



**UNIVERSIDADE EDUARDO MONDLANE**  
**FACULDADE DE ENGENHARIA**  
**DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA**

# **Inteligência Artificial**

Resolução de Problemas Por Meio de Busca  
Busca Não Informada

**Docentes:** Eng Roxan Cadir  
Eng Ruben Manhiça

**Maputo, 28 de agosto de 2024**



# **Conteúdo da Aula**

1. Resolução de Problemas Por meio de Buscas
2. Algoritmos de busca não Informada





# Resolução de problemas por meio de busca

Capítulo 3 – Russell & Norvig  
Seções 3.4 e 3.5





# Formulação de problemas

Um **problema** é definido por quatro itens:

1. **Estado inicial** ex., “em Arad”
  2. **Acções** ou **função sucessor**  $S(x)$  = conjunto de pares ação-estado
    - ex.,  $S(Arad) = \{ \langle Arad \rightarrow Zerind, Zerind \rangle, \dots \}$
  3. **Teste de objetivo**, pode ser
    - **explícito**, ex.,  $x = \text{“em Bucareste”}$
    - **implícito**, ex.,  $Cheque-mate(x)$
    -
  4. **Custo de caminho** (aditivo)
    - ex., soma das distâncias, número de ações executadas, etc.
    - $c(x, a, y)$  é o **custo do passo**, que deve ser sempre  $\geq 0$
- Uma **solução** é uma sequência de acções que levam do estado inicial para o estado objetivo.
  - Uma **solução ótima** é uma solução com o menor custo de caminho.





# Algoritmo geral de busca em árvore

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure  
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)  
  loop do  
    if fringe is empty then return failure  
    node  $\leftarrow$  REMOVE-FRONT(fringe)  
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)  
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

---

```
function EXPAND(node, problem) returns a set of nodes  
  successors  $\leftarrow$  the empty set  
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do  
    s  $\leftarrow$  a new NODE  
    PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result  
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)  
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1  
    add s to successors  
  return successors
```





# Estratégias de Busca

## Sem Informação (ou Busca Cega)

- Estratégias de busca **sem informação** usam apenas a informação disponível na definição do problema.
  - Apenas geram sucessores e verificam se o estado objetivo foi atingido.
- As estratégias de busca sem informação se distinguem pela **ordem** em que os nós são expandidos.
  - Busca em extensão ou largura (*Breadth-first*)
  - Busca de custo uniforme
  - Busca em profundidade (*Depth-first*)
  - Busca em profundidade limitada
  - Busca de aprofundamento iterativo





# Estratégias de busca

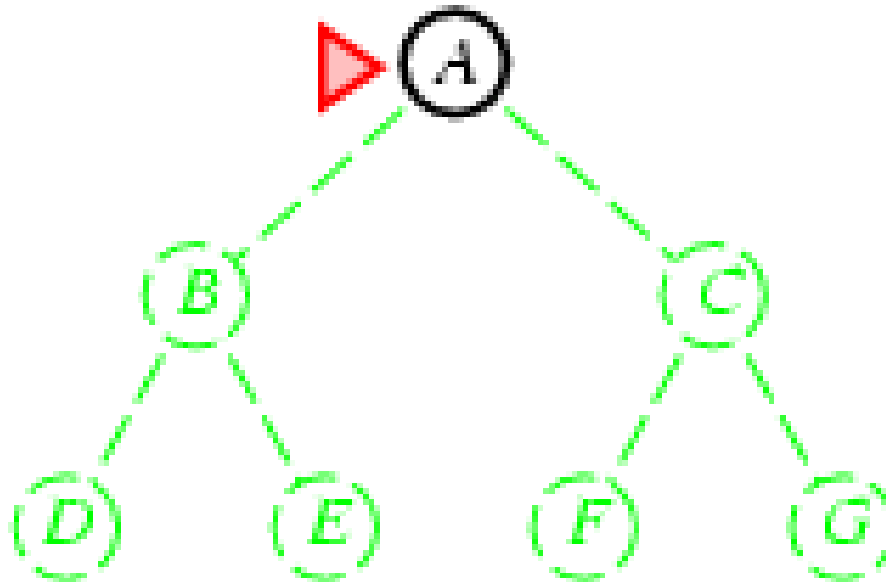
- Estratégias são avaliadas de acordo com os seguintes critérios:
  - **completeza**: o algoritmo sempre encontra a solução se ela existe?
  - **complexidade de tempo**: número de nós gerados
  - **complexidade de espaço**: número máximo de nós na memória
  - **otimização**: a estratégia encontra a solução ótima?
- Complexidade de tempo e espaço são medidas em termos de:
  - *b*: máximo fator de ramificação da árvore (número máximo de sucessores de qualquer nó)
  - *d*: profundidade do nó objetivo menos profundo
  - *m*: o comprimento máximo de qualquer caminho no espaço de estados (pode ser  $\infty$ )





# Busca em extensão ou largura (BFS)

- Expandir o nó não-expandido mais perto da raiz.
- **Implementação:**
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.

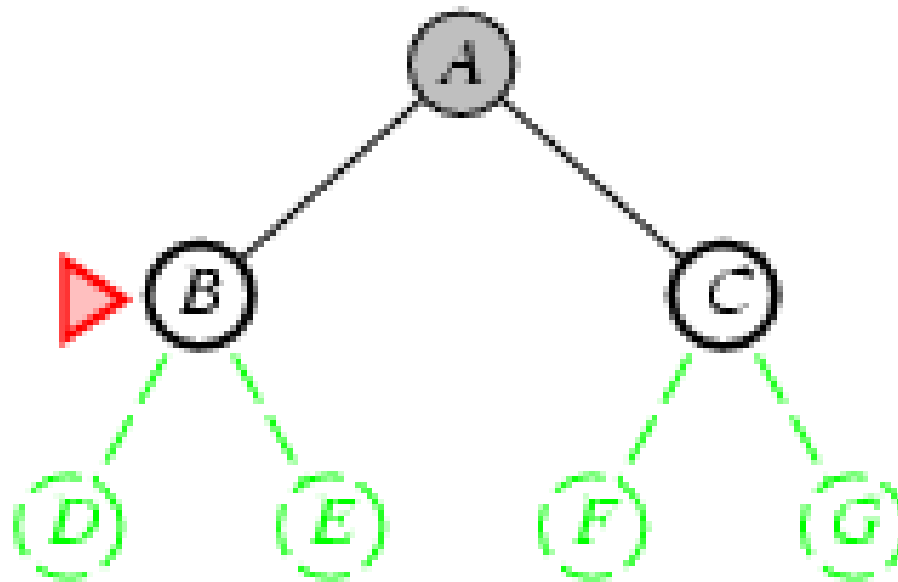






# Busca em extensão

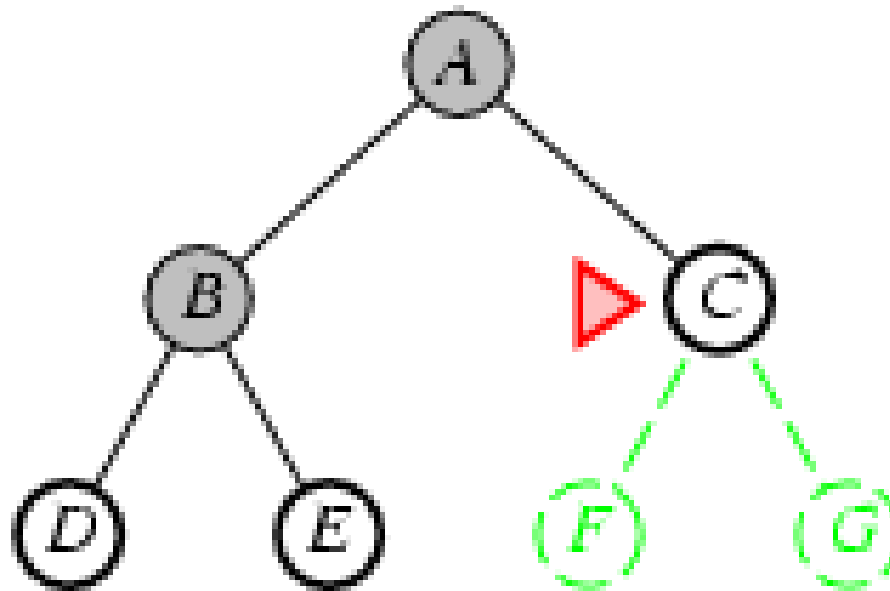
- Expandir o nó não-expandido mais perto da raiz.
- **Implementação:**
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.





# Busca em extensão

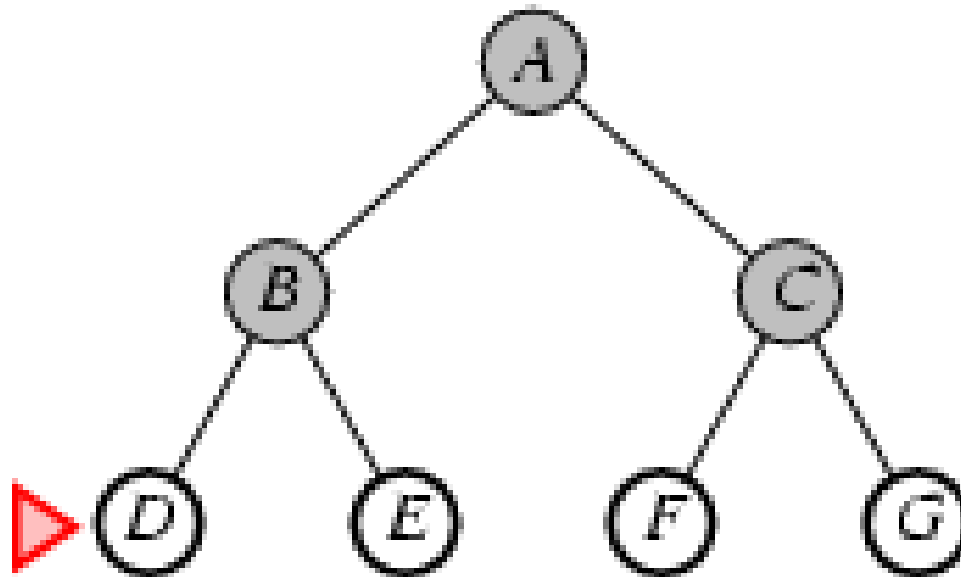
- Expandir o nó não-expandido mais perto da raiz.
- **Implementação:**
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.





# Busca em extensão

- Expandir o nó não-expandido mais perto da raiz.
- **Implementação:**
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.





# Propriedades da busca em extensão

- 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 na memória)
- Ótima? Sim (se todas as acções tiverem o mesmo custo)





# Requisitos de Tempo e Memória para a Busca em Extensão

- Busca com fator de ramificação  $b=10$ .
- Supondo que 10.000 nós possam ser gerados por segundo e que um nó exige 1KB de espaço.

Profundidade	Nós	Tempo	Memória
2	1100	0.11 segundo	1 megabyte
4	111.100	11 segundos	106 megabytes
6	$10^7$	19 minutos	10 gigabytes
8	$10^9$	31 horas	1 terabyte
10	$10^{11}$	129 dias	101 terabytes
12	$10^{13}$	35 anos	10 petabytes
14	$10^{15}$	3.523 anos	1 exabyte





# Busca de custo uniforme

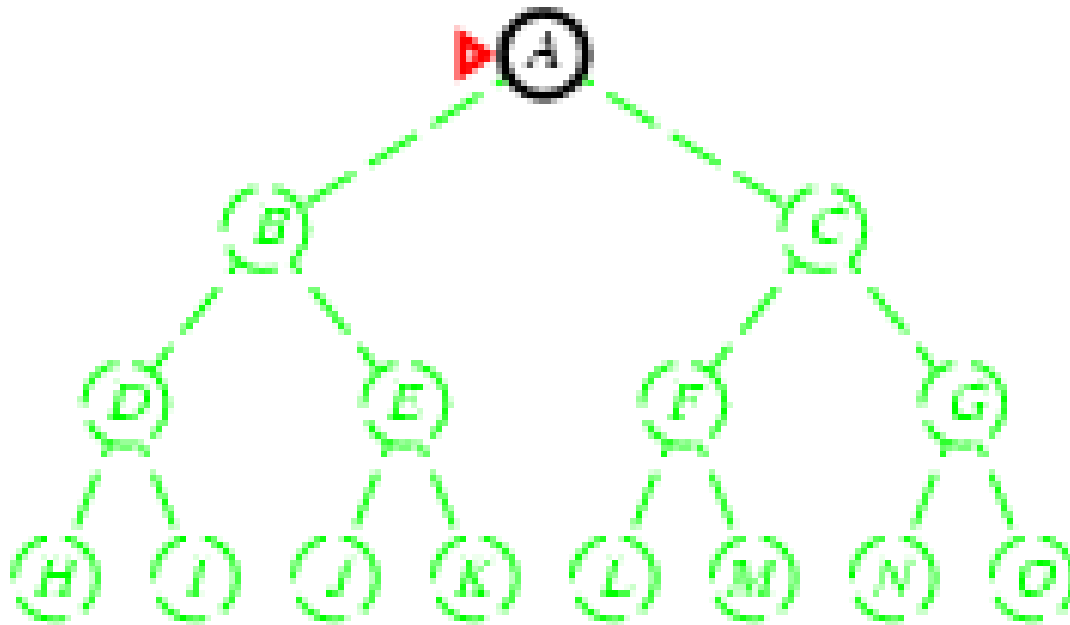
- Expande o nó não-expandido que tenha o caminho de custo mais baixo.
- **Implementação:**
  - *borda* = fila ordenada pelo custo do caminho
- Equivalente a busca em extensão (largura) se os custos são todos iguais
- Completa? Sim, se o custo de cada passo  $\geq \epsilon$
- Tempo? numero de nós com  $g \leq$  custo da solução ótima,  $O(b^{\lceil C^*/\epsilon \rceil})$  onde  $C^*$  é o custo da solução ótima
- Espaço? de nós com  $g \leq$  custo da solução ótima,  $O(b^{\lceil C^*/\epsilon \rceil})$
- Ótima? Sim pois os nós são expandidos em ordem crescente de custo total.





# Busca em Profundidade

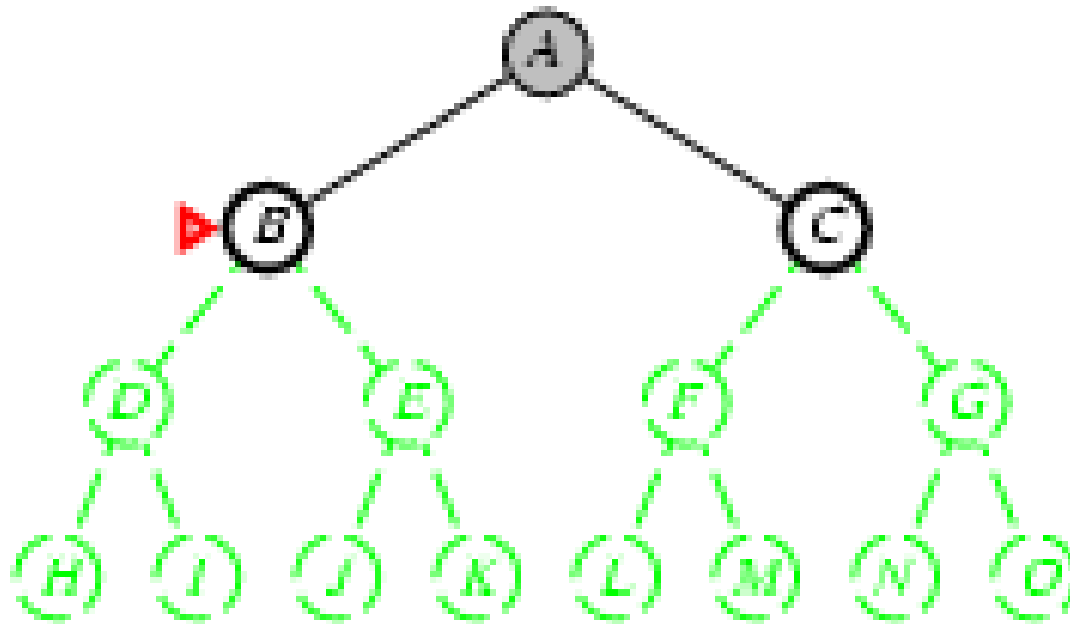
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

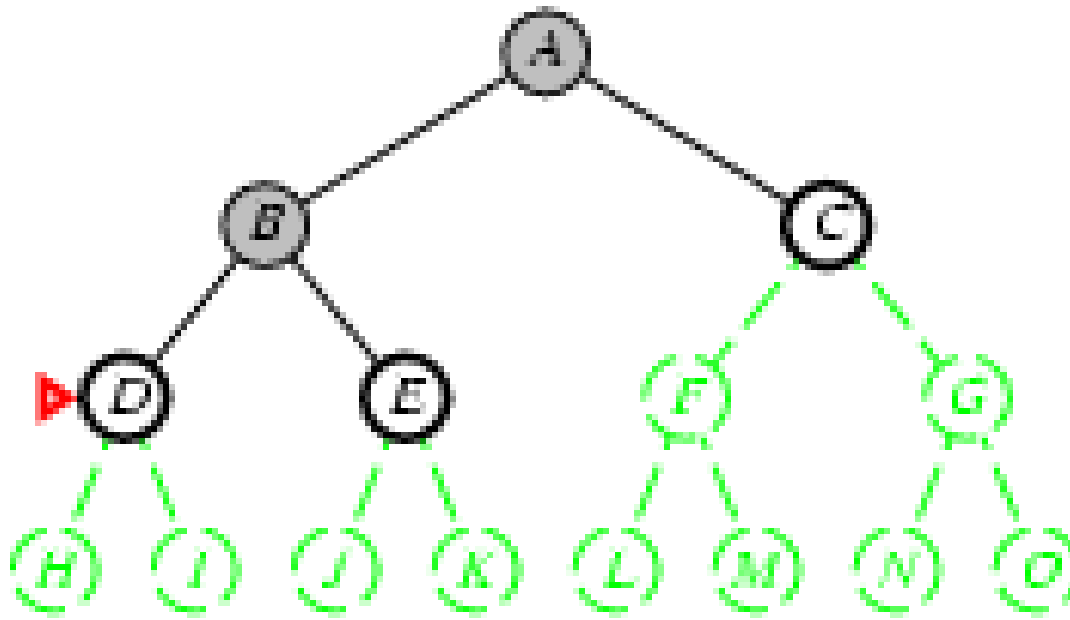






# Busca em Profundidade

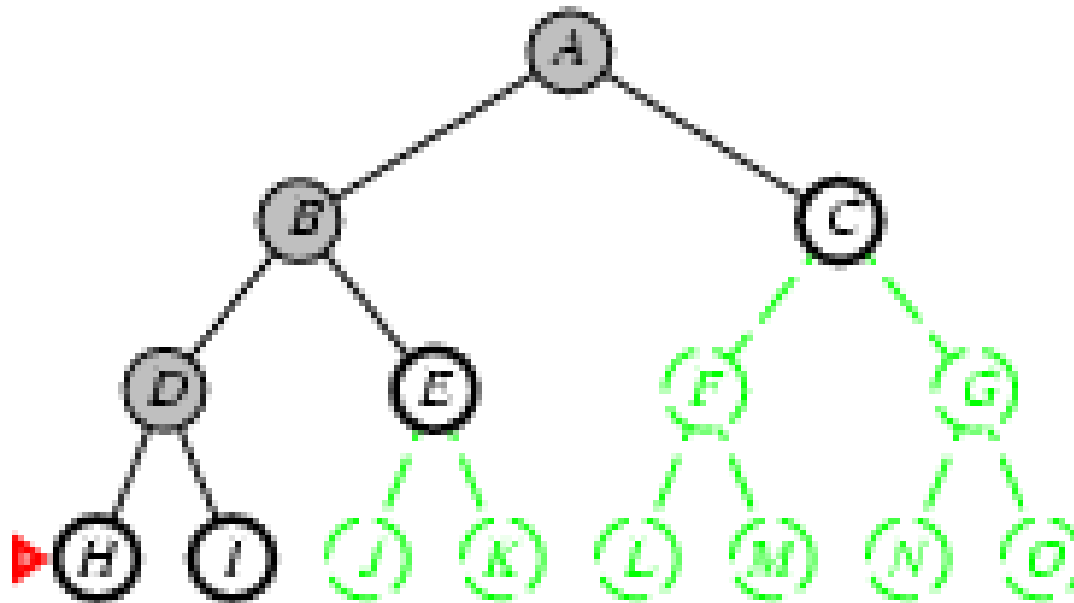
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

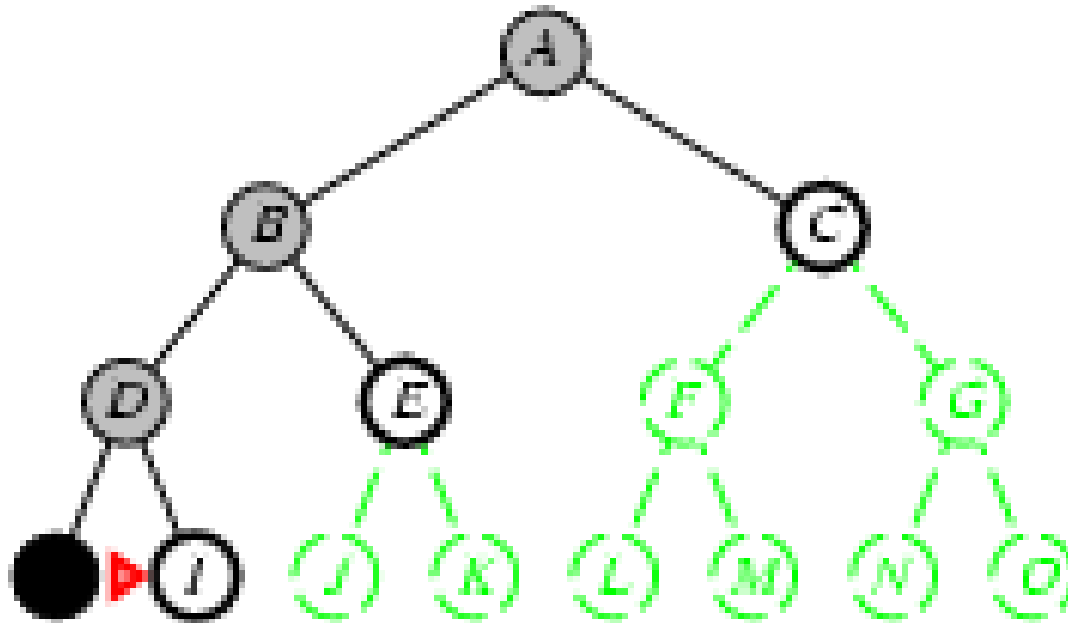
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

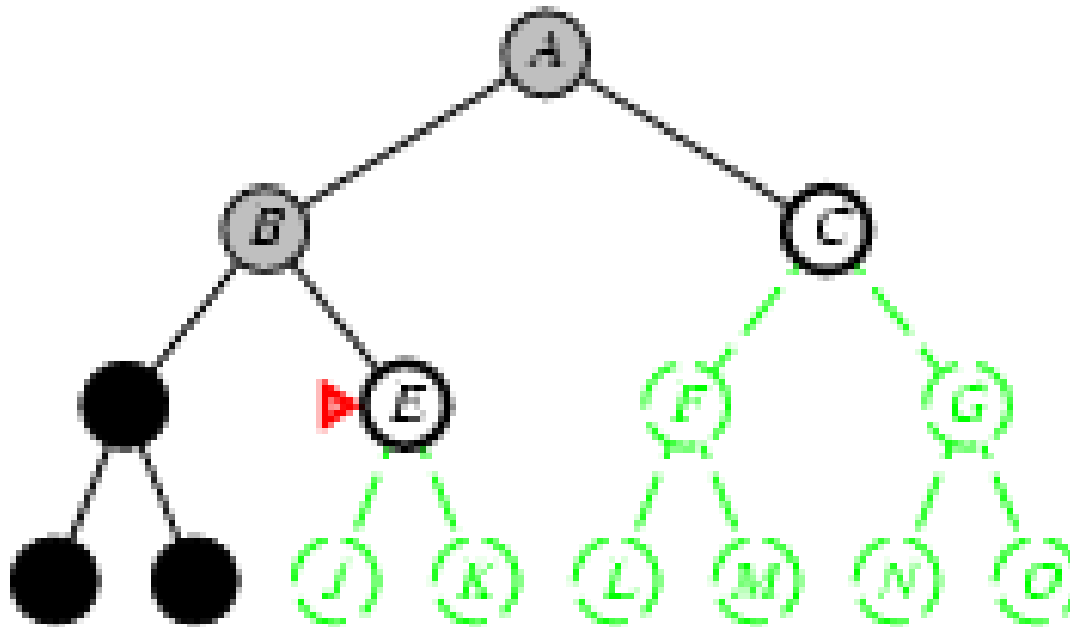
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

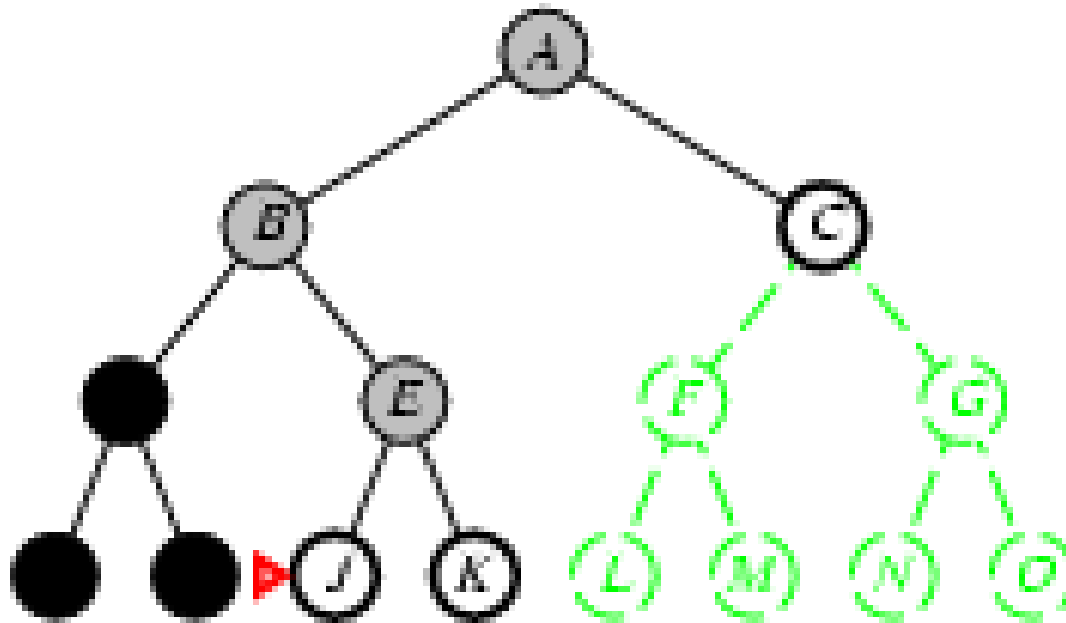
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

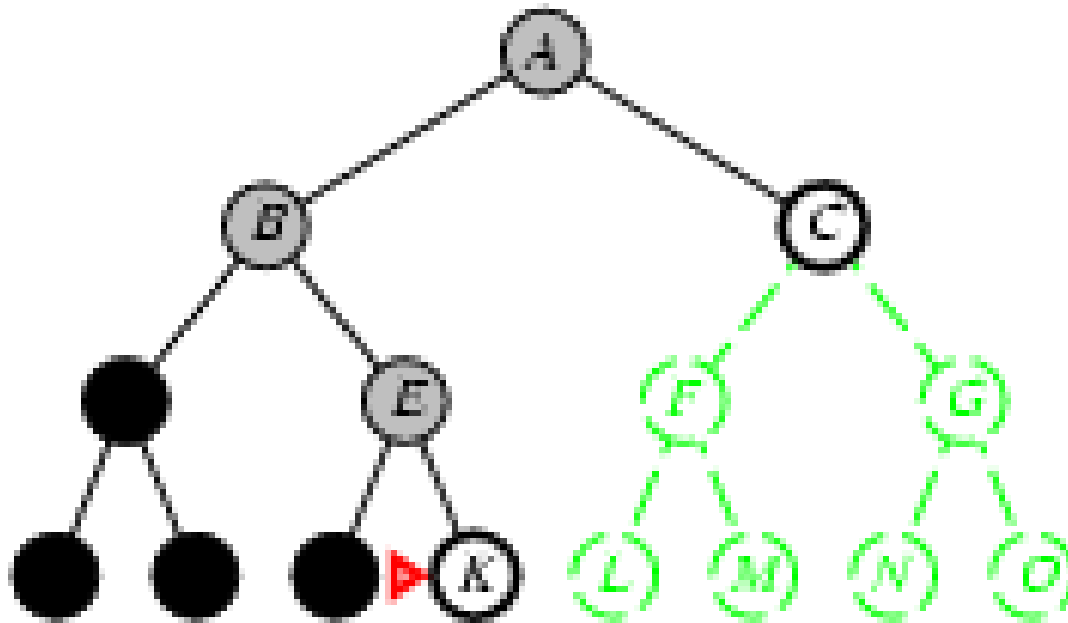
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

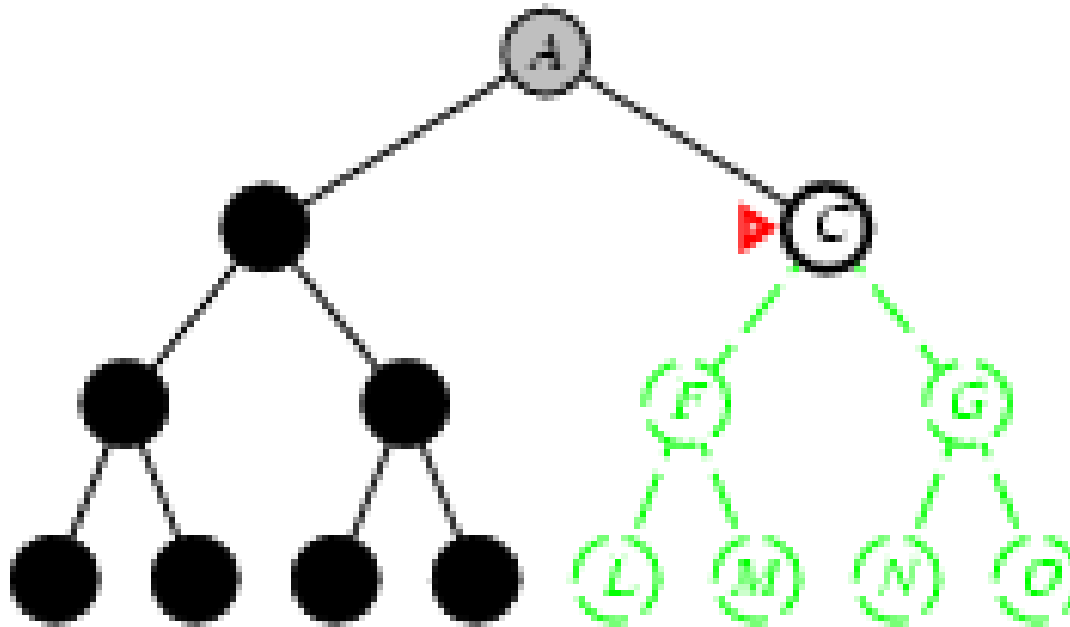
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

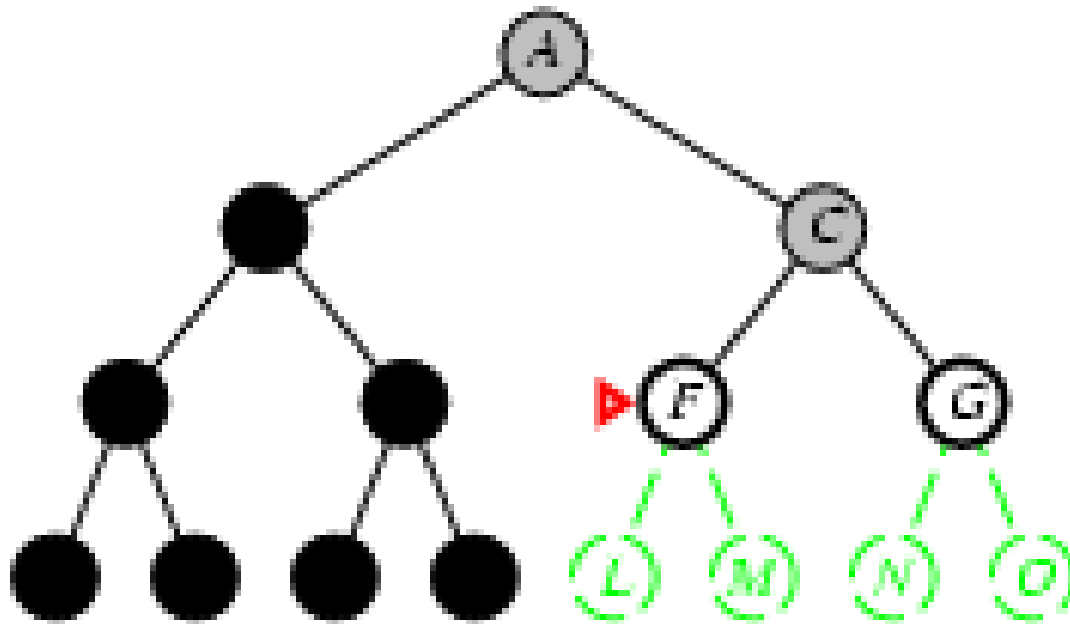
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

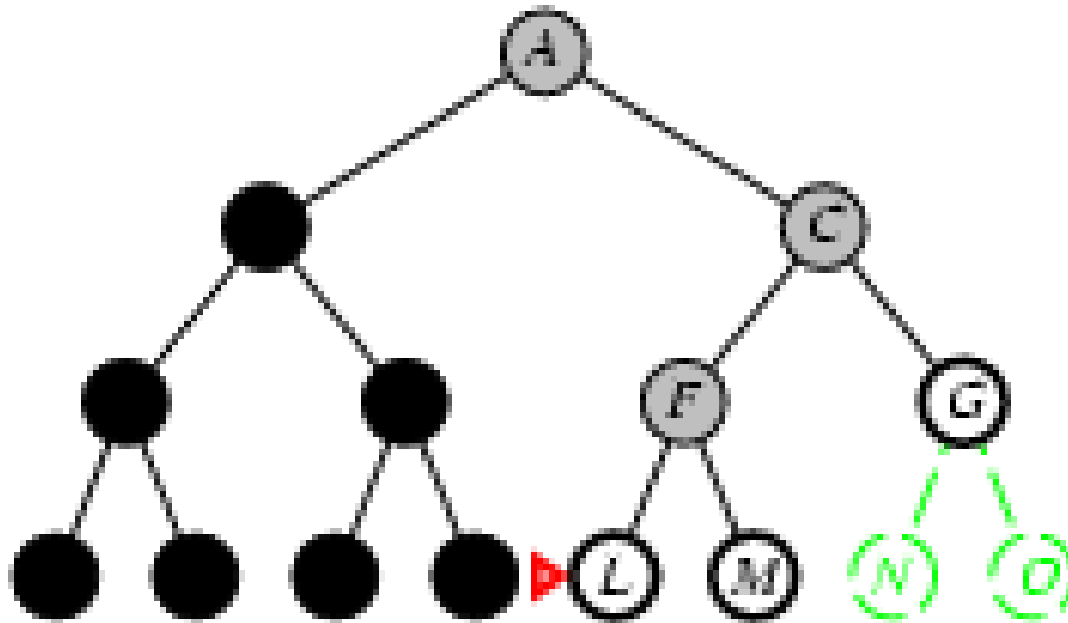






# Busca em Profundidade

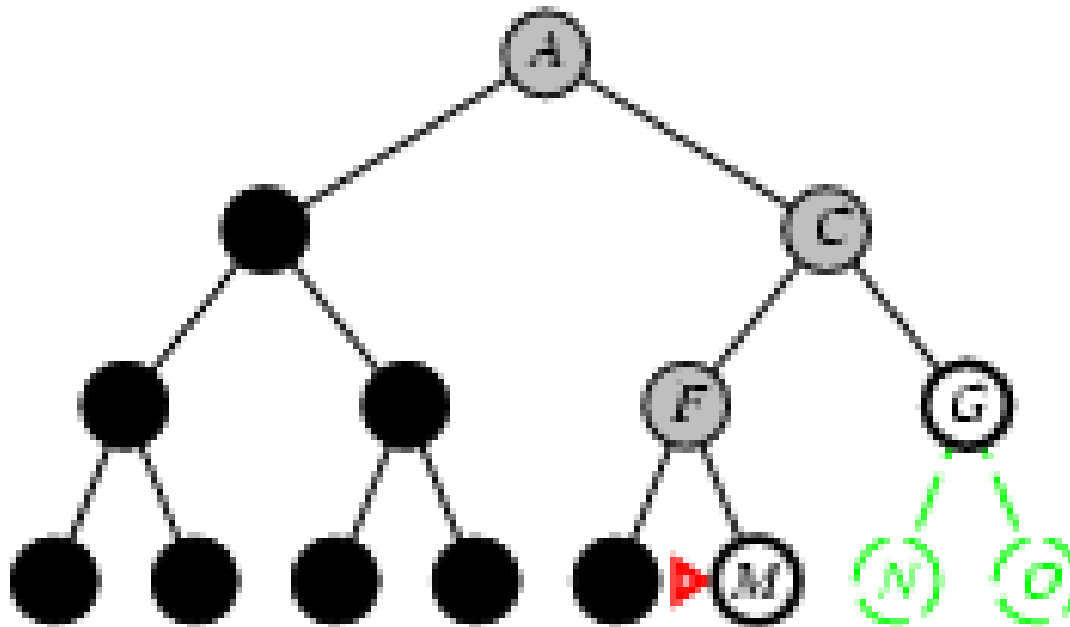
- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- **Implementação:**
  - *borda* = fila LIFO (*last-in, first-out*) = pilha





# Propriedades da Busca em Profundidade

- Completa? Não: falha em espaços com profundidade infinita, espaços com loops
  - Se modificada para evitar estados repetidos é completa para espaços finitos
- Tempo?  $O(b^m)$ : péssimo quando  $m$  é muito maior que  $d$ .
  - mas se há muitas soluções pode ser mais eficiente que a busca em extensão
- Espaço?  $O(bm)$ , i.e., espaço linear!
  - 118 kilobytes ao invés de 10 petabytes para busca com  $b=10$ ,  $d=m=12$
- Ótima? Não





# Busca em Profundidade Limitada

= busca em profundidade com limite de profundidade  $l$ , isto é, nós com profundidade  $l$  não tem sucessores

- Implementação Recursiva:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred?  $\leftarrow$  false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    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
```





# Propriedades da Busca em Profundidade Limitada

- Completa? Não; a solução pode estar além do limite.
- Tempo?  $O(b^l)$
- Espaço?  $O(bl)$
- Ótima? Não





# Busca de Aprofundamento Iterativo em Profundidade

```
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
```





# Busca de Aprofundamento Iterativo em Profundidade $\neq 0$

Limit = 0



# Busca de Aprofundamento Iterativo em Profundidade / =1

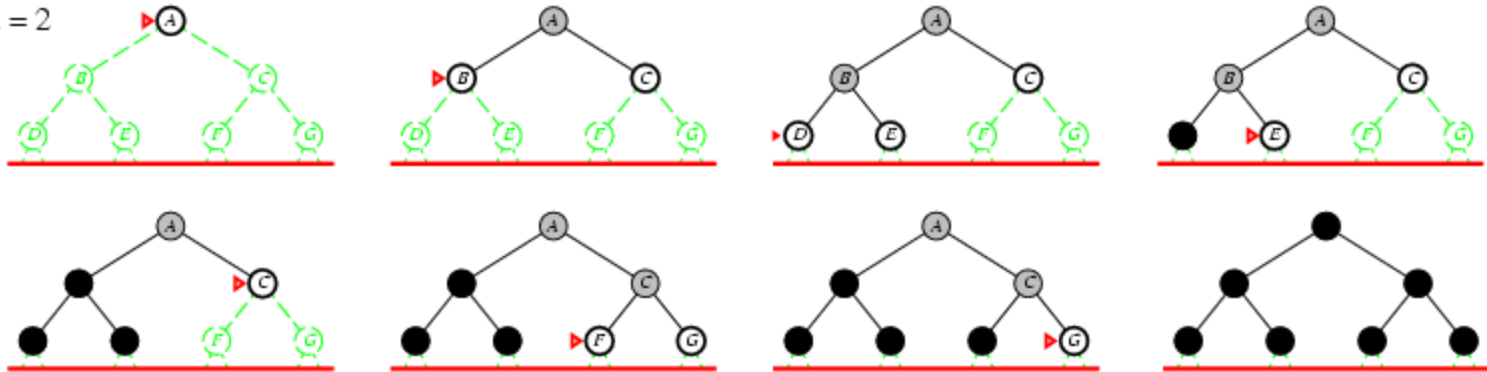
Limit = 1





# Busca de Aprofundamento Iterativo em Profundidade / =2

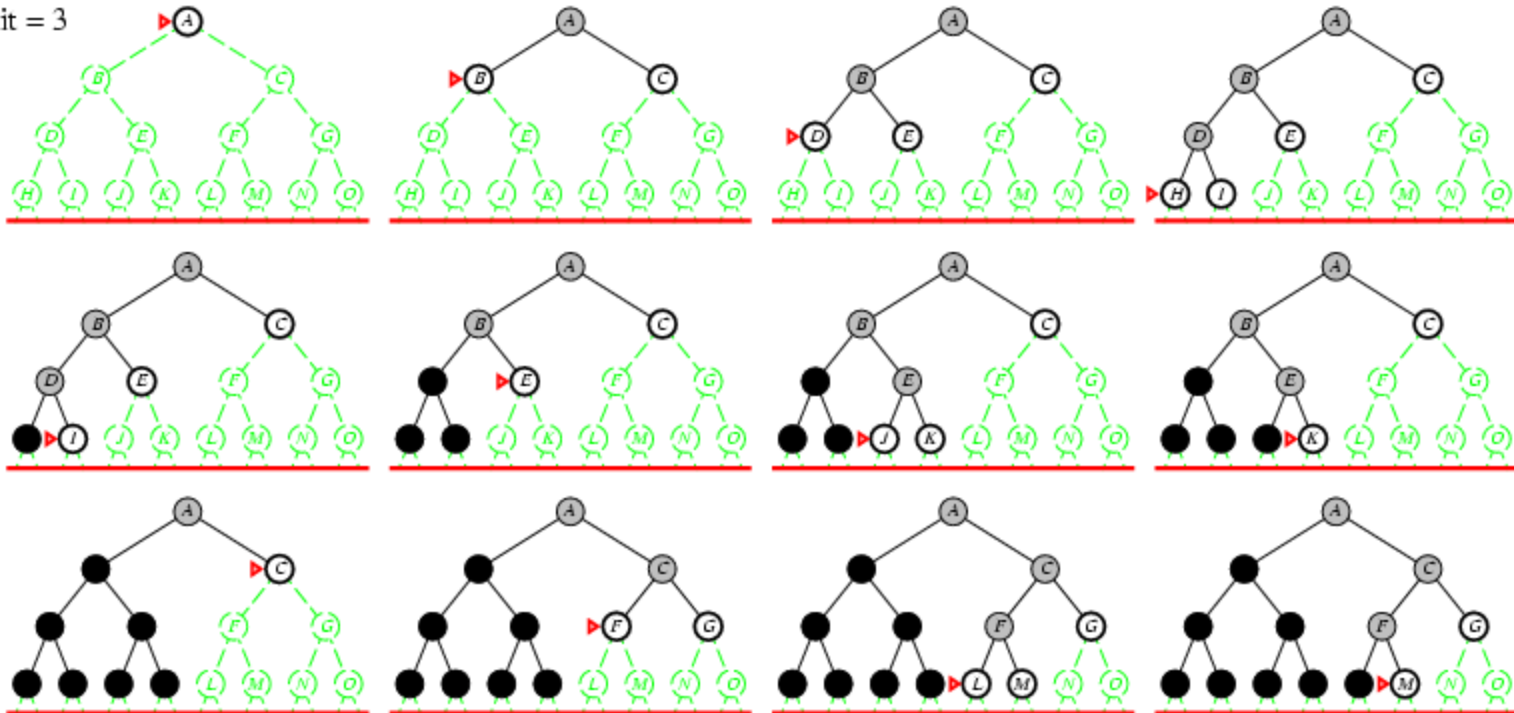
Limit = 2





# Busca de Aprofundamento Iterativo em Profundidade / =3

Limit = 3





# Propriedades da busca de aprofundamento iterativo

- Completa? Sim
- Tempo?  $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Espaço?  $O(bd)$
- Ótima? Sim, se custo de passo = 1





# Resumo dos algoritmos

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes





# Estados repetidos

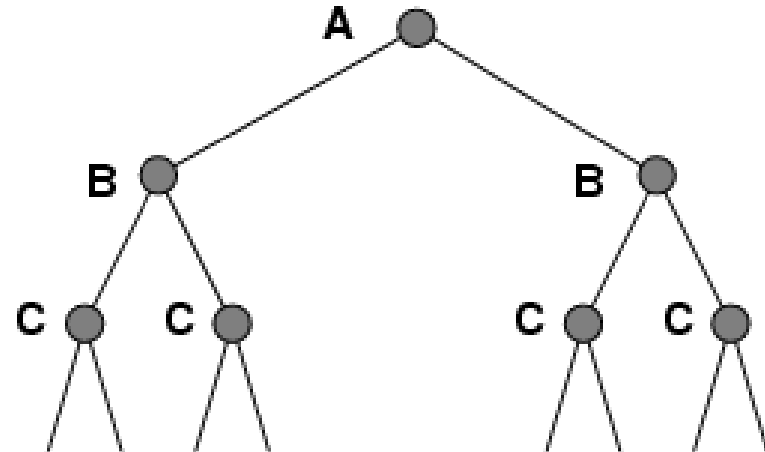
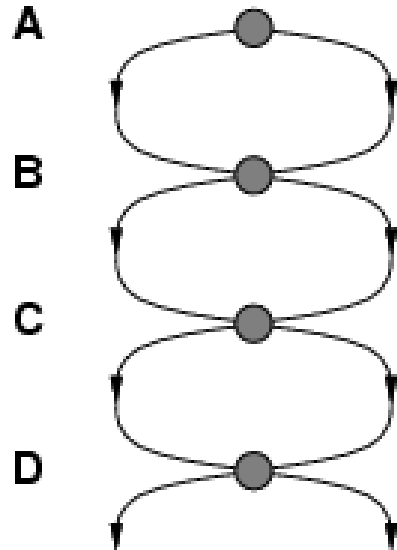
- O processo de busca pode perder tempo expandindo nós já explorados antes
  - Estados repetidos podem levar a loops infinitos
  - Estados repetidos podem transformar um problema linear em um problema exponencial





# Estados Repetidos

- Não detectar estados repetidos pode transformar um problema linear em um problema exponencial.





# Detecção de estados repetidos

- Comparar os nós prestes a serem expandidos com nós já visitados.
  - Se o nó já tiver sido visitado, será descartado.
  - Lista “closed” (fechado) armazena nós já visitados.
    - Busca em profundidade e busca de aprofundamento iterativo não tem mais espaço linear.
  - A busca percorre um grafo e não uma árvore.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```





# Resumo

- A formulação de problemas usualmente requer a abstração de detalhes do mundo real para que seja definido um espaço de estados que possa ser explorado através de algoritmos de busca.
- Há uma variedade de estratégias de busca sem informação (ou busca cega).
- A busca de aprofundamento iterativo usa somente espaço linear e não muito mais tempo que outros algoritmos sem informação.

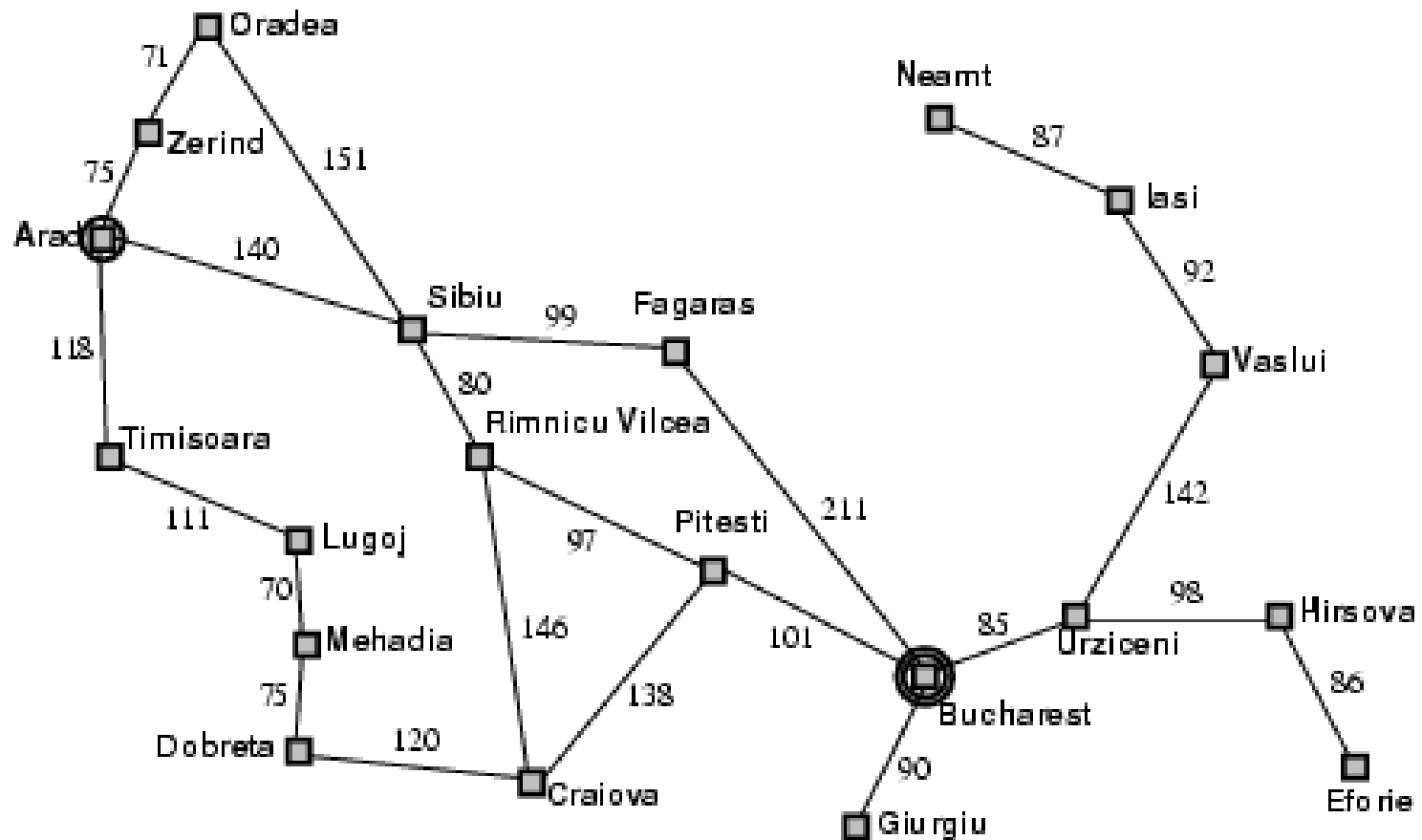






# TPC

- Aplicar busca de custo uniforme para encontrar o caminho mais curto entre Arad e Bucareste.



**FIM!!!**

Duvidas e Questões?

