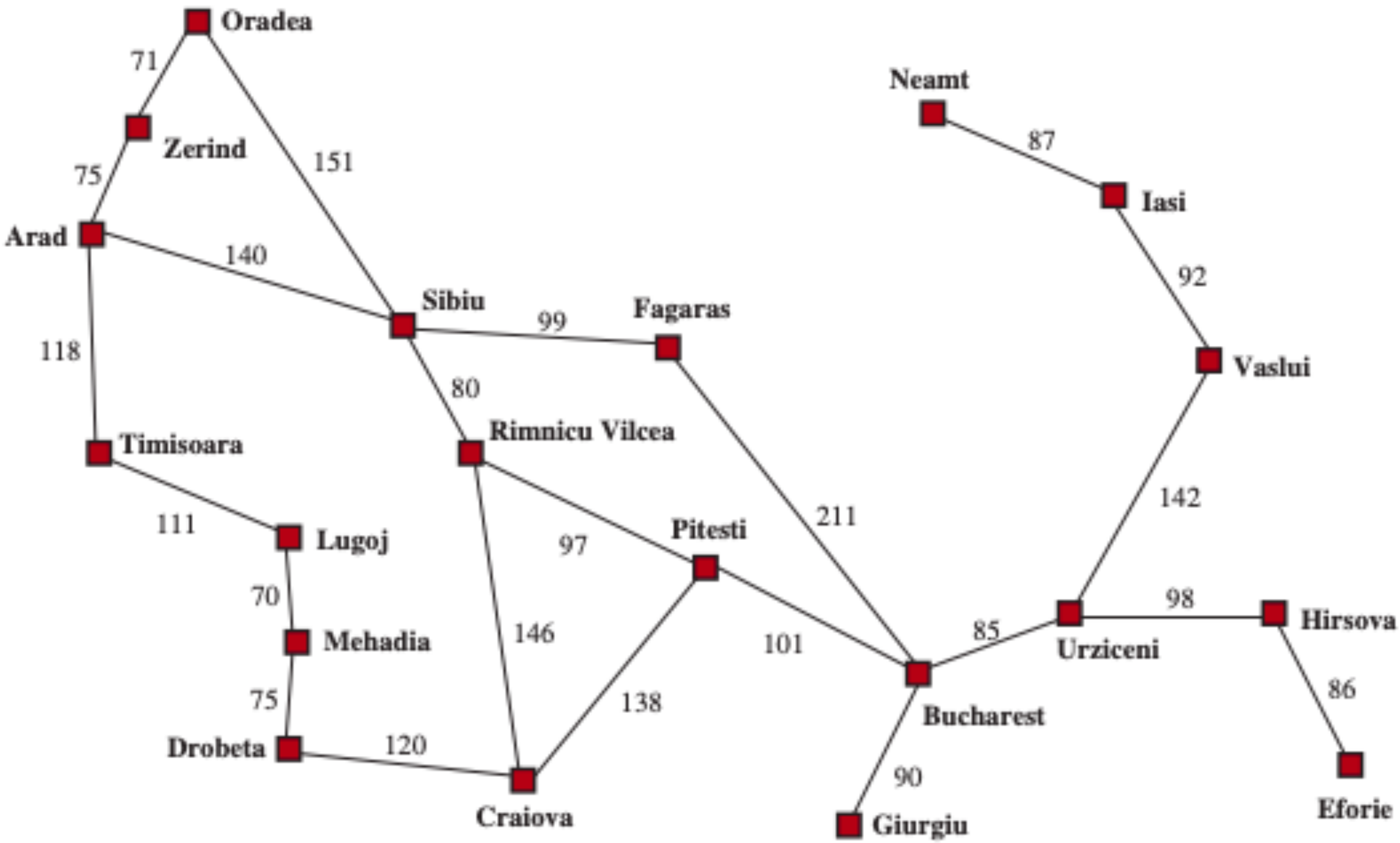


- estados? qualquer tabuleiro de 8x8 com *n* rainhas ($0 \leqslant n \leqslant 8$), com 1 rainha por coluna, e colocadas nas n colunas mais à esquerda
- estado inicial? tabuleiro sem rainhas
- ações? adicionar uma rainha na coluna mais à esquerda não preenchida (de modo a que não seja atacada por nenhuma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

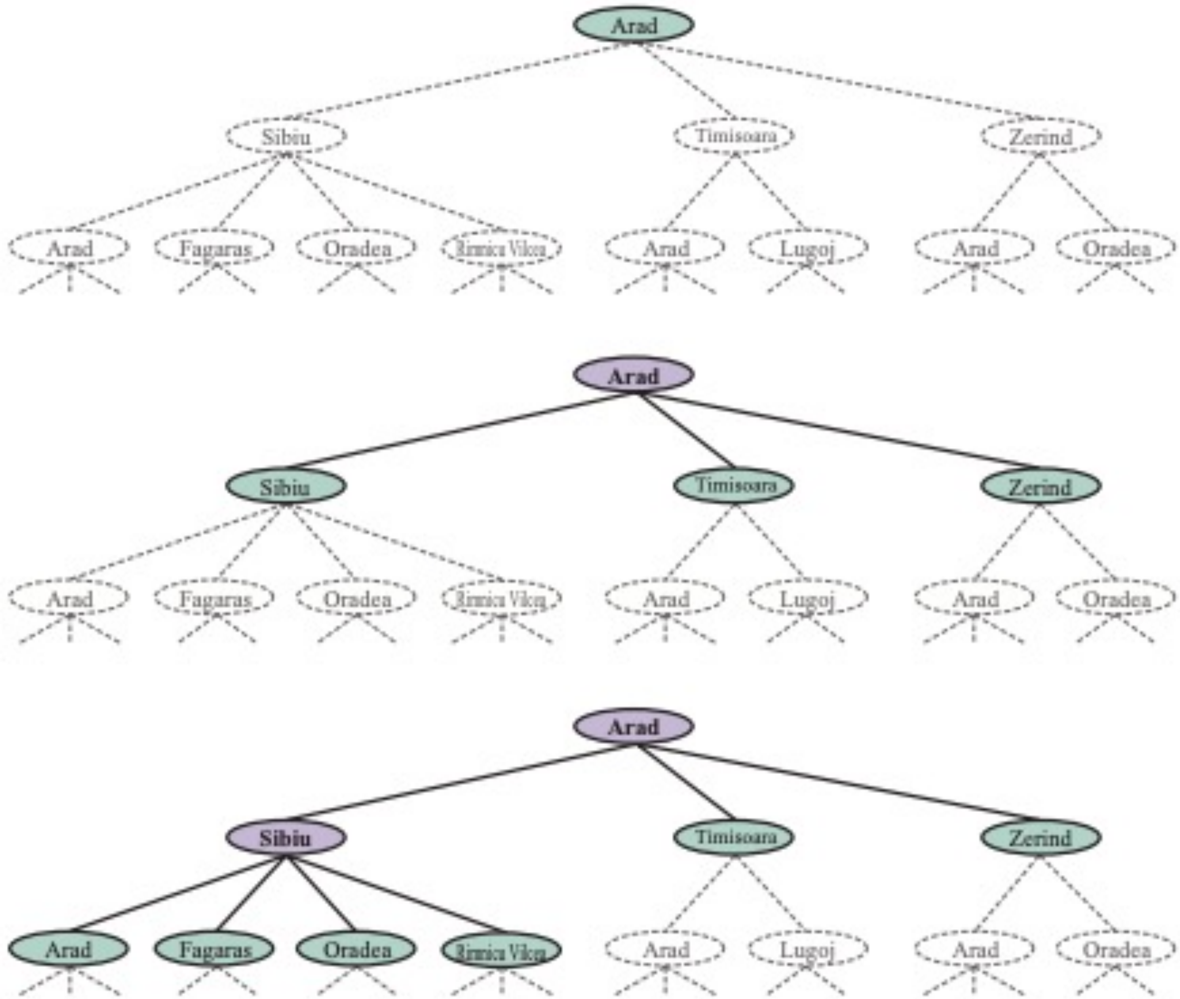
Árvore de Procura

- Raiz: estado inicial
- Ramos: ações
- Nós: estados no espaço de estados
- Algoritmo:
  - Expansão do estado atual a partir da geração dos seus sucessores
  - Fronteira (ou lista de abertos *vs. fechados*) contém todos os nós folha que ainda não foram expandidos
  - Árvore de procura vs. espaço de estados
    - O mesmo estado pode estar em nós diferentes (loop?)
    - Caminho bem definido

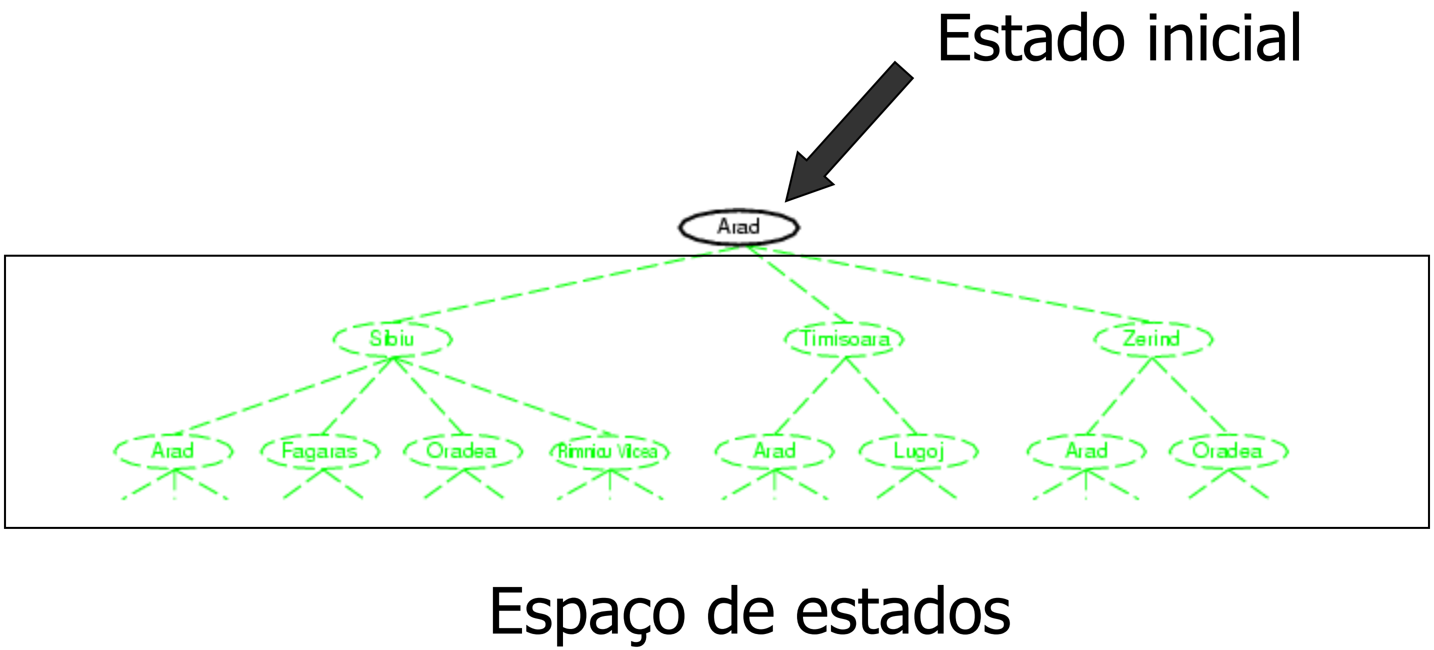
Exemplo: Roménia



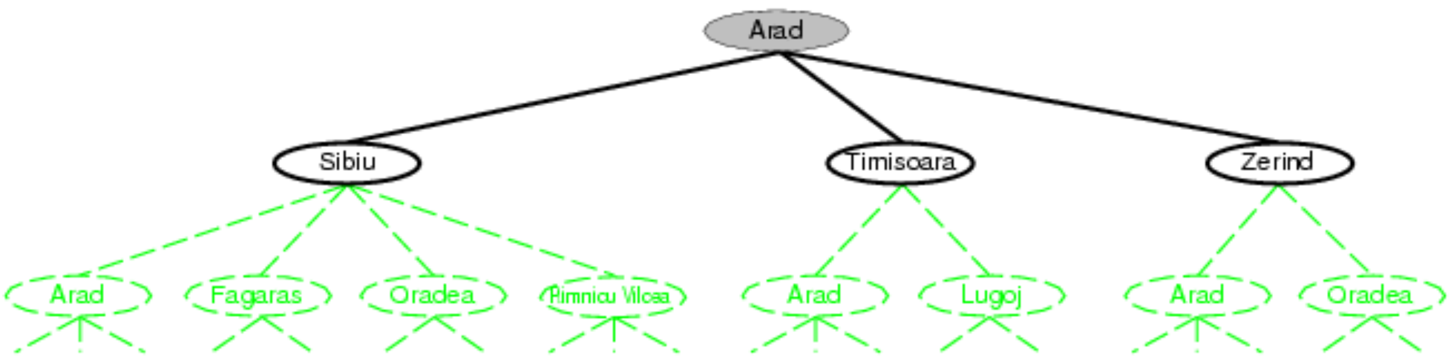
Árvore de Procura: exemplo



Árvore de Procura: exemplo



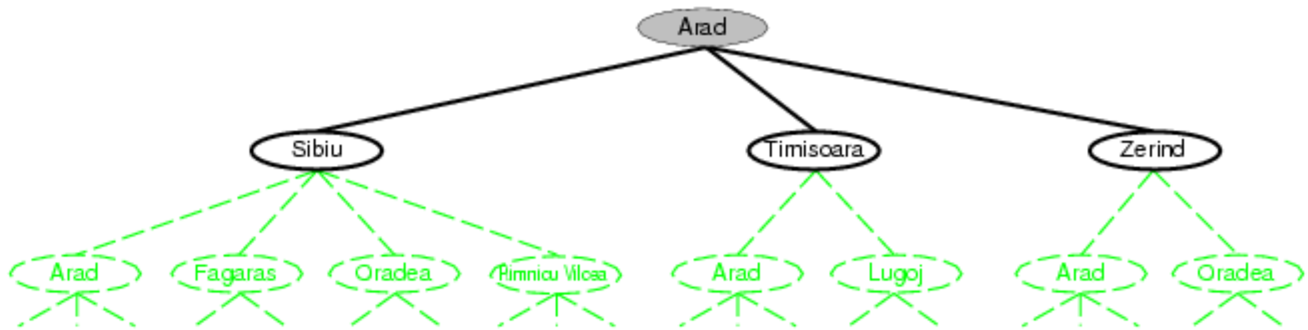
Árvore de Procura: exemplo



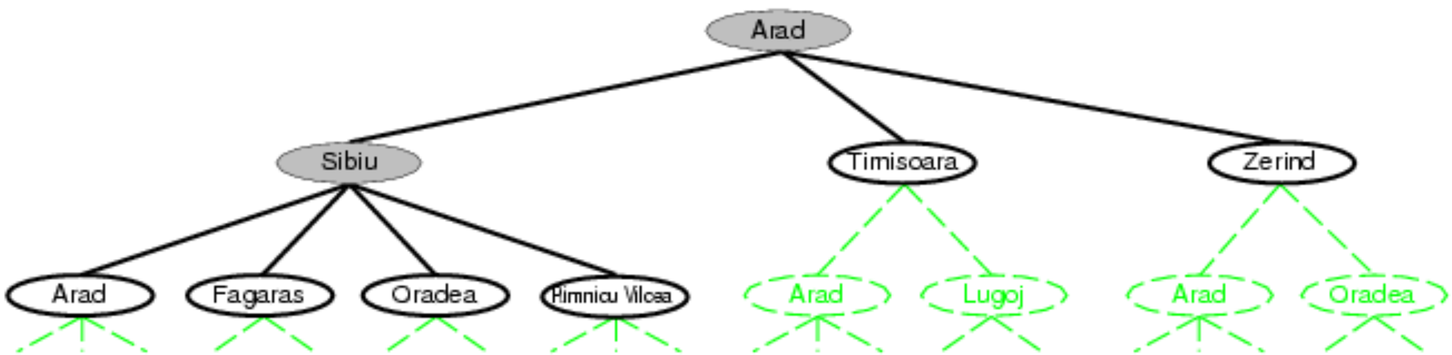
- Inicialmente existe um único nó folha que pode ser expandido: Arad
- Nós resultantes da expansão do nó Arad: Sibiu, Timisoara, Zerind

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



Árvore de Procura: exemplo

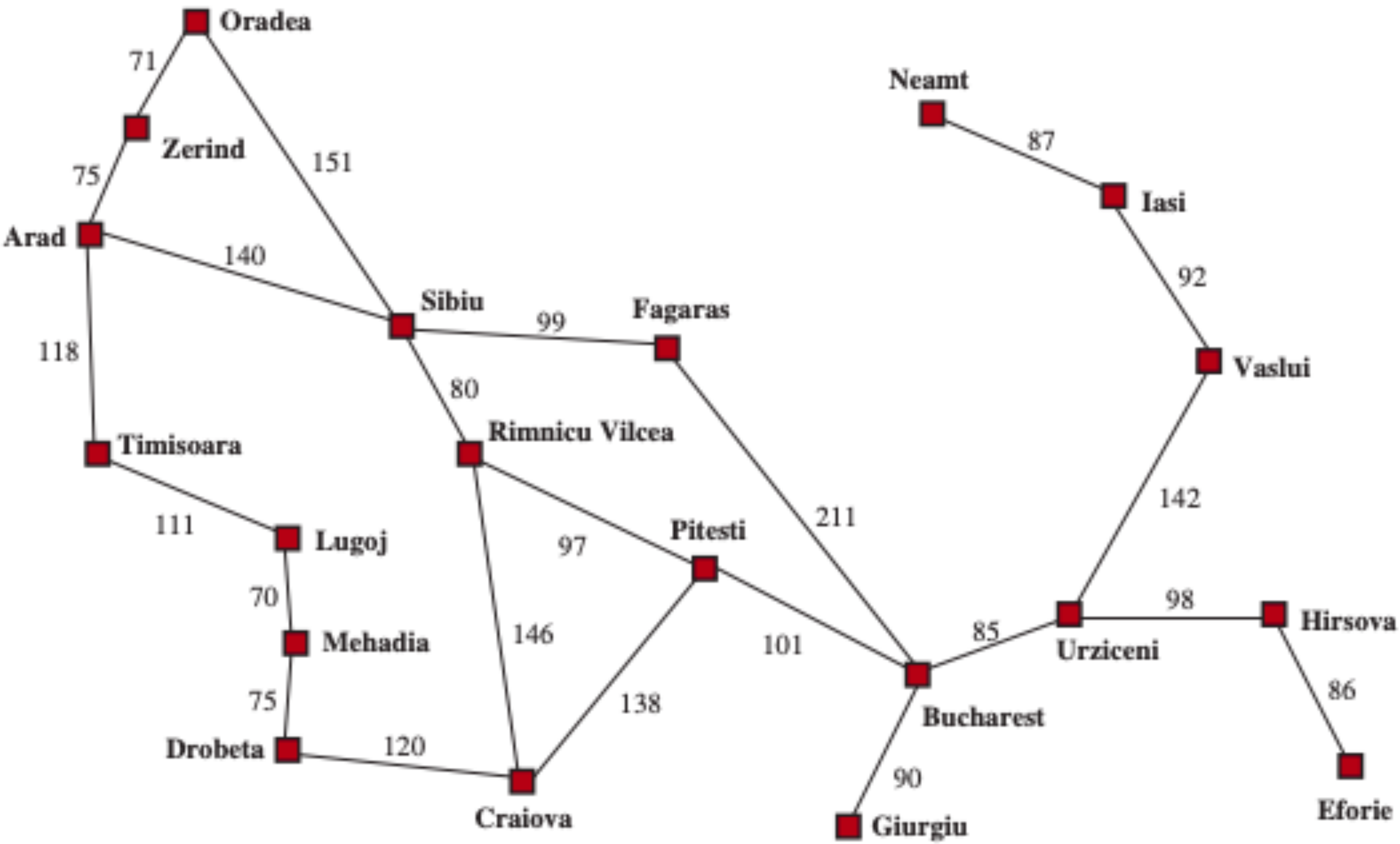


- Existem três nós folha que podem ser expandidos: *Sibiu, Timisoara, Zerind*
- Nó *Sibiu* é escolhido pela estratégia de procura como o próximo nó a ser expandido
- Nós resultantes da expansão do nó *Sibiu* são adicionados ao conjunto de nós folha / fronteira

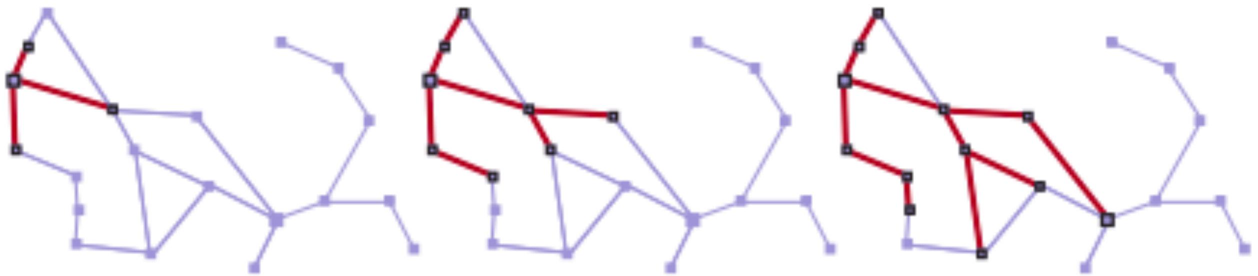
Grafo de Procura

- Motivação: nós com estados repetidos
- Mantém conjunto de nós expandidos (ou lista de fechados)
- Nós gerados que já foram expandidos são descartados!

Exemplo: Roménia

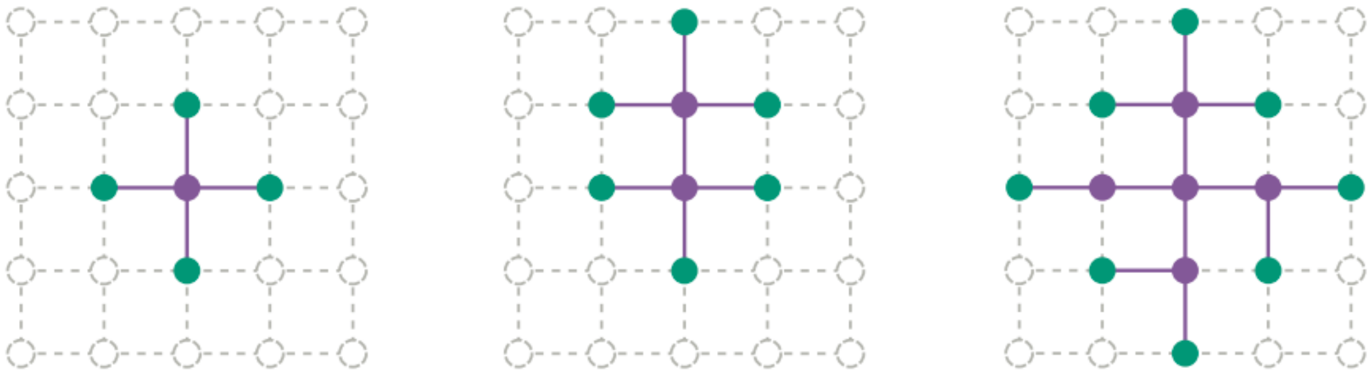


Grafo de Procura: exemplo



- Na 3ª iteração a cidade Oradea é considerada *dead-end*
  - Porque os seus dois sucessores já tinham sido explorados através de outros caminhos

Grafo de Procura: exemplo



(a) (b) (c)

- (a) raiz expandida
- (b) um nó folha expandido
- (c) restantes sucessores da raiz expandidos
  - Ordem ponteiros relógio
- Propriedade de separação
  - Separação clara entre fronteira (nós verdes) vs nós explorados (lilás)

Procura Melhor Primeiro

- Abordagem genérica
- Questão: “Como escolher o nó da fronteira que deve ser expandido?”
- Resposta: escolher o nó com o menor valor da função de avaliação

Procura Melhor Primeiro

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Estratégias de Procura

- As estratégias são avaliadas de acordo com 4 aspectos:
  - Completude: 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
  - Otimalidade: encontra a solução de menor custo
- Complexidade temporal e espacial são medidas em termos de:
  - b : máximo fator de ramificação da árvore de procura *(branching factor)*
  - d : profundidade da solução de menor custo *(depth)*; nó com estado inicial tem profundidade 0
  - m : máxima profundidade do espaço de estados (pode ser ∞)

Estratégias de Procura

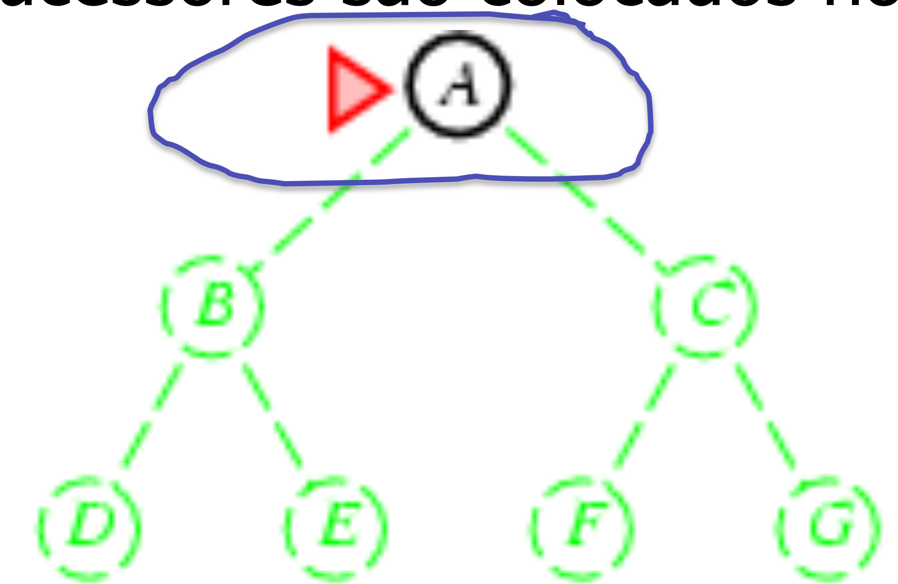
- Complexidade temporal
  - Função do número de nós gerados durante a procura
  - Não de nós expandidos!
    - Tempo para expandir um nó depende do número de nós por ele gerados
- Complexidade espacial
  - Função do número de nós guardados em memória

Procura Não Informada

- Estratégias de procura não informada usam apenas a informação disponível na definição do problema
  - Largura Primeiro
  - Custo Uniforme (algoritmo de Dijkstra)
  - Profundidade Primeiro
  - Profundidade Limitada
  - Profundidade Iterativa
  - Bi-Direcional
- Também chamadas estratégias de procura cega

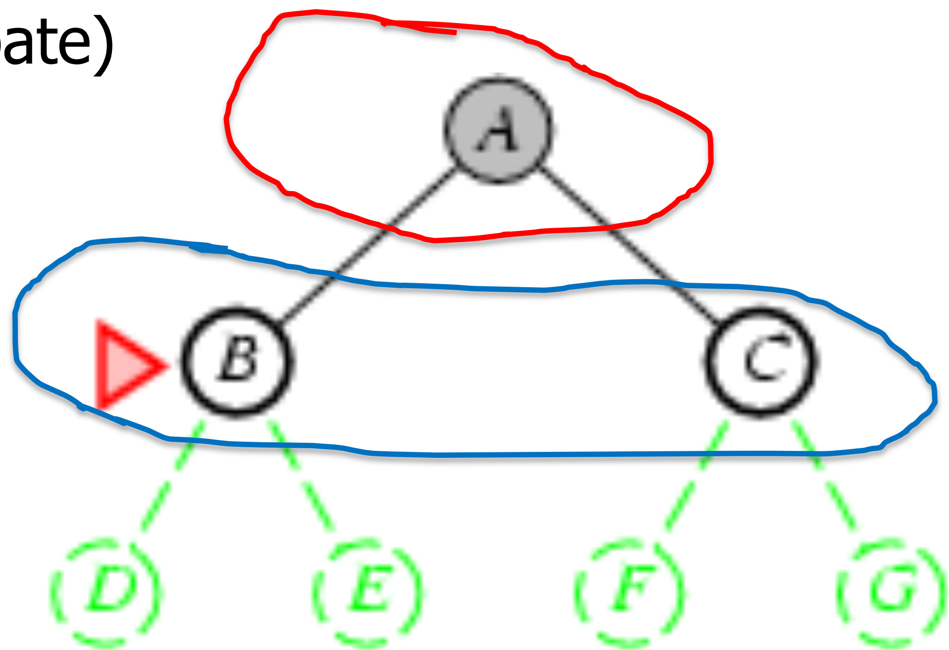
Procura Largura Primeiro

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



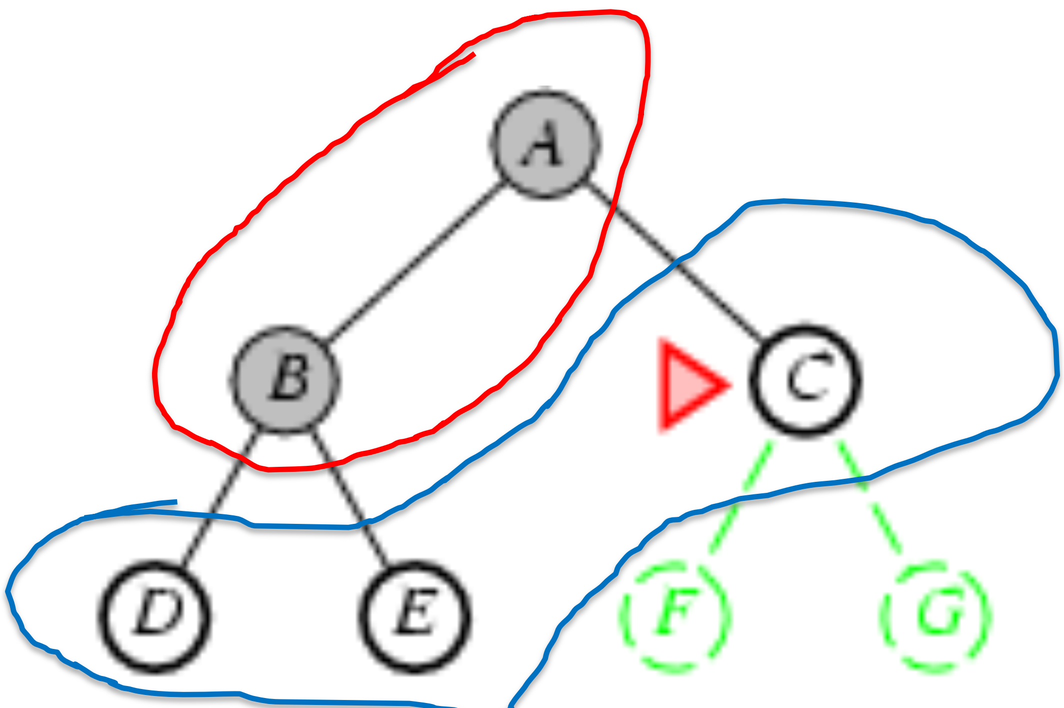
Procura Largura Primeiro

- 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)



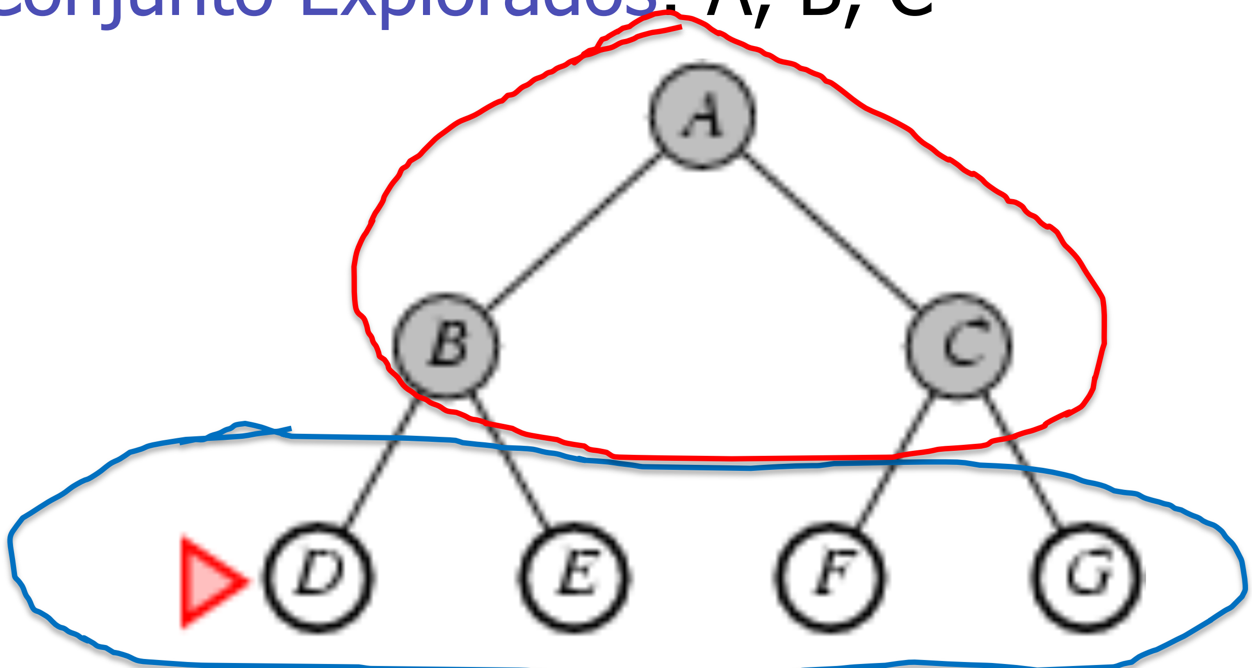
Procura Largura Primeiro

- 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



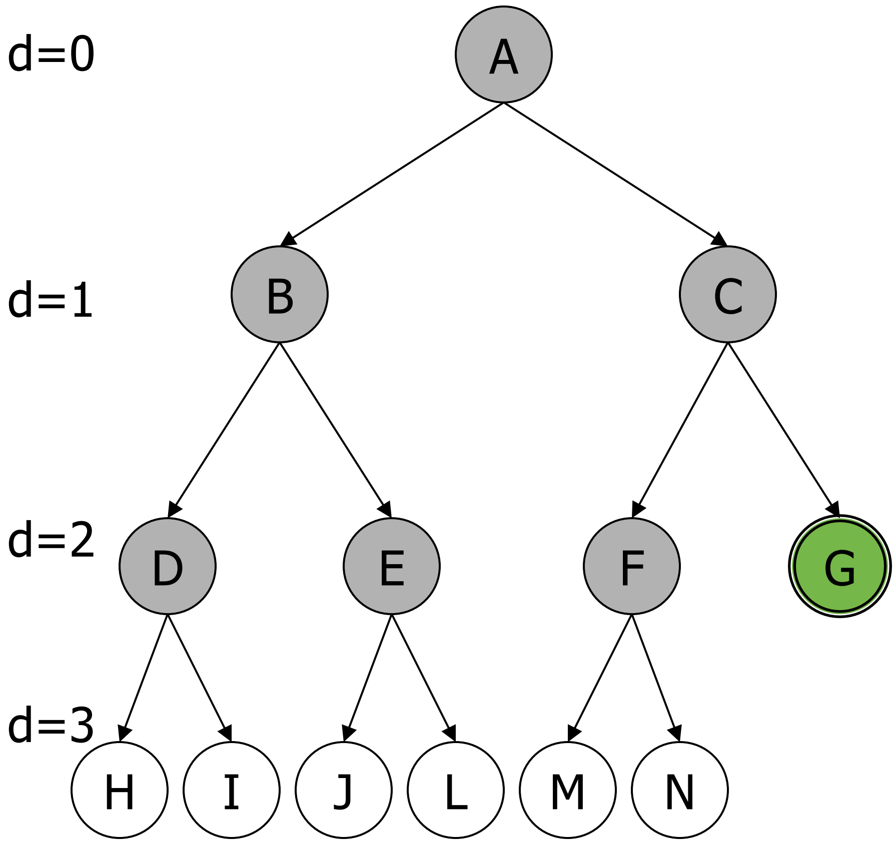
Procura Largura Primeiro

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

PLP: teste objetivo?

- Antes da expansão?
- Na geração?

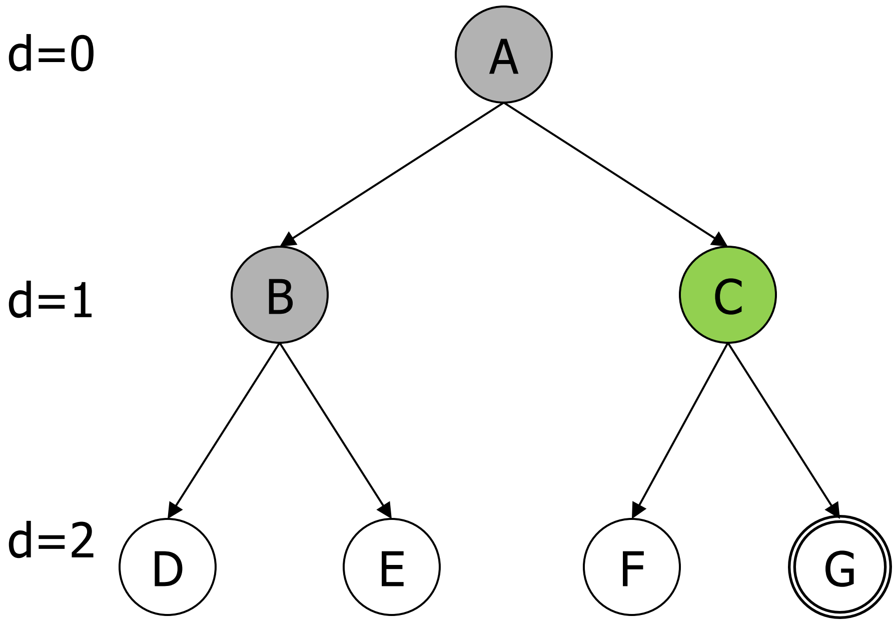
PLP com teste na expansão



- Estado objetivo G
- Fator 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, $O(b^{d+1})$

b: máximo fator de ramificação da árvore de procura
d: profundidade da solução de menor profundidade

PLP com teste na geração



- Estado objetivo G
- Fator 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? $b+b^2+b^3+\ldots+b^d = O(b^d)$, i.e. exponencial em d
- Espaço? $O(b^d)$ (todos os nós por expandir em memória)
- Ótima?
  - Sim, se custo de caminho for uma função não-decrescente da profundidade (e.g. se custo = 1 por ação)
    - i.e. a solução ótima é a que está mais acima na árvore
  - logo não é ótima no caso geral

b: máximo fator de ramificação da árvore de procura

d: profundidade da solução de menor profundidade

Problema PLP

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

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Procura Custo Uniforme

- *Corresponde ao algoritmo de Dijkstra da área de Theoretical Computer Science*

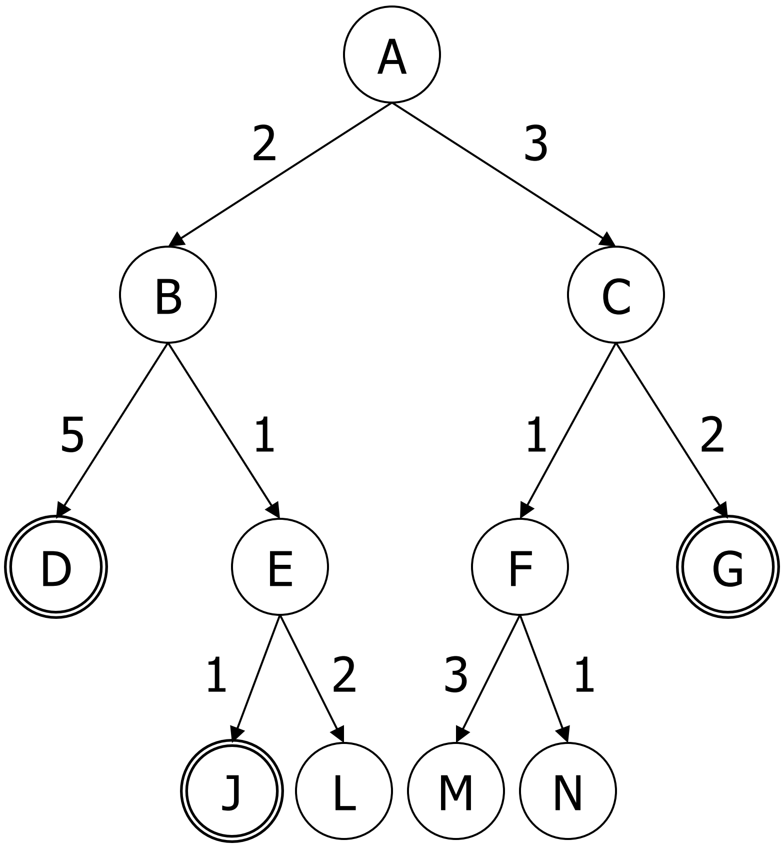
- Designação de custo uniforme deriva de expansão ser realizada em "ondas" de custo uniforme

- 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

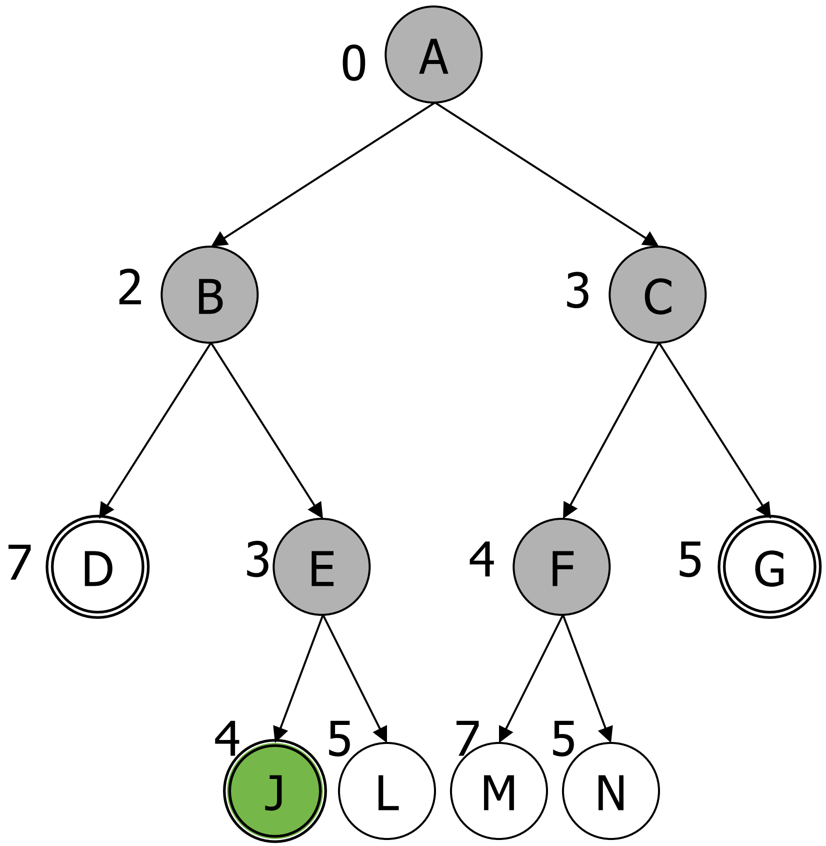
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

Procura Custo Uniforme



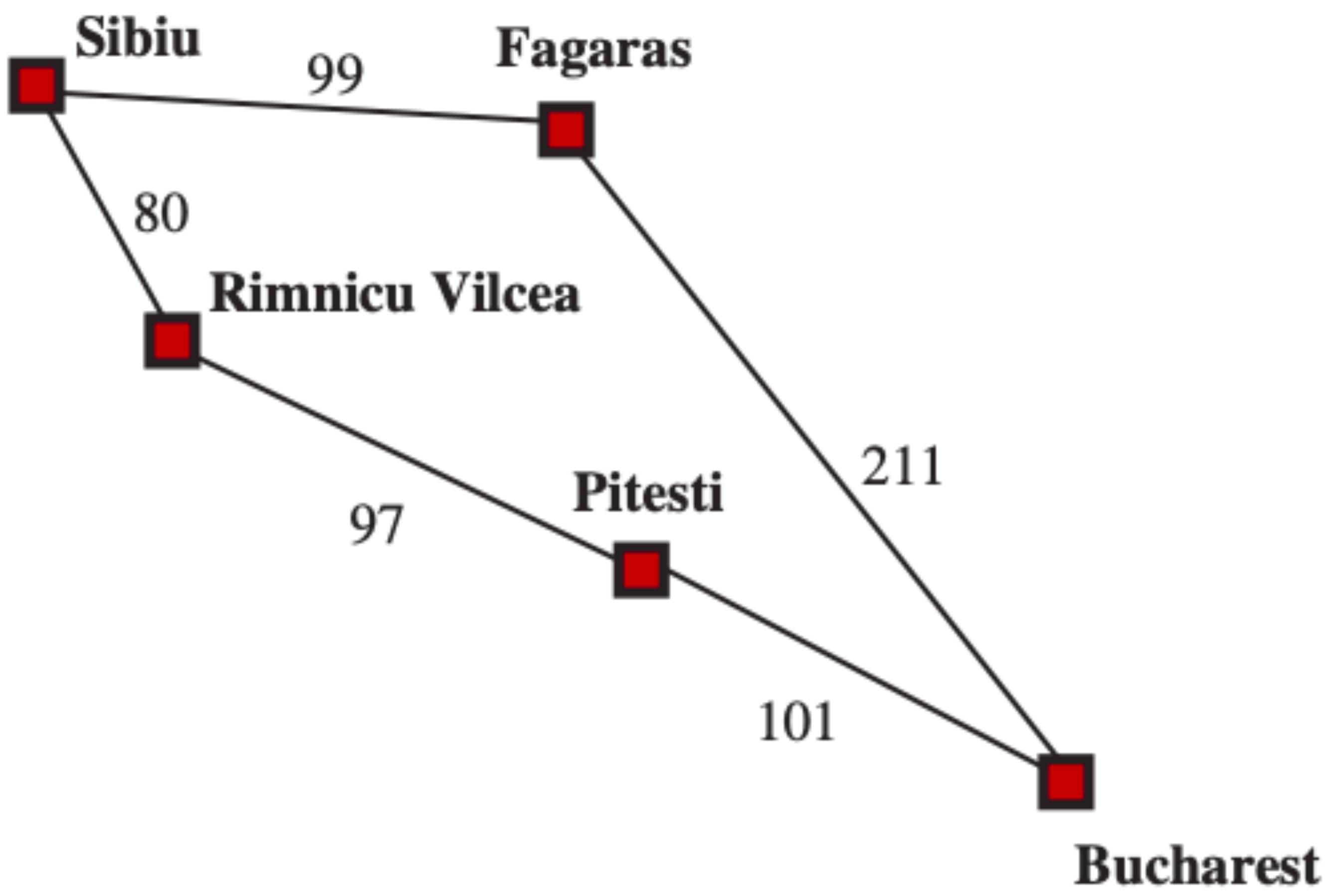
- Custo associado a cada ramo
- Ordem de expansão dos nós?
  - Desempate: ordem alfabética
- Solução encontrada?

Procura Custo Uniforme

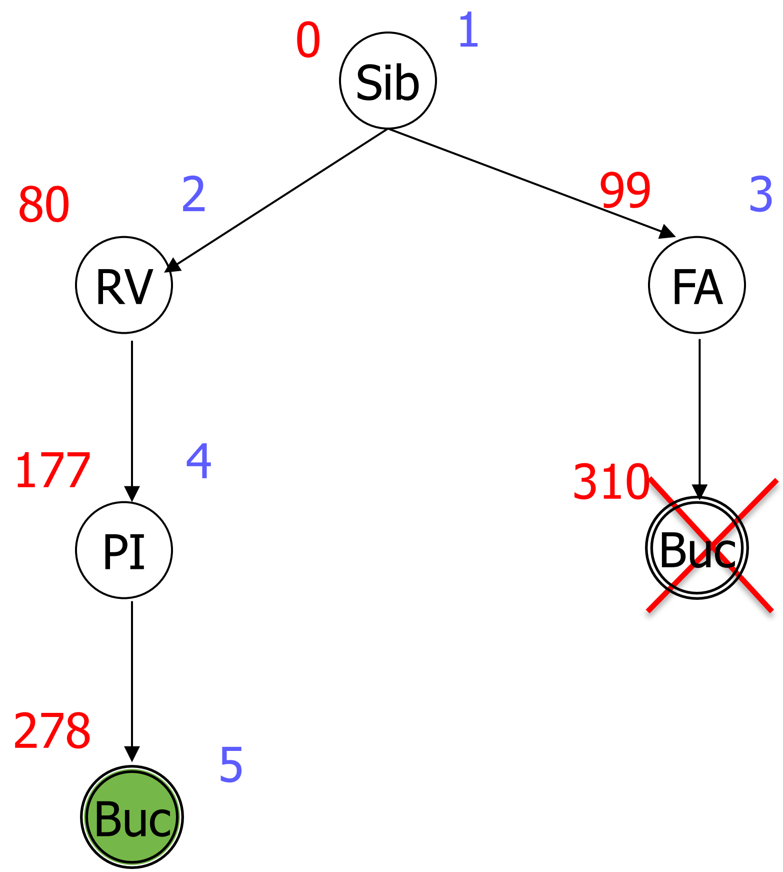


- Ordem de expansão dos nós?
  - A(0), B(2), C(3), E(3), F(4),
- Solução encontrada?
  - J (custo 4)

Exemplo: Sibiu → Bucharest



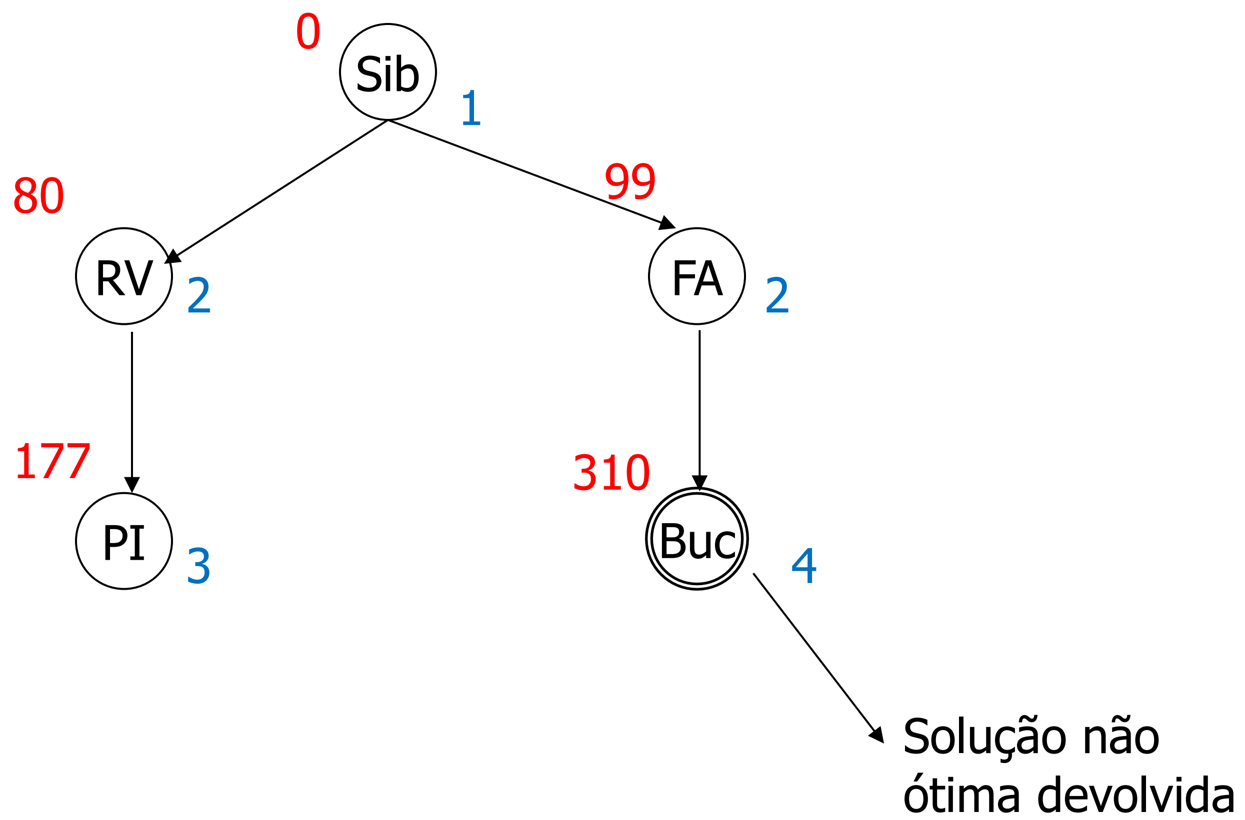
Exemplo: Sibiu → Bucharest



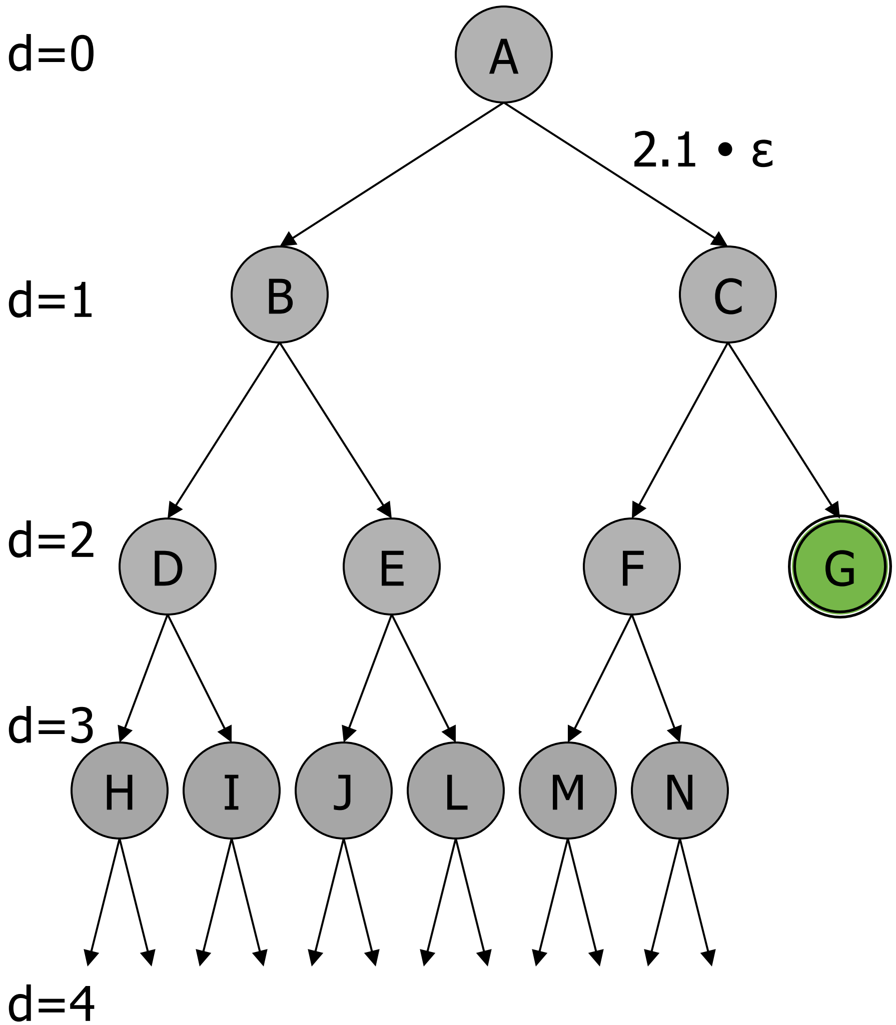
Teste Objetivo PCU

- Porque é que o teste objetivo é feito na expansão e não na geração?
  - Senão perdemos otimalidade

PCU com teste geração



Complexidade: exemplo



- Todos os ramos com custo ε, com exceção do ramo assinalado
- Objetivo G ($C^*=3.1 \bullet \varepsilon$)
- Fator de ramificação b=2
- Tempo e Espaço
  - $1+2^1+2^2+(2^3-2)+(2^4-4)$
  - $O(2^4)$, i.e, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
    - vs $O(b^{d+1})$ para PLP

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

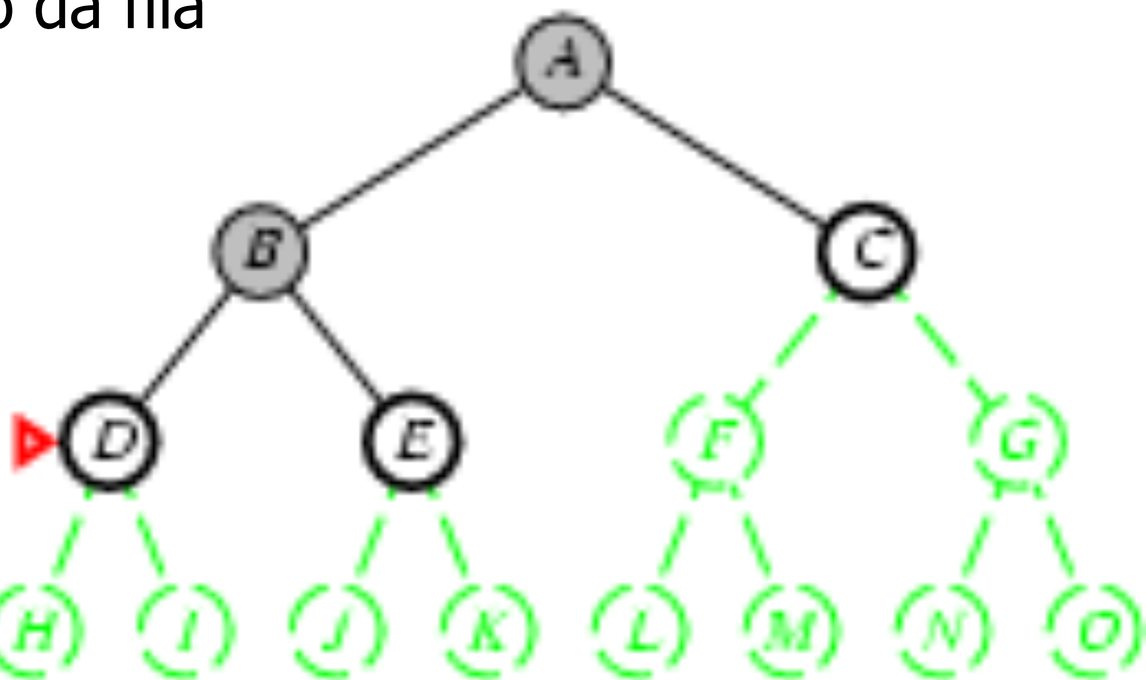
- Tempo? número de nós com $g \leq$ custo da solução ótima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ onde C^* é o custo da solução ótima
  - Todos os ramos com o mesmo custo → $O(b^{1+\lfloor C^*/\varepsilon \rfloor}) = O(b^{d+1})$

- Espaço? número de nós com $g \leq$ custo da solução ótima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

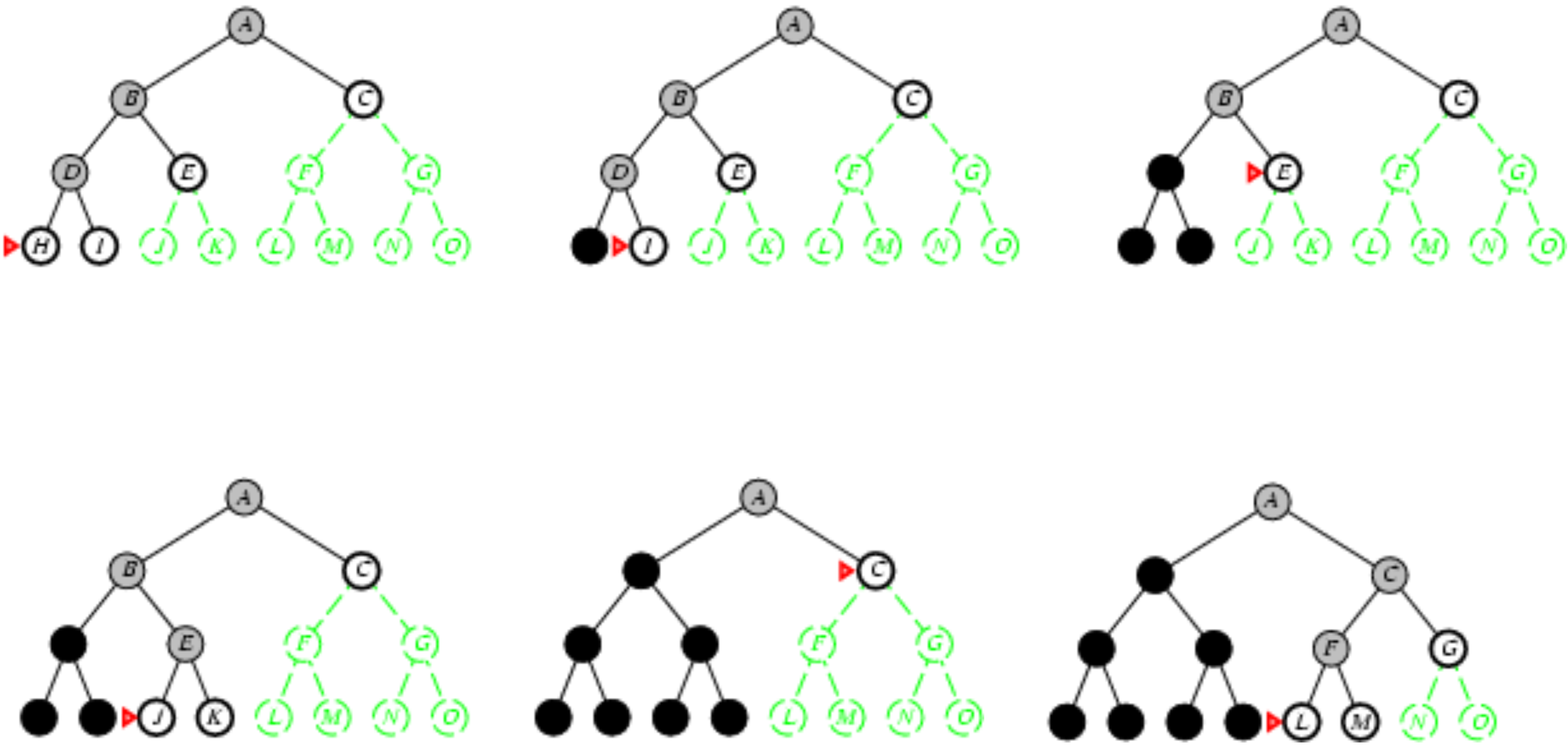
- Ótima? Sim – nós expandidos por ordem crescente de g

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? $O(b*m)$ - espaço linear (só um caminho)
- Ótima? Não

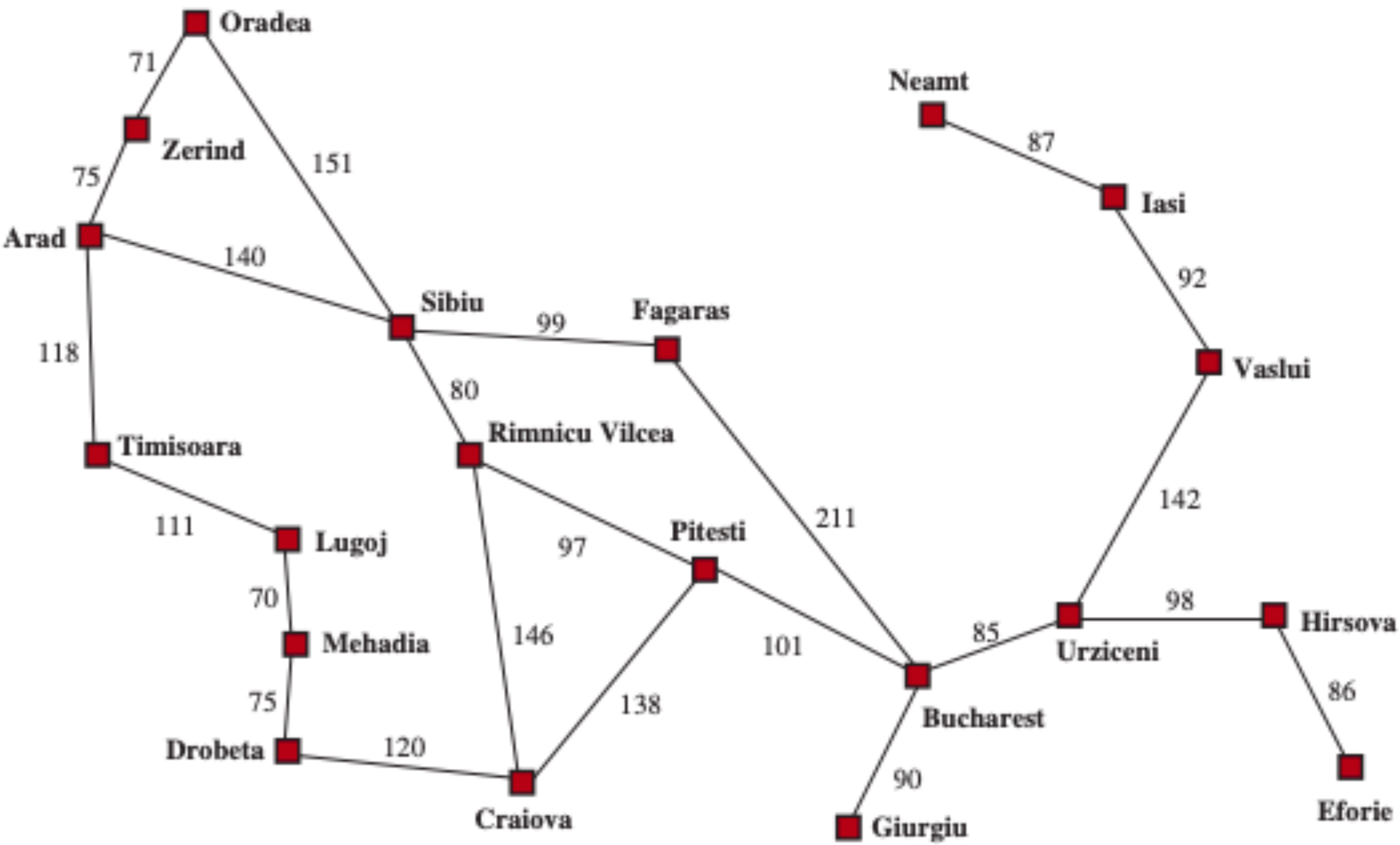
P. Profundidade Primeiro

- 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 $O(bl)$

Exemplo: diâmetro? 9



Distância máxima de 9 troços entre quaisquer duas cidades

P. Profundidade Limitada

function DEPTH-LIMITED-SEARCH(*problem*, ℓ) **returns** a node or *failure* or *cutoff*
  frontier ← a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element
  result ← *failure*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    if DEPTH(*node*) >= ℓ **then**
      result ← *cutoff*
    else if not IS-CYCLE(*node*) **do**
      for each *child* **in** EXPAND(*problem*, *node*) **do**
        add *child* to *frontier*
  return *result*

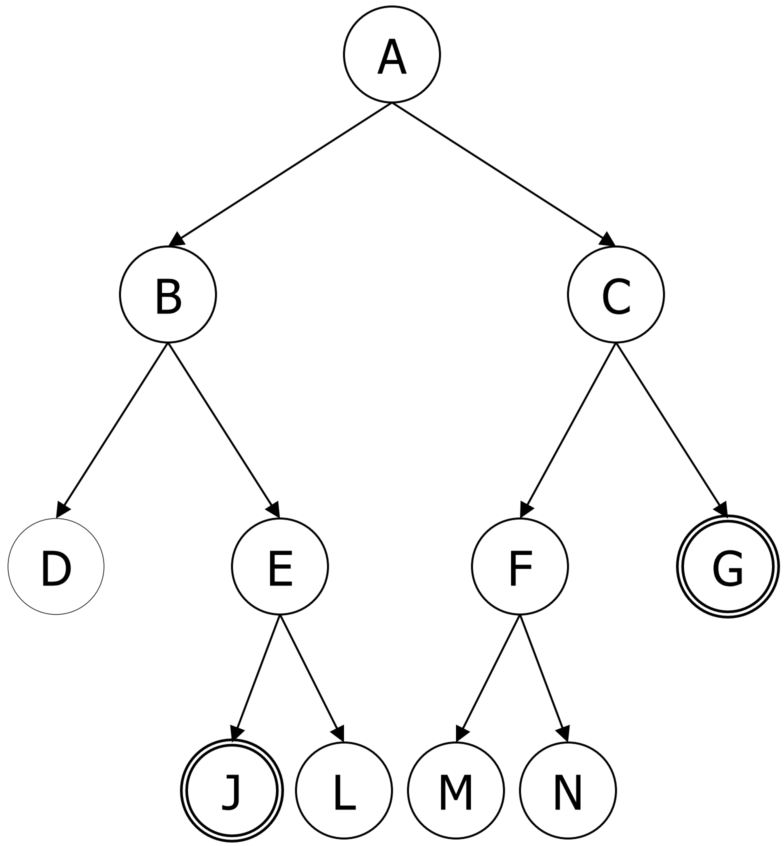
Implementação Recursiva

- 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

Implementação Teste Objetivo

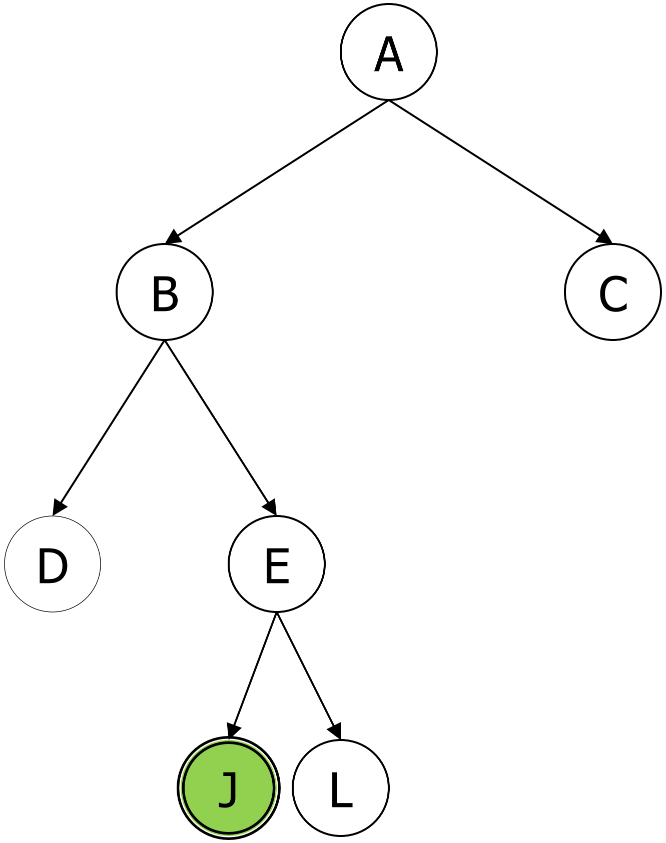
- Expande sempre o nó de maior profundidade
- Procura Melhor Primeiro
  - Função f dada por simétrico da profundidade
- Teste objetivo feito antes da expansão!

Profundidade Limitada: exemplo



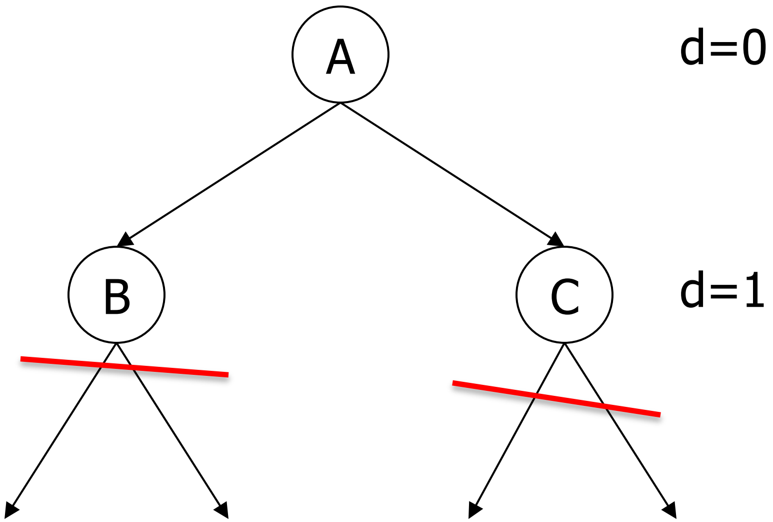
- Solução encontrada?
  - Profundidade Primeiro
  - Profundidade Limitada
    - l =1
    - l =2

Profundidade Limitada: exemplo



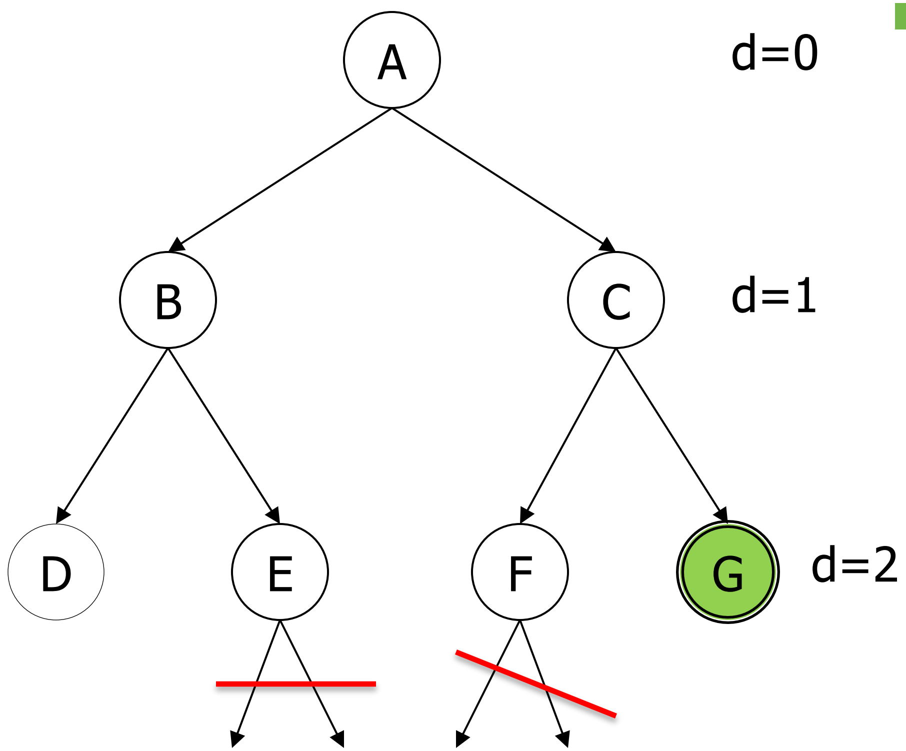
- Solução encontrada?
  - Profundidade Primeiro

Profundidade Limitada: exemplo



- Solução encontrada?
  - Profundidade Limitada
    - l =1
    - Output: corta!

Profundidade Limitada: exemplo



- Solução encontrada?
  - Profundidade Limitada
    - l =2
    - Output: solução

P. Profundidade Iterativa

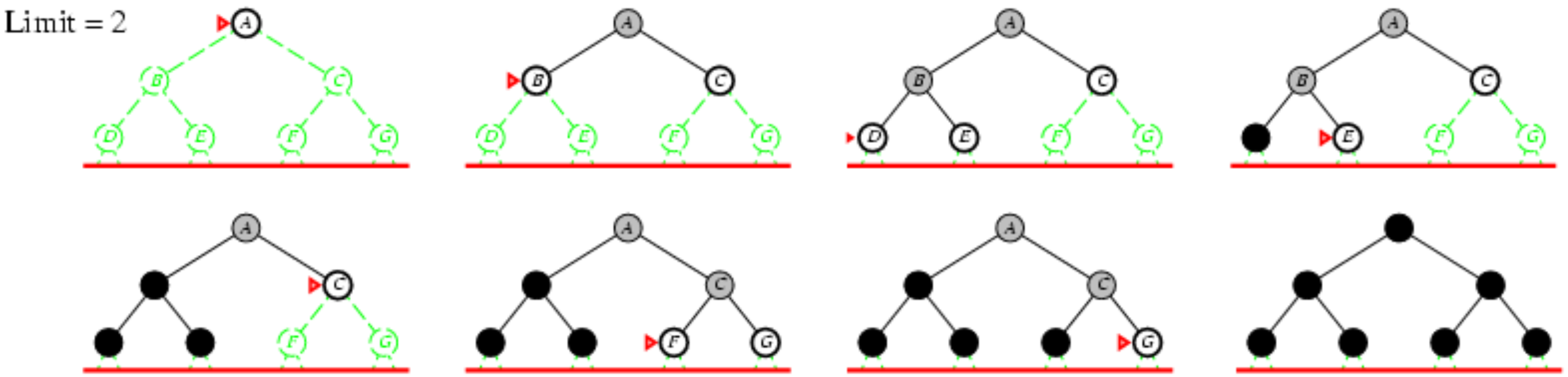
- Profundidade limitada com limite incremental: l=0, l=1, l=2, l=3, ..., l=d
- Combina vantagens da largura primeiro e da profundidade primeiro

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ *cutoff* **then return** *result*

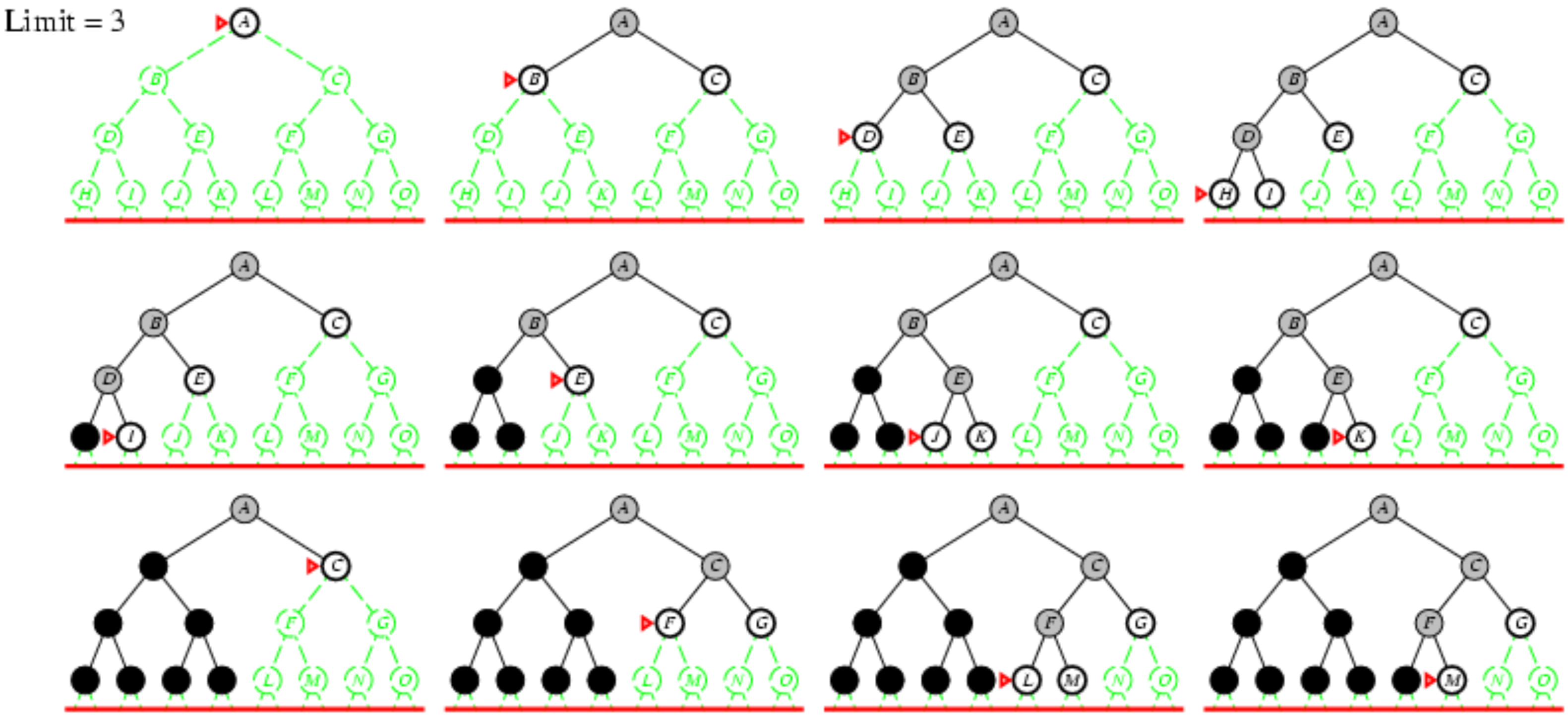
Profundidade Iterativa l=0

Profundidade Iterativa l=1

Profundidade Iterativa $l=2$



Profundidade Iterativa l=3



Profundidade Iterativa

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

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

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

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

- 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%

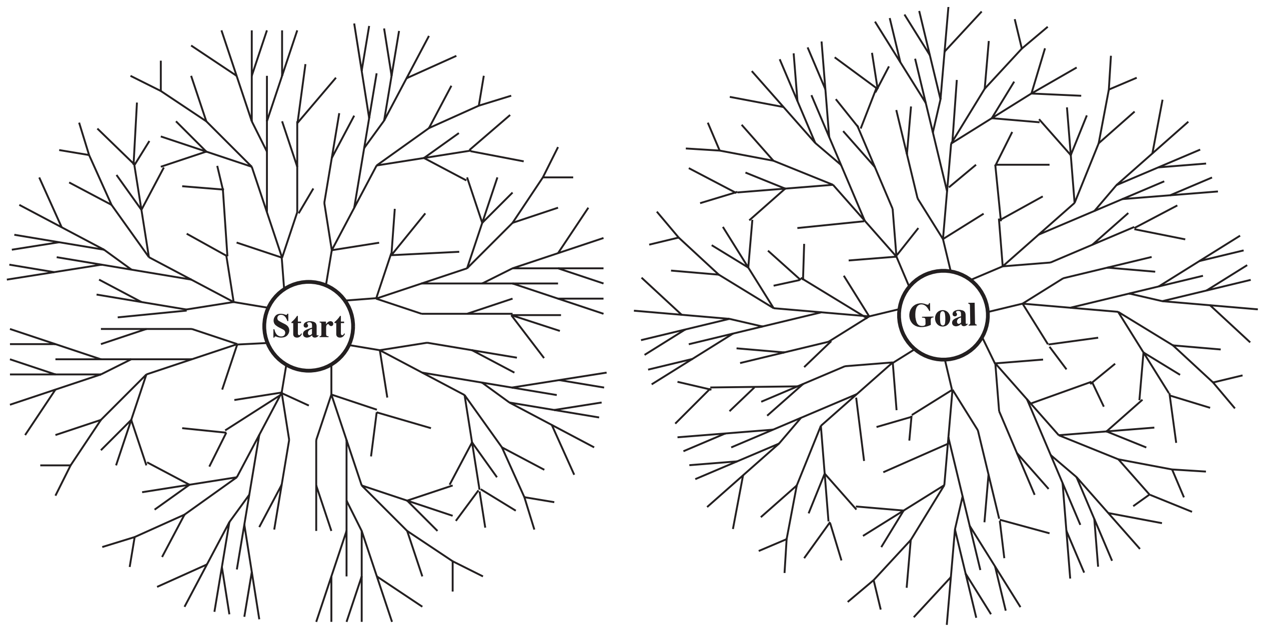
Profundidade Iterativa: propriedades

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

Procura Bi-Direcional

- 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: $b^{d/2} + b^{d/2} << b^d$
- Necessidade de calcular eficientemente os predecessores de um nó
- Problemática quando estados objetivos são descritos implicitamente (por exº, checkmate)

Procura Bi-Direcional



Procura Bi-Direcional

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                          // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                          // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
          // Expand node on frontier; check against the other frontier in reached_2.
          // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Procura Bi-Direcional: propriedades

- Completa? Sim, se *p* é finito e se executa procura em largura primeiro em ambas as direções
- Tempo? $O(b^{d/2})$
- Espaço? $O(b^{d/2})$
- Ótima? 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	$O(b^{d})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{m})$	$O(b^{l})$	$O(b^{d})$	$O(b^{d/2})$
Espaço	$O(b^{d})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Ótima?	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] ótima se todos os ramos têm o mesmo custo
- [d] se ambas as direções executam procura em largura primeiro