

# PROCURA CEGA

## CAPÍTULO 3

---

Parcialmente adaptado de  
<http://aima.eecs.berkeley.edu>

# Resumo

- Agentes resolvidor de problemas
- Tipos de problemas
- Formulação de problemas
- Problemas típicos
- Algoritmos básicos de procura

# Agente resolvidor de problemas

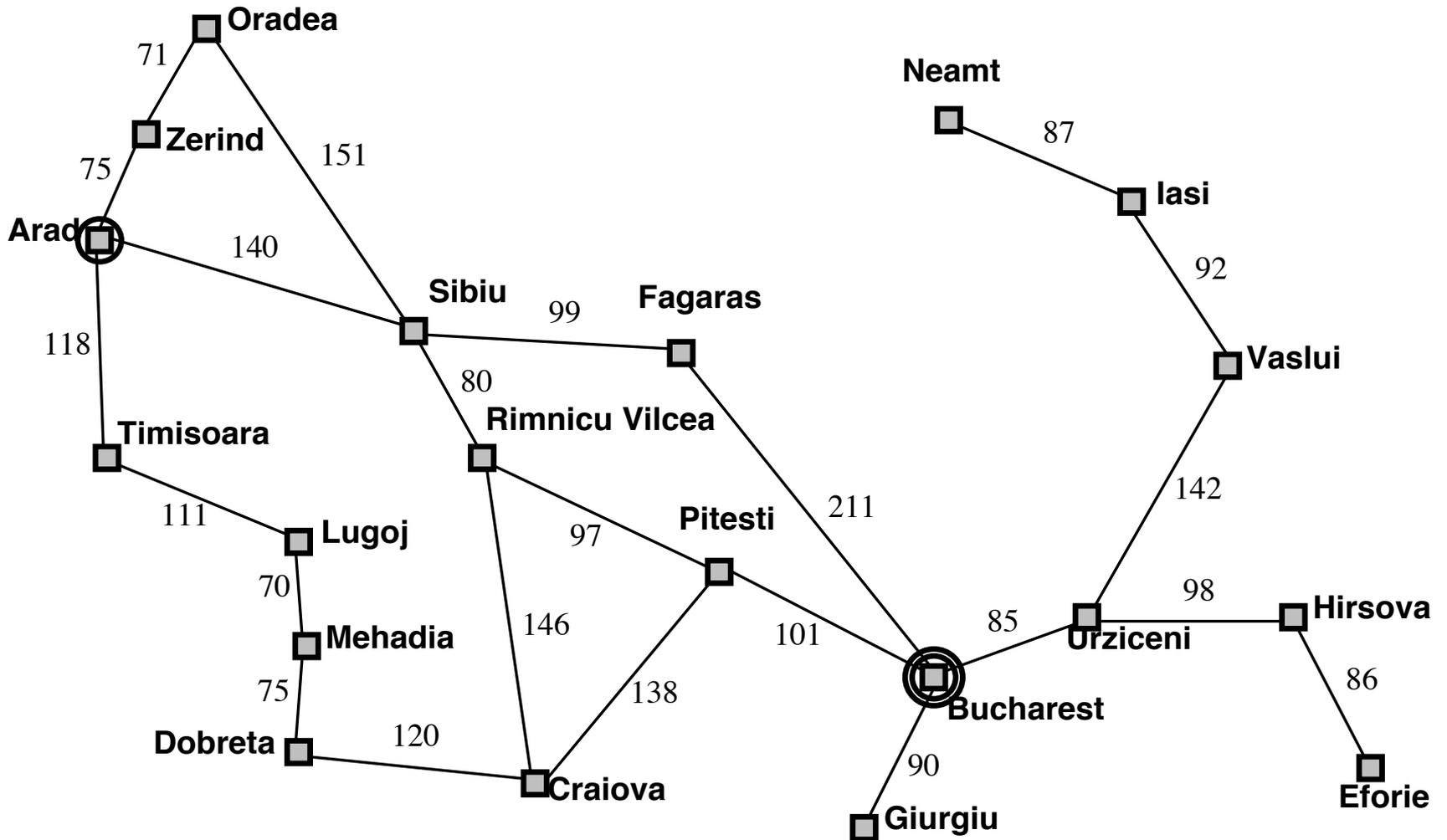
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

# Exemplo: Roménia

- De férias na Roménia; correntemente em Arad.
- Voo sai amanhã de Bucareste
- **Formular objectivo:**
  - Chegar a Bucareste
- **Formular problema:**
  - **estados:** várias cidades
  - **acções:** guiar entre as cidades
- **Solução:**
  - Sequência de cidades: Arad, Sibiu, Fagaras, Bucareste

# Exemplo: Roménia

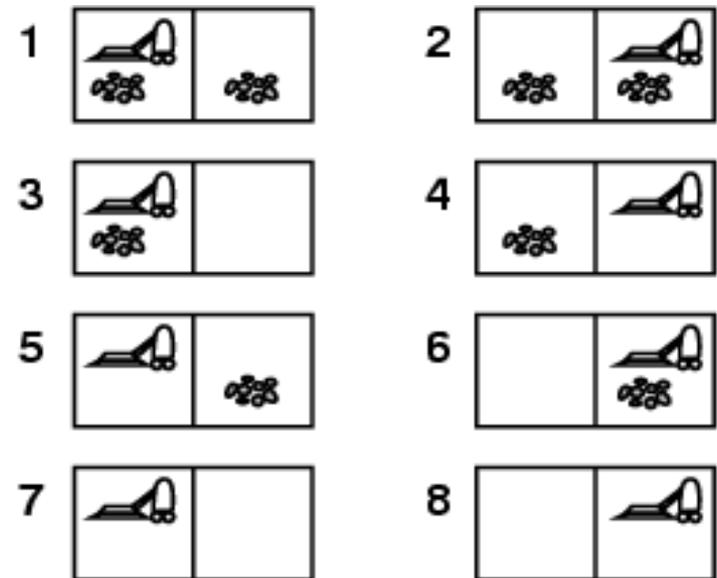


# Tipos de problemas

- **Determinista, observável** → problema com estado único
  - Agente sabe exatamente em que estado se encontrará; solução é uma sequência de ações
- **Não observável** → problema de conformidade
  - Agente pode não saber onde está; caso exista, solução é sequência de ações
- **Não determinista e/ou parcialmente observável** → problema de contingência
  - Percepção fornece **nova** informação acerca do estado corrente
  - Solução é uma árvore ou plano de ação
  - Habitualmente **intercala** procura com execução
- **Espaço de estados desconhecido** → problema de exploração (“online”)

# Exemplo: mundo do aspirador

- **Estado único**, início em 5.
  - Solução: [Right, Suck]
- **Conformidade**, início em {1,2,3,4,5,6,7,8}  
**Right** transita para {2,4,6,8}
  - Solução: [Right, Suck, Left, Suck]
- **Contingência**.  
Lei de Murphy. *Suck* pode sujar carpete limpa.  
Percepção local: [A, Clean]
  - Sol: [Right, **while** dirt **then** Suck]



# Formulação de problema de estado único

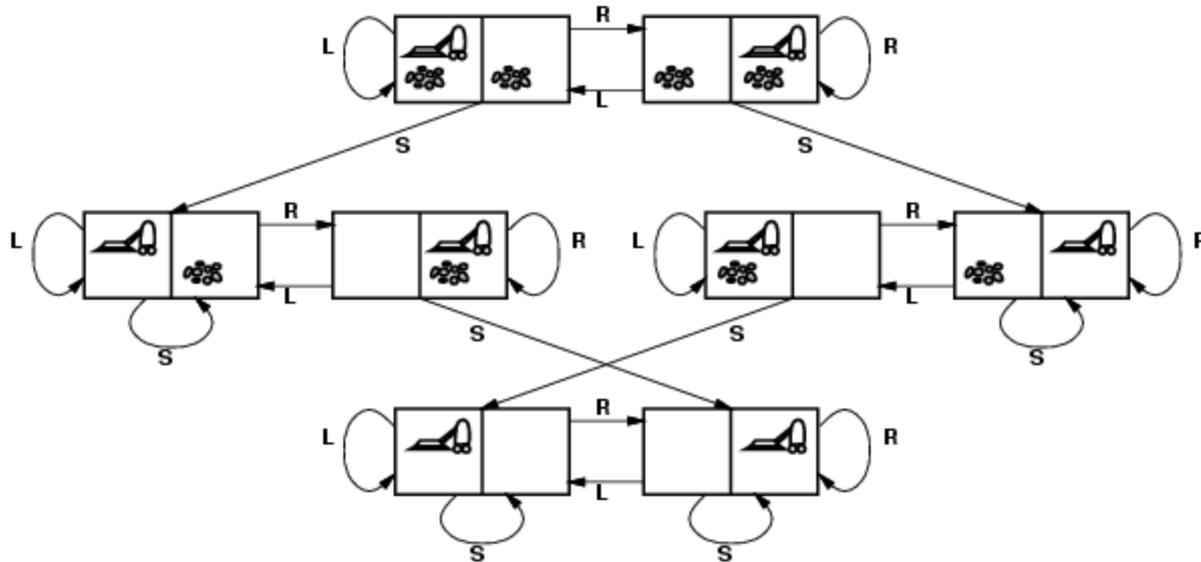
Um **problema de procura** é definido por 4 constituintes:

1. **estado inicial** e.g., “em Arad”
  2. **acções, operadores, função sucessor** ou **modelo de transição**  
**Função sucessor:**  $S(x)$  = conjunto de pares acção-estado
    - e.g.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$**Modelo de transição** especificado pela função  $\text{RESULT}(s,a)$ 
    - e.g.,  $\text{RESULT}(\text{Go}(\text{Zerind}), \text{Arad}) = \text{Zerind}$
  3. **teste objectivo**, pode ser
    - **explícito**, e.g.,  $x = \text{“em Bucareste”}$
    - **implícito**, e.g.,  $\text{XequeMate}(x)$
  4. **custo do caminho** (aditivo)
    - e.g., soma de distâncias, número de acções executadas, etc.
    - $c(x,a,y)$  é o **custo de um passo**, sendo  $\geq 0$
- Uma **solução** é uma sequência de acções que partindo do estado inicial permite atingir o estado objectivo

# Seleccção de um espaço de estados

- A realidade é absurdamente complexa
  - o espaço de estados deve ser **abstraído** para a resolução de problemas
- Estado (abstrato) = conjunto de estados reais
- Ação (abstrata) = combinação complexa de ações reais
  - "Arad → Zerind" representa um conjunto complexo de rotas.
- Para ser concretizável, **qualquer** estado real "em Arad" deve permite chegar a **algum** estado real "em Zerind"
- Solução (abstrata) =
  - Conjunto de caminhos reais que são soluções na realidade
- Cada ação abstrata deverá ser mais simples do que no problema original!

# Grafo de espaços de estados do aspirador



- estados? vector booleanos e inteiro
- acções? *Left, Right, Suck* e *NoOp*
- teste objetivo? tudo limpo
- custo caminho? número de acções  
(1 por acção, 0 para *NoOp*)

## Exemplo: charada de 8

7	2	4
5		6
8	3	1

Start State

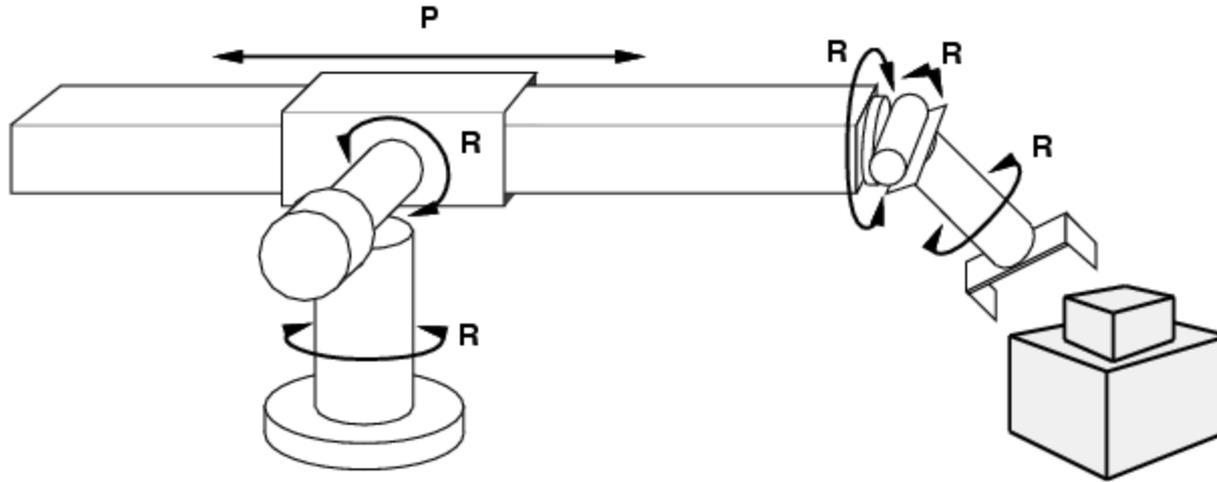
	1	2
3	4	5
6	7	8

Goal State

- estados? inteiros com localização das peças
- acções? movimentar casa nas 4 direcções
- teste objectivo? = estado objectivo
- custo caminho? número de movimentos (1 por movimento)

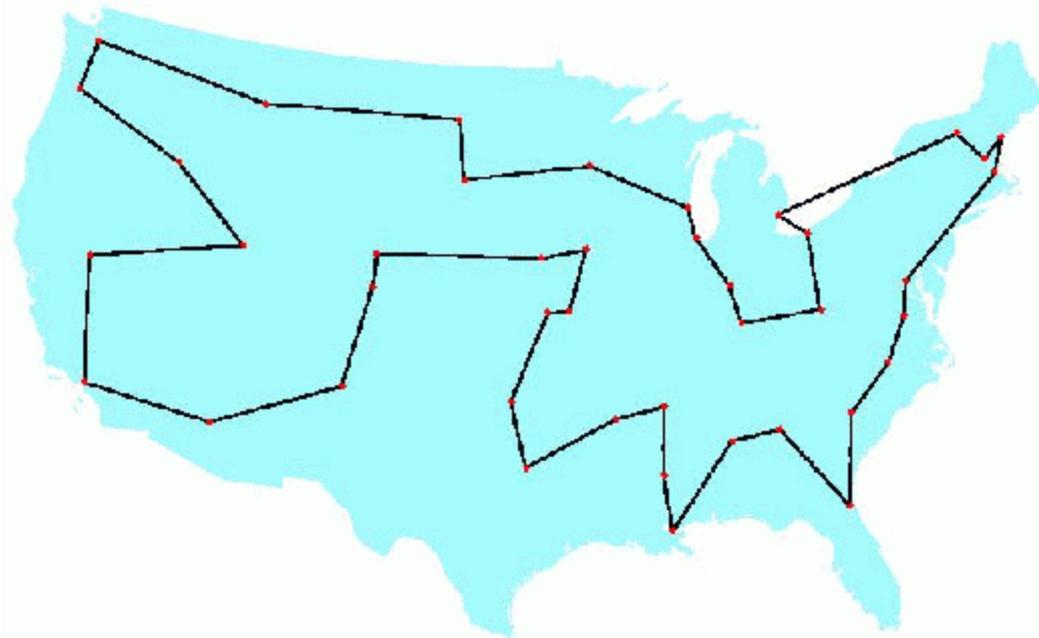
[Nota: solução óptima para a família de charadas-n é NP-hard]

## Exemplo: montagem robótica



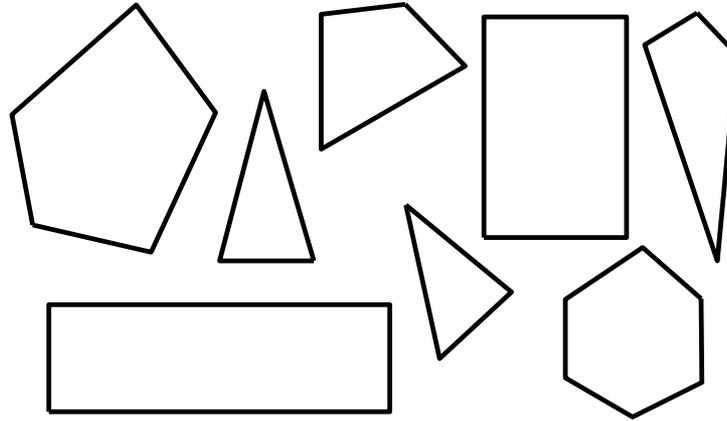
- estados? Coordenadas reais das articulações do robô e dos objetos a montar
- acções? Movimento contínuo das articulações
- teste objectivo? montagem completa
- custo caminho? tempo para executar

## Exemplo: caixeiro viajante



- estados? sequência de cidades sem repetições
- acções? viajar para nova cidade ou voltar à inicial
- teste objectivo? circuito das cidades
- custo caminho? distância total

## Exemplo: navegação robótica



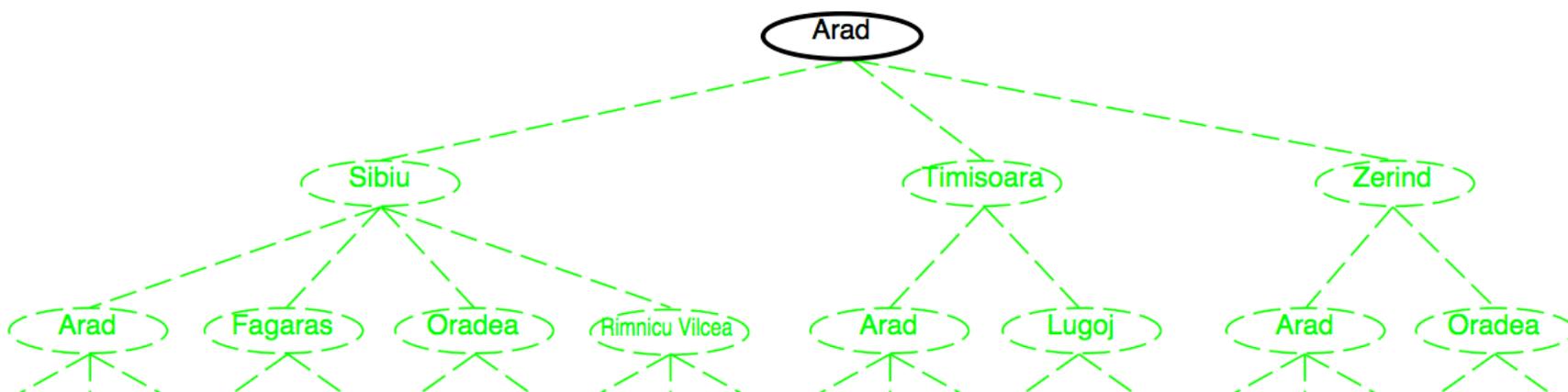
- estados? vértices dos polígonos, posições inicial e final
- acções? viajar para outro vértice
- teste objectivo? chegar ao objectivo
- custo caminho? distância percorrida

# Algoritmos de procura em árvores

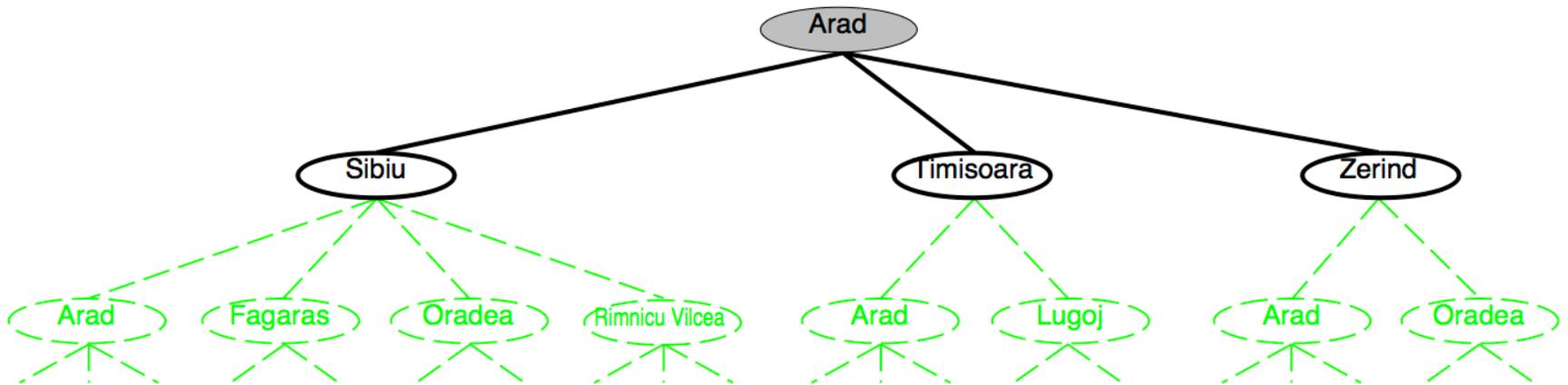
- Ideia básica:
  - Simulação *offline* da exploração do espaço de estados através da geração de sucessores de estados já explorados

```
function TREE-SEARCH( problem ) returns a solution, or failure
  initialize the frontier using the initial state of 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
```

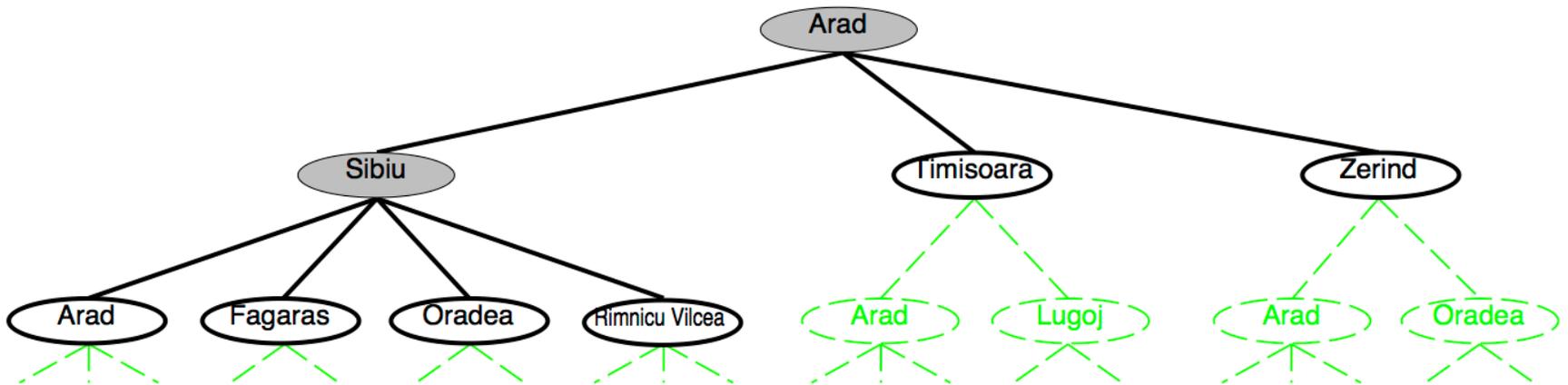
# Exemplo de procura em árvore



# Exemplo de procura em árvore

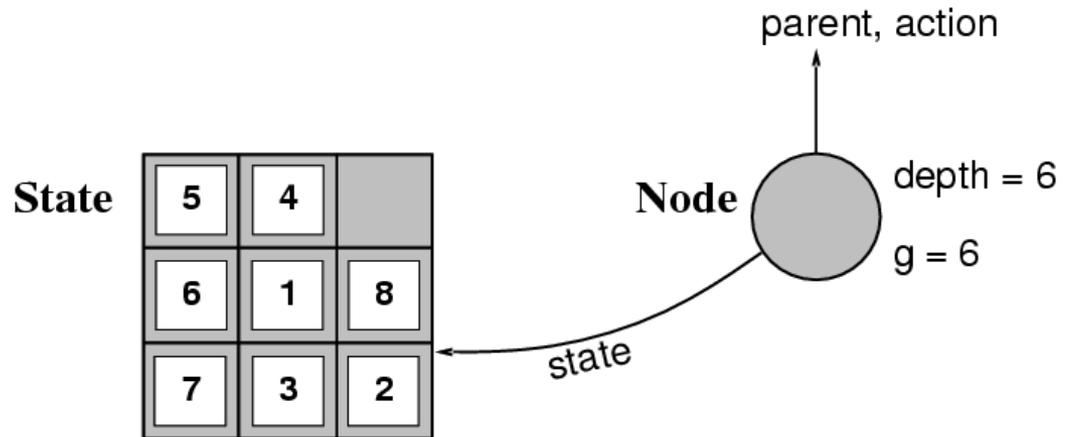


# Exemplo de procura em árvore



# Implementação: estados vs. nós

- Um **estado** é uma (representação) de uma configuração física.
- Um **nó** é uma estrutura de dados constituinte da árvore de procura incluindo o pai, o estado, e outros detalhes relevantes para o algoritmo, e.g. a ação, a profundidade, o custo de caminho acumulado  $g(x)$ , etc...



- A função `CHILD-NODE` cria um novo nó a partir do pai e da acção a executar.
- Estados não têm pais, acções, profundidade ou custo do caminho!

# Implementação: procura genérica em árvores

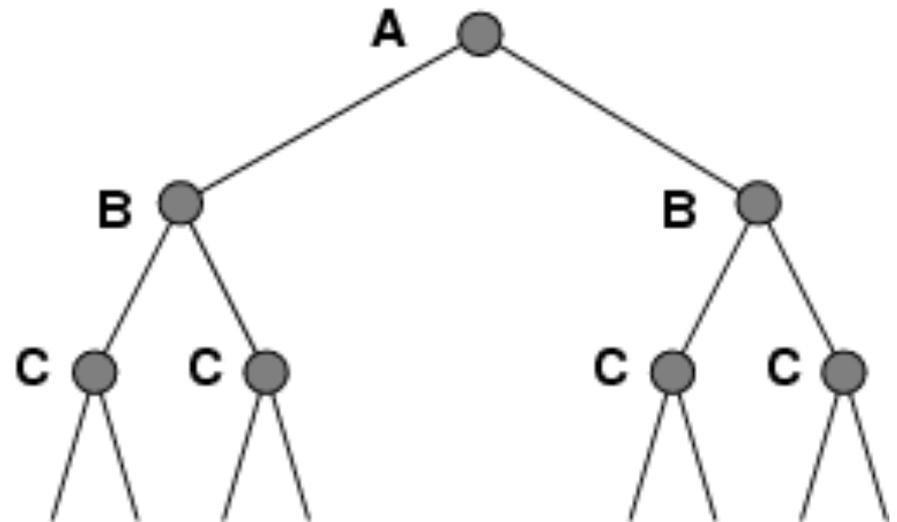
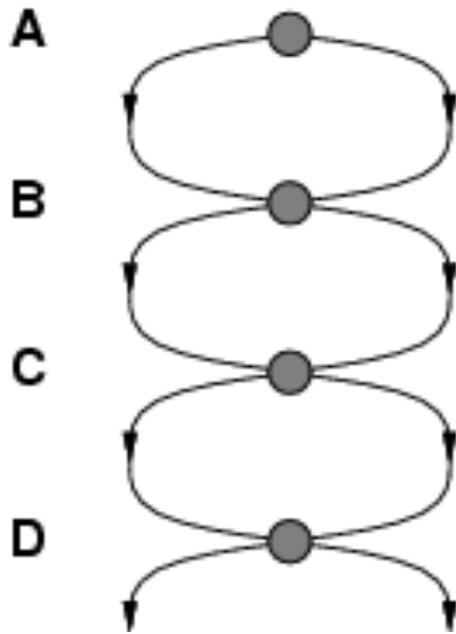
```
function TREE-SEARCH( problem, frontier ) returns a solution, or failure
  node ← node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← INSERT(node, frontier)
  loop do
    if EMPTY?(frontier) return failure
    node ← POP( frontier )
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← INSERT-ALL(EXPAND(node,problem),frontier)
```

```
function EXPAND( node, problem ) returns a set of nodes
  successors ← the empty set
  for each action in problem.ACTIONS (node.STATE) do
    s ← CHILD-NODE(problem,node,action)
    add s to successors
  return successors
```

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

# Estados repetidos

- A não detecção de estados repetidos pode tornar um problema linear num problema exponencial!



# Procura em grafos

```
function GRAPH-SEARCH( problem, frontier ) returns a solution, or failure
  explored ← an empty set
  node ← node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← INSERT(node, frontier)
  loop do
    if EMPTY?(frontier) return failure
    node ← POP( frontier )
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    if node.STATE is not in explored then
      add node.STATE to explored
      frontier ← INSERT-ALL(EXPAND(node,problem),frontier)
```

Esquema genérico! Pode ser melhorado em algumas circunstâncias.

# Estratégias de procura

- Uma estratégia de procura é definida pela **ordem de expansão dos nós**.
- As estratégias são avaliadas segundo as dimensões:
  - **completude**: encontra garantidamente uma solução, caso exista?
  - **complexidade temporal**: número de nós gerados
  - **complexidade espacial**: número máximo de nós em memória
  - **optimalidade**: encontra sempre uma solução de custo mínimo?
- A complexidade temporal e espacial são avaliadas em função de
  - ***b***: factor de ramificação máximo da árvore de procura
  - ***d***: profundidade da solução de custo mínimo
  - ***m***: profundidade máxima do espaço de estados (pode ser  $\infty$ )

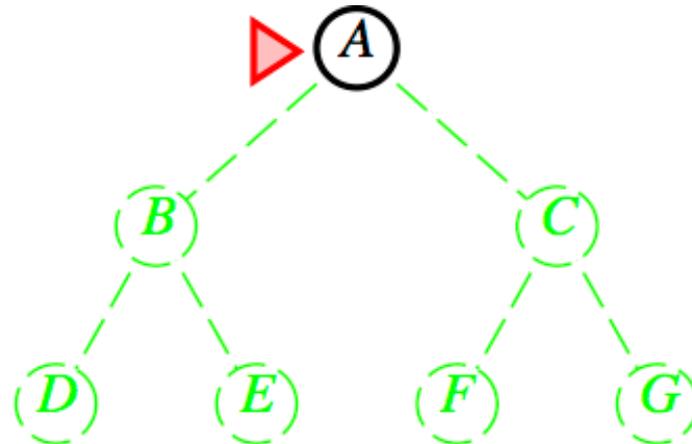
# Estratégias de procura cegas

As estratégias de procura **cegas** (ou não informadas) recorrem apenas à informação disponibilizada no problema

- Procura em largura primeiro (breadth-first)
- Procura de custo uniforme (uniform-cost)
- Procura bidireccional
- Procura em profundidade primeiro (depth-first)
- Procura em profundidade limitada (depth-limited)
- Procura por aprofundamento progressivo (iterative deepening)

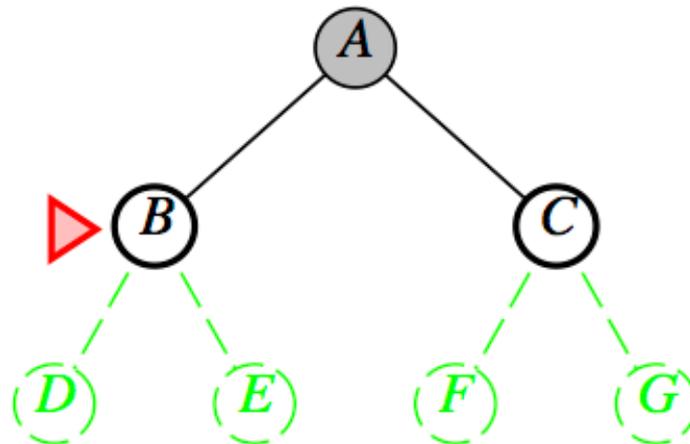
# Procura em largura primeiro

- Expandir um nó de menor profundidade
- **Implementação:** *frontier* é uma fila FIFO; novos sucessores vão para o fim



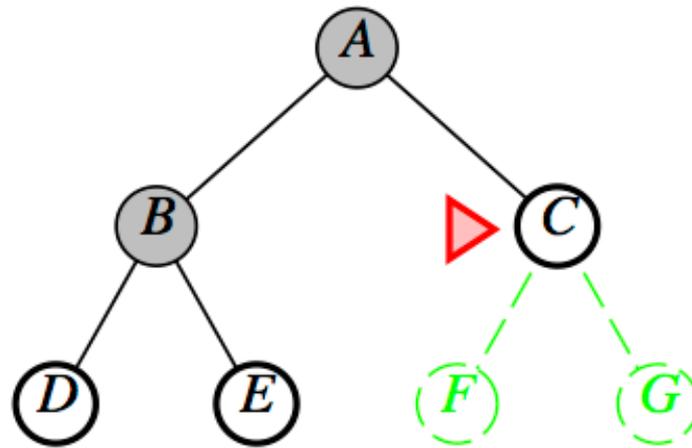
# Procura em largura primeiro

- Expandir um nó de menor profundidade
- **Implementação:** *frontier* é uma fila FIFO; novos sucessores vão para o fim



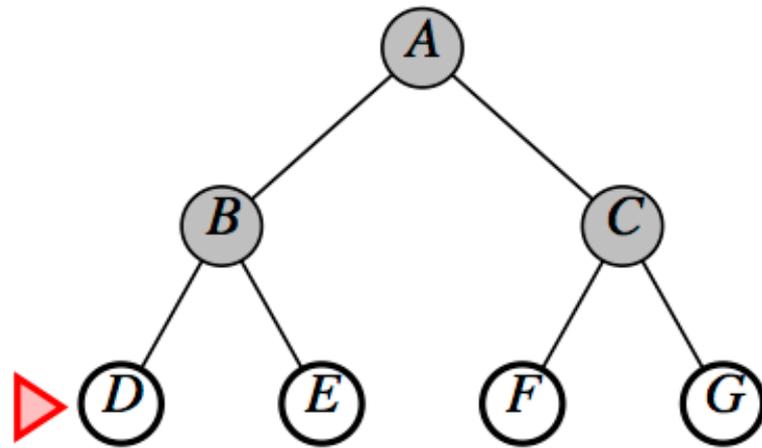
# Procura em largura primeiro

- Expandir um nó de menor profundidade
- **Implementação:** *frontier* é uma fila FIFO; novos sucessores vão para o fim



# Procura em largura primeiro

- Expandir um nó de menor profundidade
- **Implementação:** *frontier* é uma fila FIFO; novos sucessores vão para o fim



## Propriedades da procura em largura primeiro

- Completa? Sim (se  $b$  é finito)
- Tempo?  $1+b+b^2+b^3+\dots +b^d + b(b^d-1) = O(b^{d+1})$
- Espaço?  $O(b^{d+1})$  (mantém todos os nós)
- Optimal? Sim (se custo das acções for idêntico)
  
- **Espaço** é o maior problema (mais do que o tempo)
- A análise anterior só é válida para procura em árvores!

# Requisitos temporais e espaciais da procura em largura

Profundidade	Nós	Tempo	Memória
2	110	1,1ms	107 Kb
4	11 110	111 ms	10,6 Mb
6	$10^6$	11 seg	1 Gb
8	$10^8$	19 minutos	103 Gb
10	$10^{10}$	31 horas	10 Tb
12	$10^{12}$	129 dias	1 Petabytes
14	$10^{14}$	35 anos	99 Petabytes
16	$10^{16}$	3500 anos	10 Exabytes

b=10 gerando 100000 nós/segundo ocupando 1000 bytes/nó

# Procura em Largura Primeiro (otimizada)

```
function BREADTH-FIRST-SEARCH( problem ) returns a solution, or failure
  node ← a node with STATE=problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?( frontier ) then return failure
    node ← POP( frontier ) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE( problem , node , action )
      if child.STATE is not in explored or frontier then do
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child )
        frontier ← INSERT(child , frontier )
```

Complexidade temporal e espacial reduzidas para  $O(b^d)$  em vez de  $O(b^{d+1})$

# Procura de custo uniforme

- Expandir o nó por tratar de menor custo
- **Implementação:**
  - *frontier* = fila ordenada pelo custo do caminho acumulado
- Equivale à procura em largura se custos forem constantes
- **Completa?** Sim, se custo do passo  $\geq \epsilon$
- **Tempo?** nº de nós com  $g \leq$  custo da solução óptima,  $O(b^{1+\text{ceil}(C^*/\epsilon)})$  em que  $C^*$  é o custo da solução óptima
- **Espaço?** nº de nós com  $g \leq$  custo da solução óptima,  $O(b^{1+\text{ceil}(C^*/\epsilon)})$
- **Óptima?** Sim – nós expandidos por ordem crescente de  $g(n)$
- **Espaço** continua a ser o maior problema (mais do que o tempo)

# Procura de custo uniforme (em grafos)

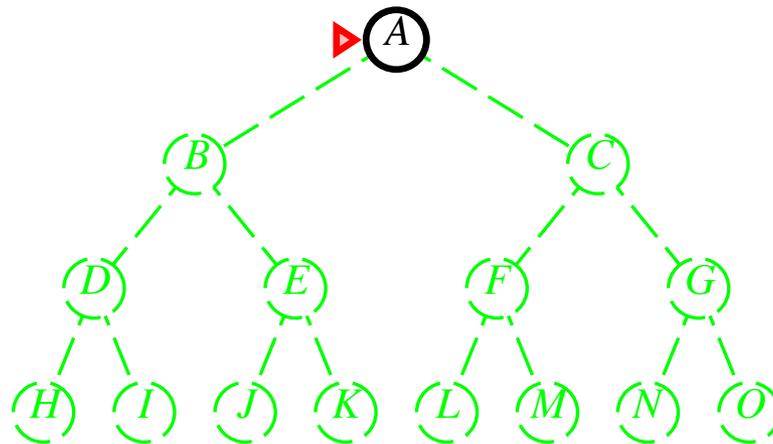
```
function UNIFORM-COST-SEARCH( problem ) returns a solution, or failure
  node ← a node with STATE=problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST with node as the only element
  explored ← a singleton set with node.STATE
  loop do
    if EMPTY?( frontier ) then return failure
    node ← POP( frontier ) /* chooses the node with lowest cost in 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 ← CHILD-NODE( problem , node , action )
      if child.STATE is not in explored or frontier then do
        frontier ← INSERT(child , frontier )
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Se já houver um nó na fronteira com o mesmo estado mas um custo maior, deve ser substituído

O Teste de objectivo volta a ser feito como na versão inicial, pois o primeiro nó com o objectivo pode ser sub-óptimo

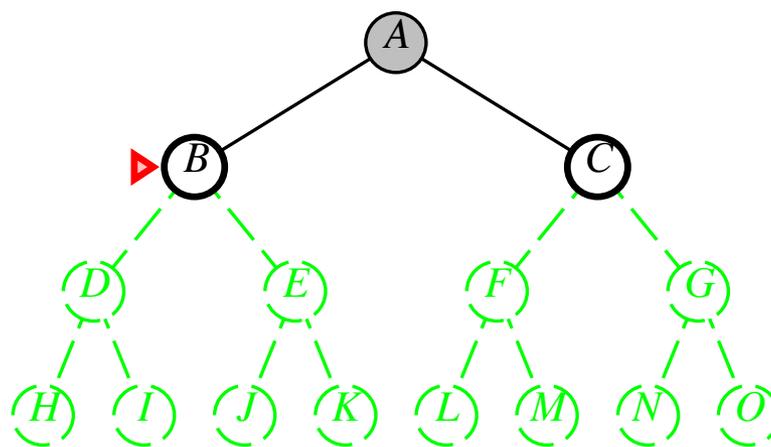
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



# Procura em profundidade primeiro

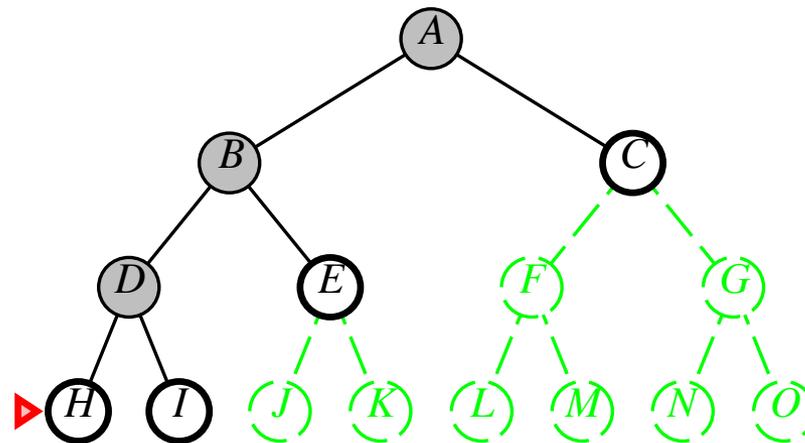
- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente





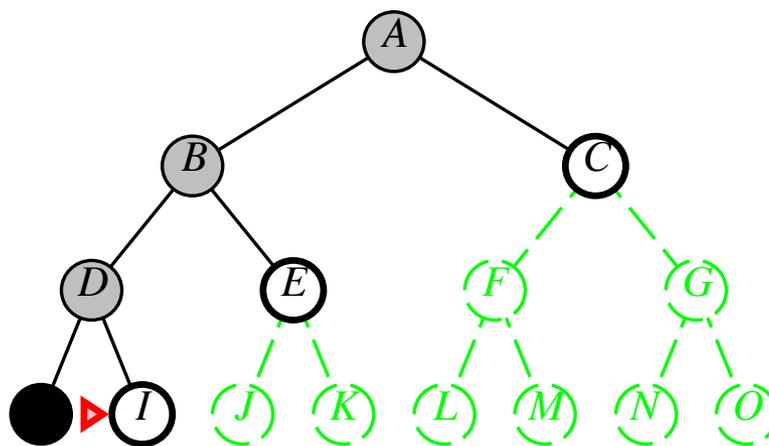
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



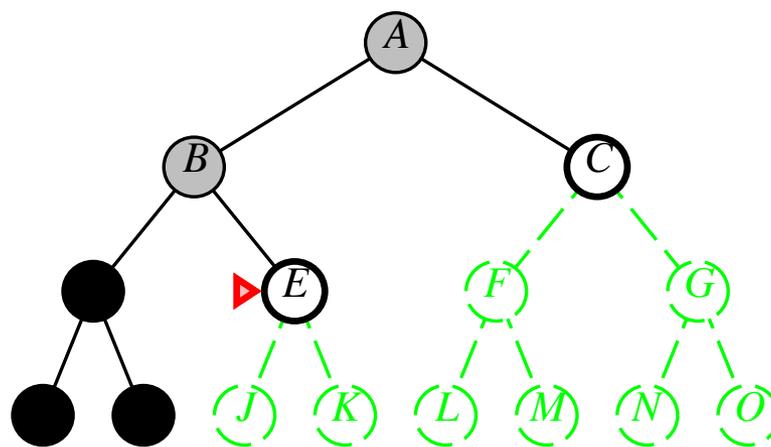
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação**: *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



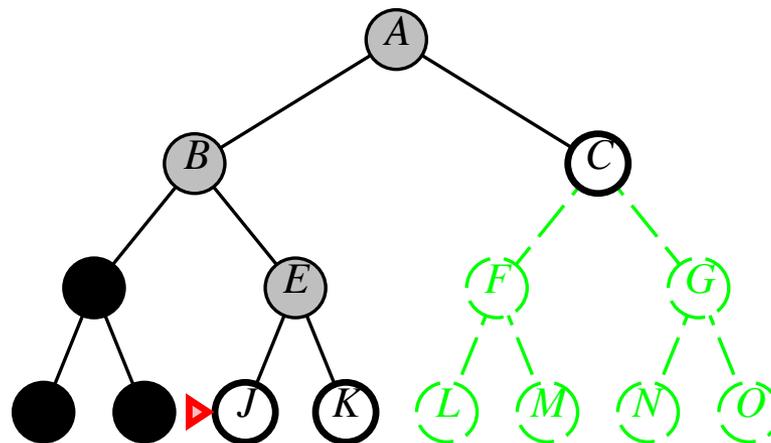
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



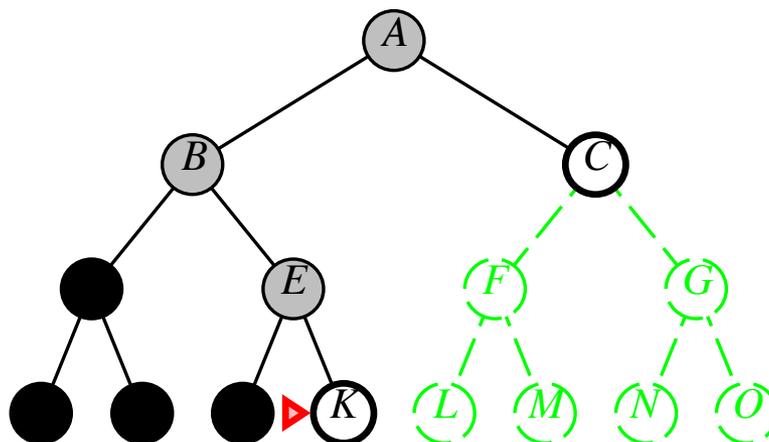
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



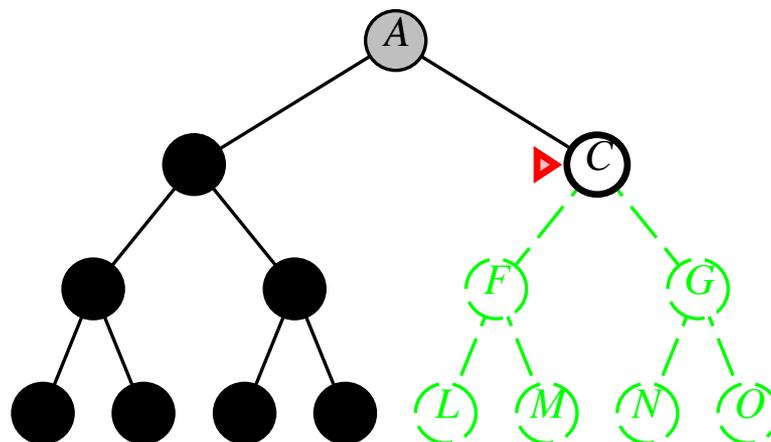
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação**: *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



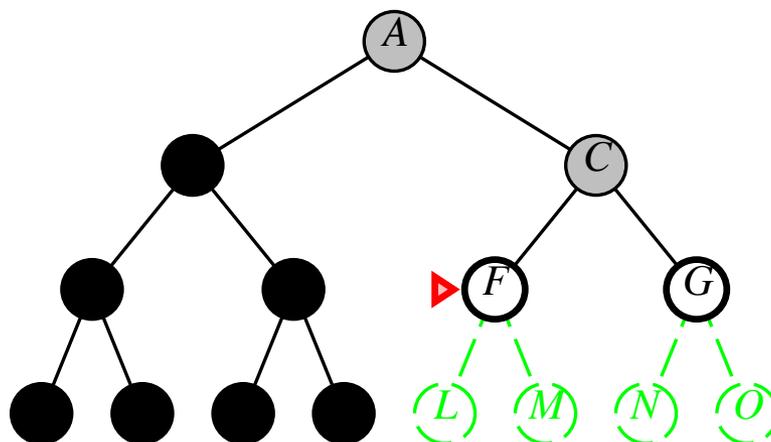
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



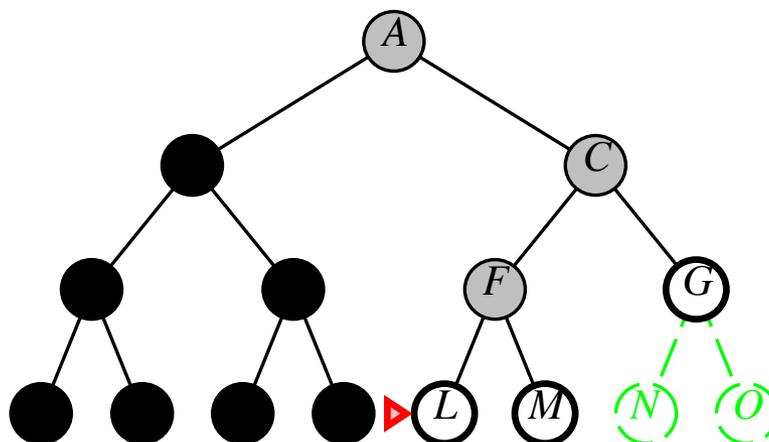
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



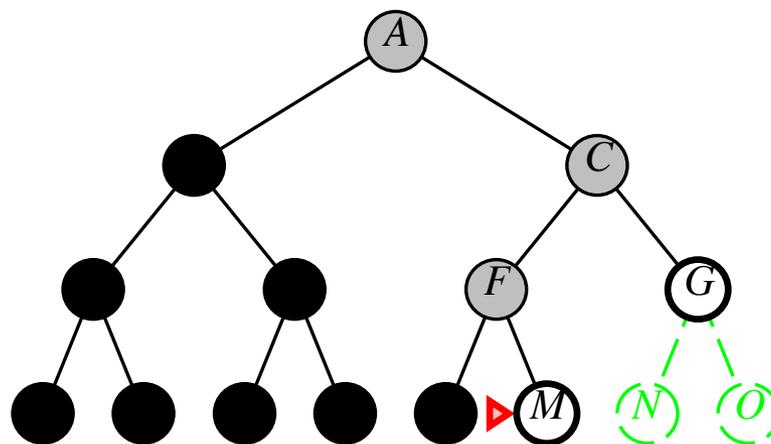
# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação**: *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



# Procura em profundidade primeiro

- Expandir um dos nós mais profundos
- **Implementação:** *fronteira* = pilha LIFO, i.e., colocar sucessores à frente



# Procura em profundidade primeiro

- Completa? Não: falha em espaços de profundidade infinita, espaços com ciclos
  - Modificação para evitar espaços repetidos no mesmo caminho
  - → completa para espaços finitos
- Tempo?  $O(b^m)$ : terrível se  $m$  muito maior do que  $d$ 
  - mas se as soluções são densas, pode ser muito mais eficiente do que a procura em largura primeiro
- Espaço?  $O(bm)$ , i.e., espaço linear!
- Óptima? Não

# Procura de profundidade limitada

= procura em profundidade primeiro com limite de profundidade  $l$ , i.e., nós à profundidade  $l$  não têm sucessores

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns 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  
  else  
    cutoff_occurred? ← 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 then cutoff_occurred? ← true  
      else if result ≠ failure then return result  
  if cutoff_occurred? then return cutoff else return failure
```

# Procura por aprofundamento progressivo

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

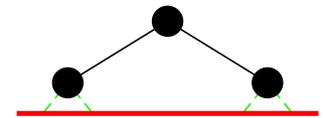
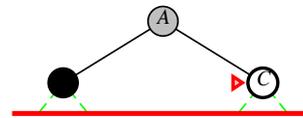
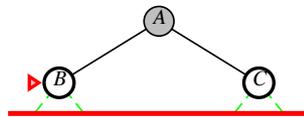
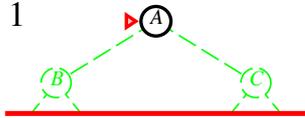
# Aprofundamento progressivo / $\neq 0$

Limit = 0



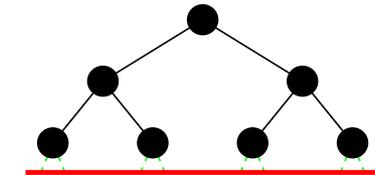
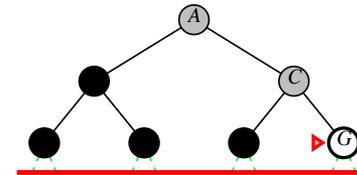
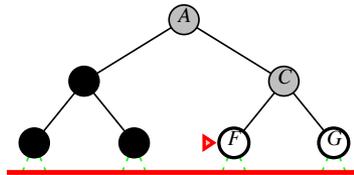
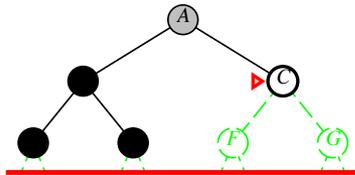
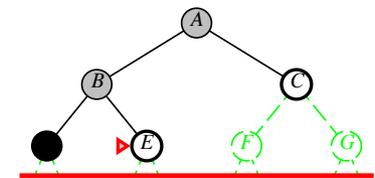
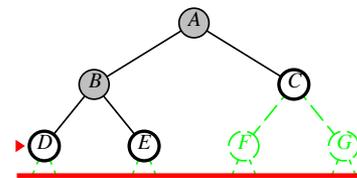
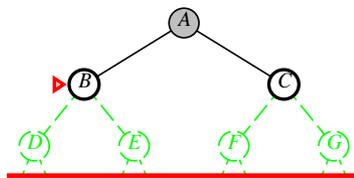
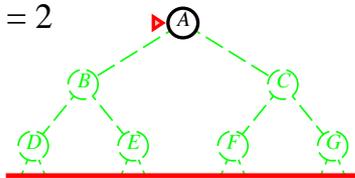
# Aprofundamento progressivo $l = 1$

Limit = 1



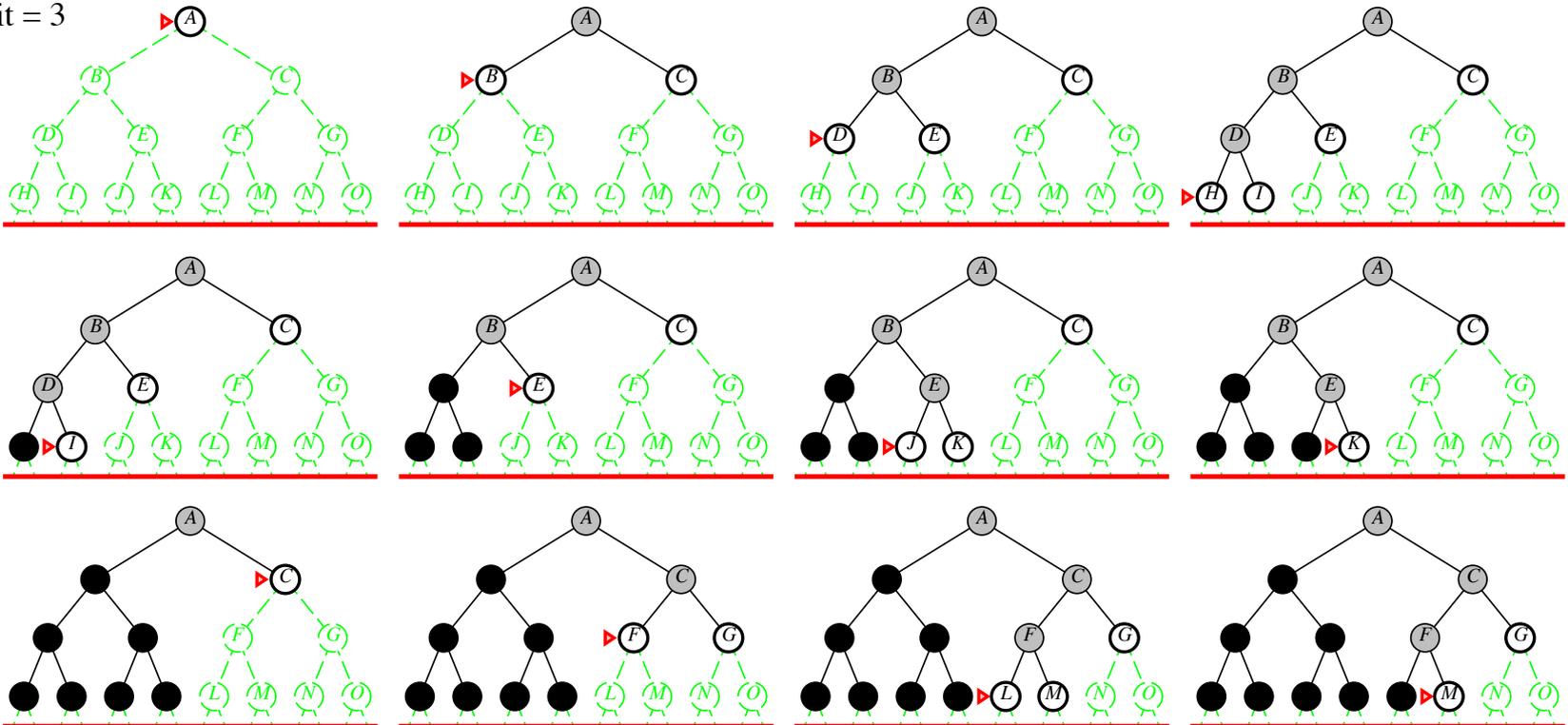
# Aprofundamento progressivo $l=2$

Limit = 2



# Aprofundamento progressivo / =3

Limit = 3



# Procura por aprofundamento progressivo

- Completa? Sim

- Tempo?

$$(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$$

*NB: Estamos a contabilizar nós gerados!*

- Espaço?  $O(bd)$

- Óptima? Sim, se custos constantes

# Comparação estratégias (nós gerados)

- Número de nós gerados com profundidade limitada  $d$  e factor ramificação  $b$ :

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

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

- Número de nós gerados em aprofundamento progressivo com profundidade  $d$  e factor ramificação  $b$ :

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

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

- **Largura Primeiro:**  $N_{BFS} = 1 + 10 + 100 + 1.000 + 10.000 + (100.000 - 10) = 111.101$
- **Profundidade Limitada:**  $N_{DLS} = 1 + 10 + 100 + 1.000 + 10.000 + 100.000 = 111.111$
- **Aprofundamento Progressivo:**  $N_{IDS} = 6 + 50 + 400 + 3.000 + 20.000 + 100.000 = 123.456$

- Sobrecarga IDS/DLS e IDS/BFS =  $(123.456 - 111.111)/111.111 = 11\% \approx b/(b-1)$

# Sumário de procura em árvore

Critério	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirecional (se aplicável)
Completa ?	Sim <sup>a</sup>	Sim <sup>a,b</sup>	Não	Não	Sim <sup>a</sup>	Sim <sup>a,d</sup>
Tempo	$O(b^d)$	$O(b^{1+\text{ceil}(C^*/\epsilon)})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Espaço	$O(b^d)$	$O(b^{1+\text{ceil}(C^*/\epsilon)})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Óptima?	Sim <sup>c</sup>	Sim	Não	Não	Sim <sup>c</sup>	Sim <sup>c,d</sup>

<sup>a</sup> completa se o factor de ramificação for finito

<sup>b</sup> completa se custos do passo  $\geq \epsilon$  para  $\epsilon$  positivo (não nulo!)

<sup>c</sup> óptima se custo do caminho for monótono na profundidade da árvore (e.g. custos constantes e idênticos)

<sup>d</sup> se ambas as direções utilizarem procura em largura primeiro

# Optimalidade de procura em grafos

O algoritmo de procura em grafos ignora novos caminhos para o mesmo estado, portanto a questão da optimalidade não é imediata

- Para grafos com custos de passos constantes, quer a procura em largura primeiro quer a procura de custo uniforme com conjunto de nós explorados garantem a solução óptima.
- Se a inserção na fronteira adicionar apenas nós correspondentes a estados **não expandidos** e mantiver o nó com o menor custo total na fronteira, então a procura de custo uniforme com lista fechada é óptima (ver atrás). Muito semelhante ao algoritmo de Dijkstra para encontrar o melhor caminho num grafo dirigido.

# Complexidade de procura em grafos

- A procura em profundidade primeiro torna-se completa para espaços finitos.
- Claramente, no pior caso, a complexidade espacial para qualquer dos algoritmos básicos de procura passa a ser da ordem  $b^{d+1}$  quando se utiliza o conjunto de nós explorados.
- Recorrendo ao conjunto de nós explorados, a complexidade dos algoritmos de procura é limitada pelo **número de estados** no espaço de procura e não pelo número de caminhos nesse espaço. Para alguns problemas, pode resultar em diminuições exponenciais em tempo e espaço.
- Contudo, em espaços de procura muito grandes pode continuar a ser proibitivo.

# Implementação dos algoritmos

- O conjunto de estados explorados é habitualmente implementado com uma tabela de dispersão (**hash table**).
- Quanto à **fronteira**, normalmente opta-se por:
  - Quando o **grafo de estados é esparso** (número reduzido de nós sucessores limitados por uma constante pequena), opta-se por uma **fila de prioridade (priority queue)**. Complexidade temporal  $O(N * \log_2 N + L * \log_2 N)$ , em que  $N$  o número de estados e  $L$  o número de arcos. Esta é a situação habitual:
    - No pior caso têm de se retirar  $N$  nós da fila de prioridade, cada uma destas operações da ordem de  $\log_2 N$
    - São necessárias no pior caso  $L$  inserções na fila de prioridade, cada uma com custo  $\log_2 N$ .
  - Quando o **grafo de estados é denso**, então deve-se utilizar uma lista ou tabela de dispersão (hash table). Complexidade temporal da ordem de  $O(N^2 + L)$ 
    - Retirar o nó com menor custo é operação  $O(N)$ , no máximo  $N$  vezes.
    - A inserção de um nó sucessor na fronteira pode ser feita com uma operação de  $O(1)$

# Comparação implementações

N	Densidade	L	$N \cdot \log N + L \cdot \log N$	$N \cdot N + L$	Rácio
10	1%	1	37	101	0,36
10	10%	10	66	110	0,60
10	50%	50	199	150	1,33
10	90%	90	332	190	1,75
10	100%	100	365	200	1,83
100	1%	100	1329	10100	0,13
100	10%	1000	7308	11000	0,66
100	50%	5000	33884	15000	2,26
100	90%	9000	60459	19000	3,18
100	100%	10000	67103	20000	3,36
1000	1%	10000	109624	1010000	0,11
1000	10%	100000	1006544	1100000	0,92
1000	50%	500000	4992858	1500000	3,33
1000	90%	900000	8979172	1900000	4,73
1000	100%	1000000	9975750	2000000	4,99
10000	1%	1000000	13420590	101000000	0,13
10000	10%	10000000	133010001	110000000	1,21
10000	50%	50000000	664518496	150000000	4,43
10000	90%	90000000	1196026991	190000000	6,29
10000	100%	100000000	1328904115	200000000	6,64
100000	1%	100000000	1662625011	10100000000	0,16
100000	10%	1000000000	16611301438	11000000000	1,51
100000	50%	5000000000	83049863336	15000000000	5,54
100000	90%	9000000000	149488425234	19000000000	7,87
100000	100%	10000000000	166098065708	20000000000	8,30

# Sumário

- A formulação do problema normalmente requer uma abstração dos detalhes da realidade para definir um espaço de estados que possa ser explorado
- Variedade de estratégias de procura cega
- Procura de aprofundamento progressivo usa apenas espaço linear e da mesma ordem de grandeza temporal do que outros algoritmos de procura cega. É o algoritmo de escolha para procura cega.
- Procura bidirecional pode reduzir enormemente a complexidade temporal, mas nem sempre é aplicável e requer espaço exponencial.