BUSCA CEGA OU SEM INFORMAÇÃO

(parte 2 – Resolução de problemas por busca)

*Capítulo 3 (Russel & Norvig)

Tópicos

- Estratégias de busca sem informação
  - Busca em largura ou extensão
    - custo uniforme (menor custo)
  - Busca em profundidade
    - profundidade limitada
    - aprofundamento iterativo em profundidade

BUSCA SEM INFORMAÇÃO OU CEGA

- Utilizam somente a informação disponível na formulação do problema

- As estratégias de busca diferenciam-se apenas pela ordem em que expandem os nós da árvore de busca

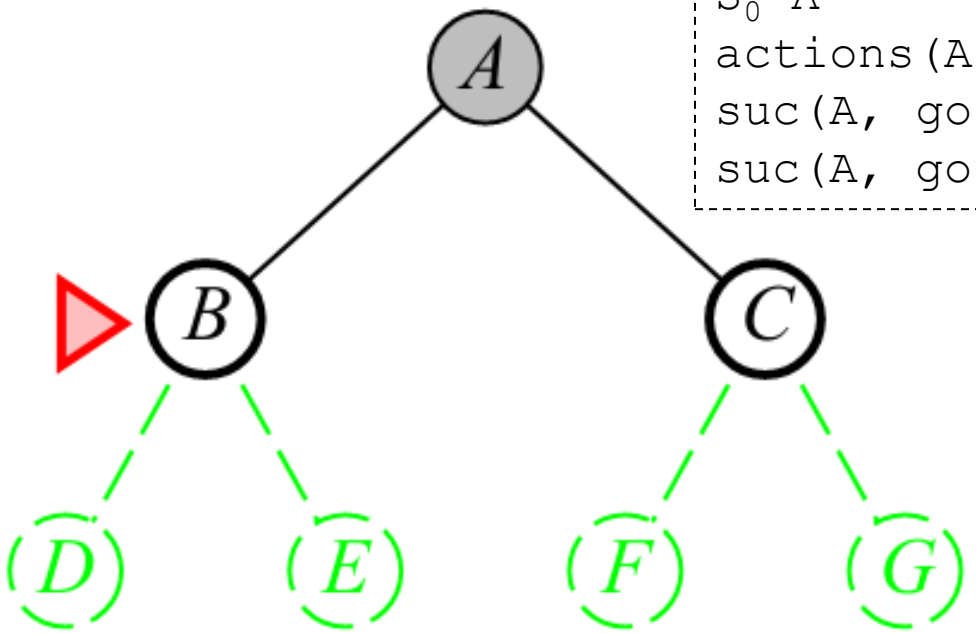
  – Estratégias que exploram primeiramente nós mais promissores com uso de informação extra-problema serão vistas mais tarde
    - busca informada ou heurística

Resolução de problemas por meio de buscas

BUSCA EM LARGURA/EXTENSÃO (BREADTH-FIRST)

BUSCA EM LARGURA

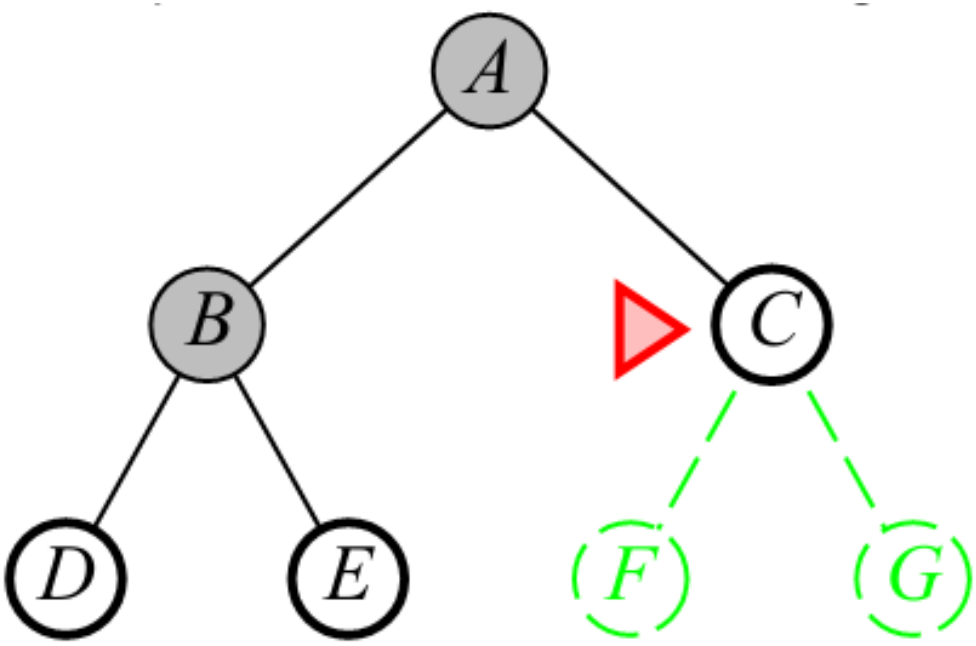
- Dentre os nós da fronteira, expande o **MAIS RASO**
  (sempre explora o caminho mais curto em qtd de ações antes)
- A fronteira é uma FILA (FIFO) – nós sucessores vão para o final da fila de fronteira – porém sempre pega o mais raso.




```
S0=A
actions(A)={go(B), go(C)}
suc(A, go(B)) -> B
suc(A, go(C)) -> C
```

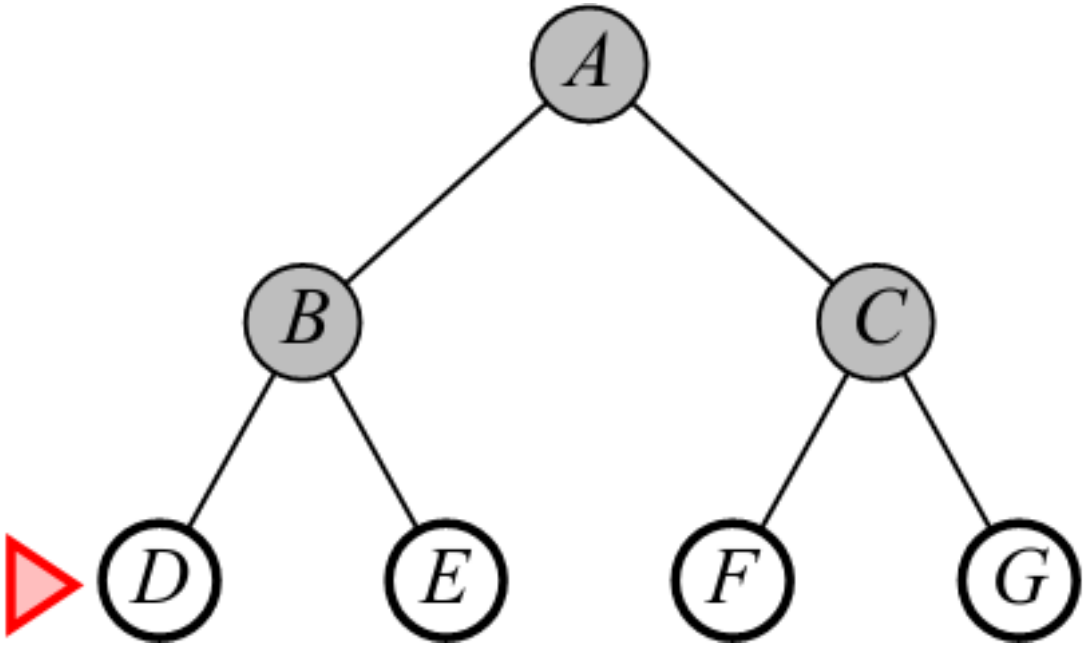
Explorados={A}
Fronteira=B<C

BUSCA EM LARGURA



Explorados={A, B}
Fronteira=C<D<E

BUSCA EM LARGURA



Explorados={A, B, C}
Fronteira=D<E<F<G

Busca em Largura

```
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
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Caso 2

Caso 1

estratégia

Cria nó que aponta ao pai

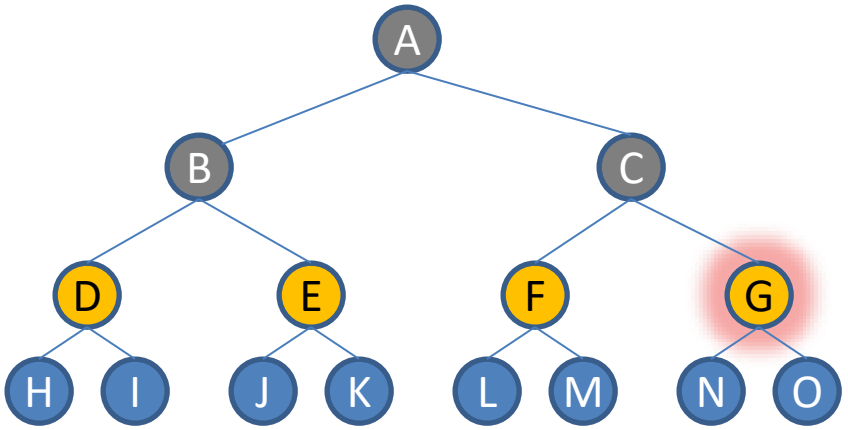
evita ciclos

Insere nó na árvore de busca

Busca em largura: análise

Critério	análise
Completo	sim, sse b for finito
Ótimo	sim, sse custos das ações são iguais
Complex. espaço	caso 1: $1+b+b^2+b^3+\ldots+b^d = O(b^d)$ caso 2: $O(b^{d+1})$
Complex. tempo	caso 1: $O(b^d)$ caso 2: $O(b^{d+1})$

Caso 1: goal test após a geração do nó filho
Caso 2: goal-test após a seleção de nó a expandir



b: número máximo de sucessores de um nó qualquer (neste caso, 2)

d: profundidade do nó objetivo mais raso (G, então d=2)

Complexidade espacial (caso 1)
nós na memória ao encontrar G:
explorados = ABC $1 + b^1$
fronteira = DEFG b^2
total = $1 + b^1 + \mathbf{b^2}$

Complexidade espacial (caso 2)
nós na memória ao encontrar G:
explorados = ABCDEFG $1 + b^1 + b^2$
fronteira = HIJKLM $b^3 - b$
total = $1 + b^1 + b^2 + \mathbf{b^3} - b$

BUSCA DE CUSTO UNIFORME (UNIFORM-COST OU CHEAPEST-FIRST SEARCH)

Resolução de problemas por meio de buscas

Custo uniforme ou 1º Menor custo

- Estende a BUSCA EM LARGURA:
  garante encontrar a solução ótima para qualquer valor de ação

- **Estratégia**
  Expandir nó com **menor custo de caminho g(*n*)**

- **Fronteira = fila ordenada pelo g(n)**

Custo uniforme ou 1º Menor custo

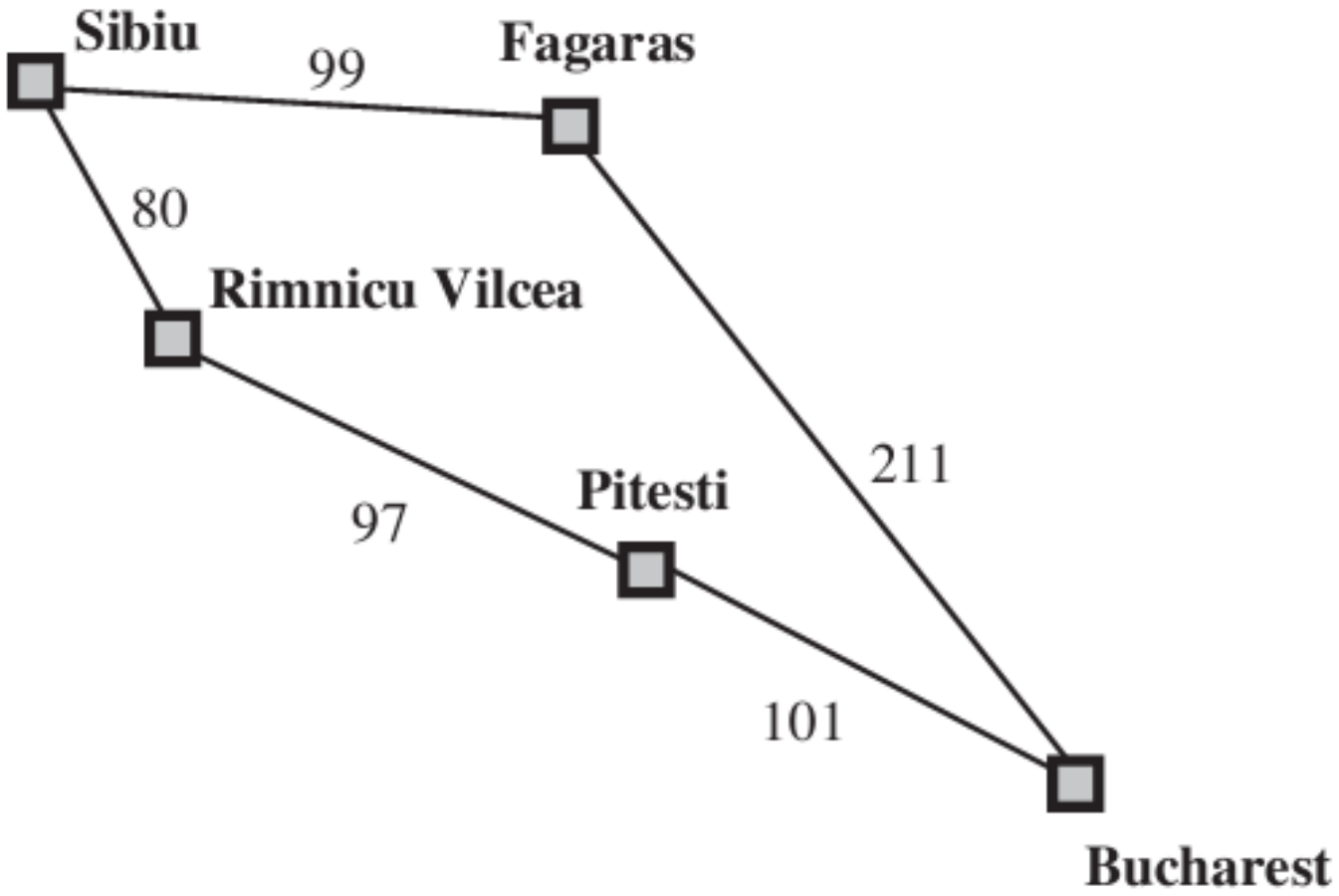
- Alterações em relação à busca em largura:
  - **Teste de objetivo:**
    quando um nó é selecionado para expansão (e não quando é criado)
    => em Pop(frontier)
  - **Se o *nó.estado* já está na fronteira,** verificar se o caminho encontrado é mais barato:
    - **Se for mais barato**:
      substituir o nó mais caro que está na fronteira pelo novo;
      reordene a fronteira
    - **Caso contrário** (custo **igual ou maior):**
      não incluir o nó na fronteira
      (dá preferência ao 1o. encontrado)

Custo Uniforme

```
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 ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node 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
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

CUSTO UNIFORME VS. LARGURA



Custo Uniforme: análise

Difícil avaliar em termos de b e d, porque expande nós em função do custo. Então, usamos C* que representa o custo da solução ótima

Critério	análise
Completo	sim, sse custo de cada ação > ε
Ótimo	sim, se expandir nós baseado em g(n)
Complex. espaço	$O(b^{[C^*/\varepsilon]+1})$
Complex. tempo	$O(b^{[C^*/\varepsilon]+1})$

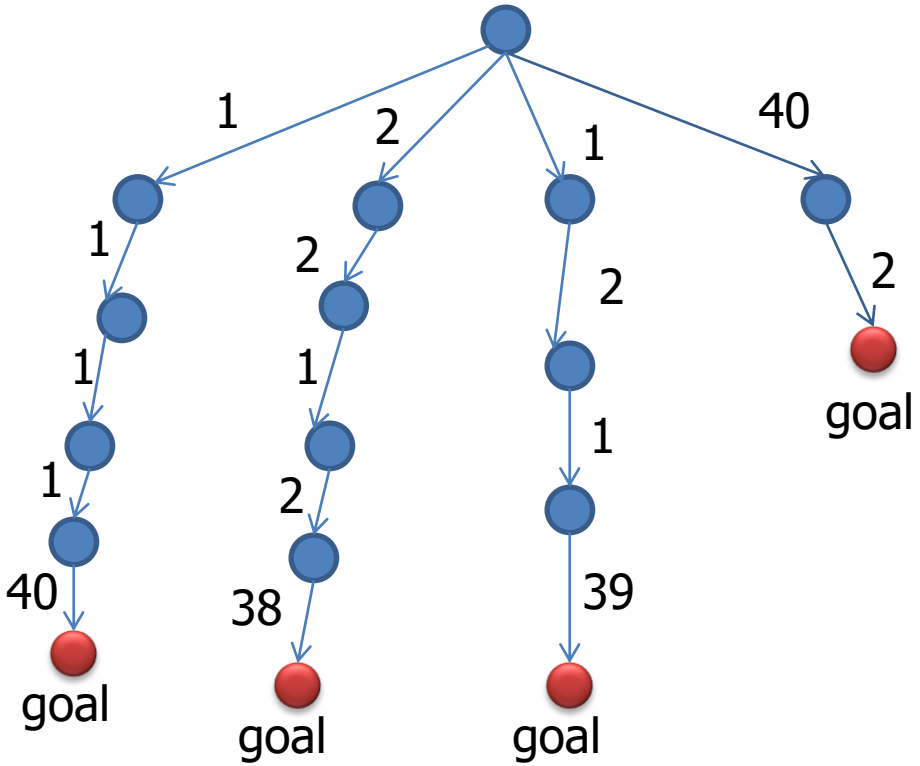
Por que [C*/ ε]?
é a profundidade *d* da solução ótima **no pior caso:** *custo da solução ótima/menor custo*

Se custos das ações são idênticos então tem-se
$C^*=d.\varepsilon$
$O(b^{[d.\ \varepsilon\ /\varepsilon]+1}) = O(b^{d+1})$

idem ao custo da busca em largura (caso 2)

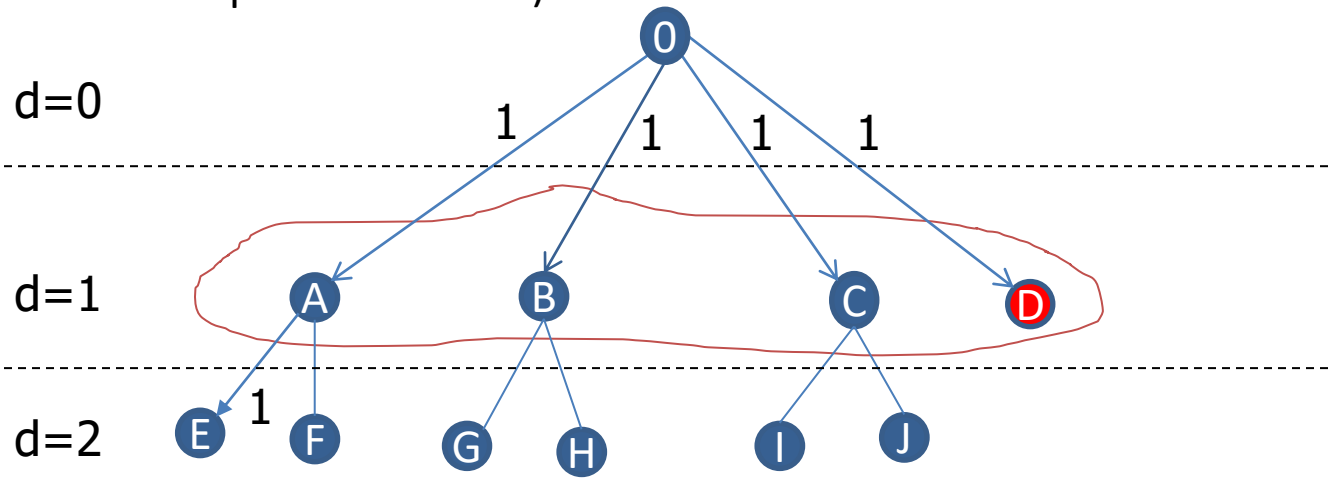
AVALIAÇÃO BUSCA CUSTO UNIFORME

A complexidade de tempo e espaço pode ser maior do que a da busca em largura, pois pode examinar grandes caminhos com pequenos passos antes de examinar caminhos com grandes passos que podem levar mais rapidamente a solução ótima.



AVALIAÇÃO BUSCA CUSTO UNIFORME

Se os custos das ações forem iguais, a busca por custo uniforme gastará mais tempo (e memória) que o breadh-first porque examina todos os nós que estão na profundidade do objetivo para verificar se há algum de menor custo (expande todos os nós da profundidade d)



Qualquer um dos nós na fronteira poderia ser o estado objetivo de menor custo: custo uniforme deve **expandir** todos antes de retornar a solução (D).
Breadh-first retorna D sem necessidade de expandir os nós da camada 2.

BUSCA EM PROFUNDIDADE (DEPTH-FIRST)

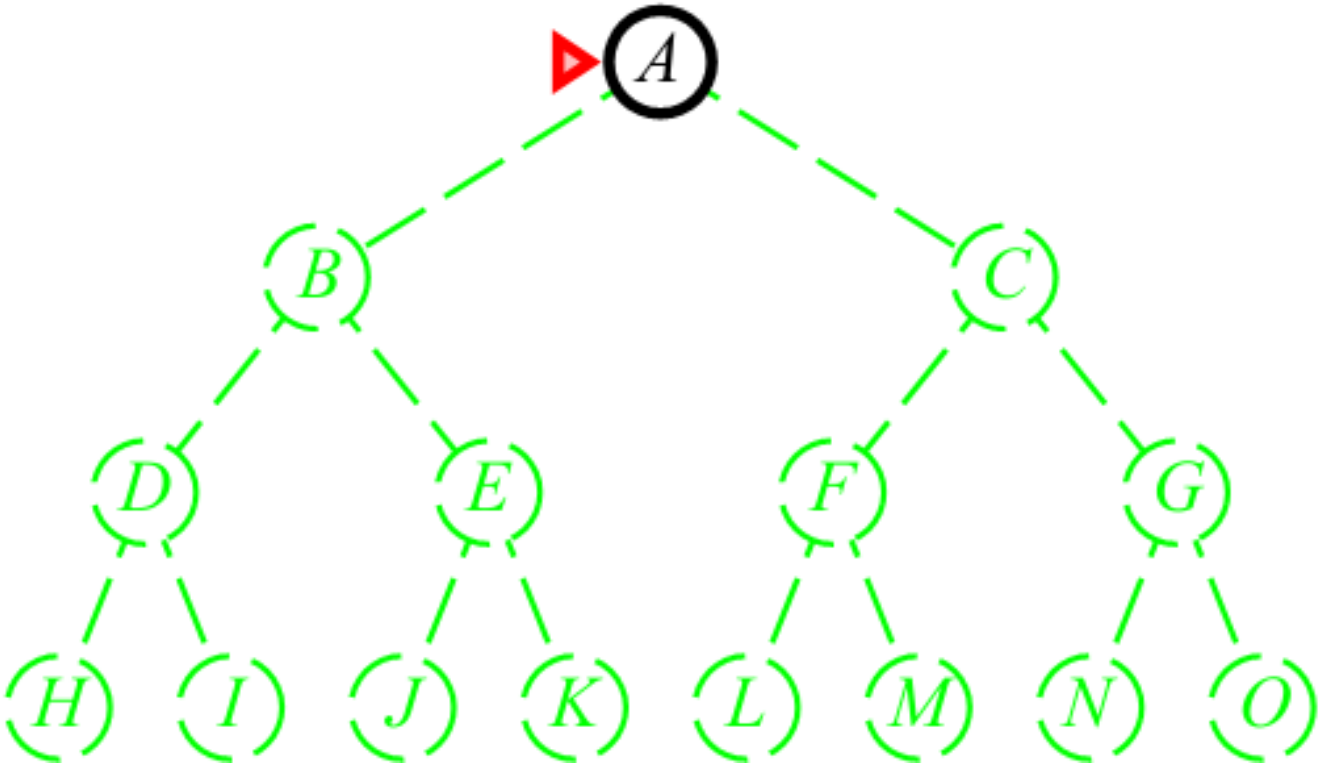
Resolução de problemas por meio de buscas

Busca em profundidade

- Expande o nó de **MAIOR PROFUNDIDADE** que esteja na fronteira da árvore de busca

- Fronteira = PILHA (LIFO)

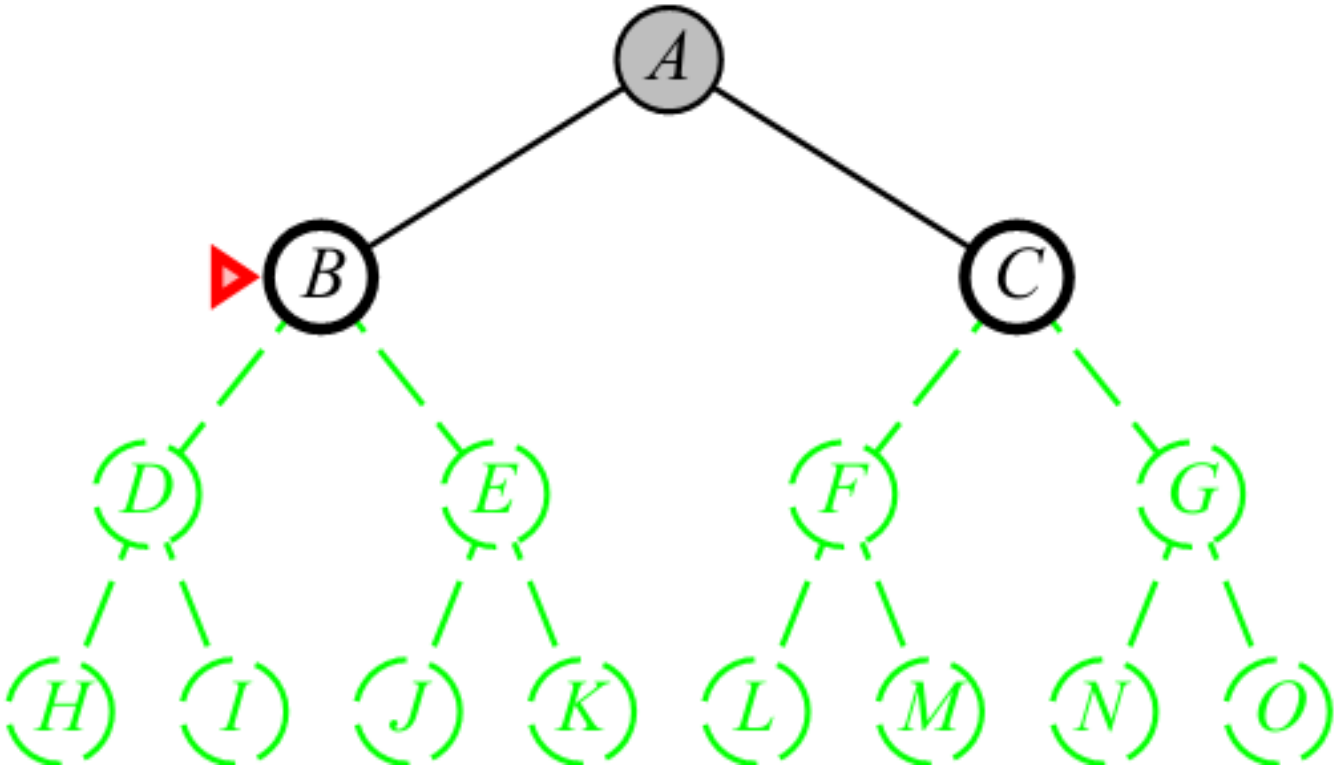
Busca em profundidade



A

BASE
FRONTEIRA

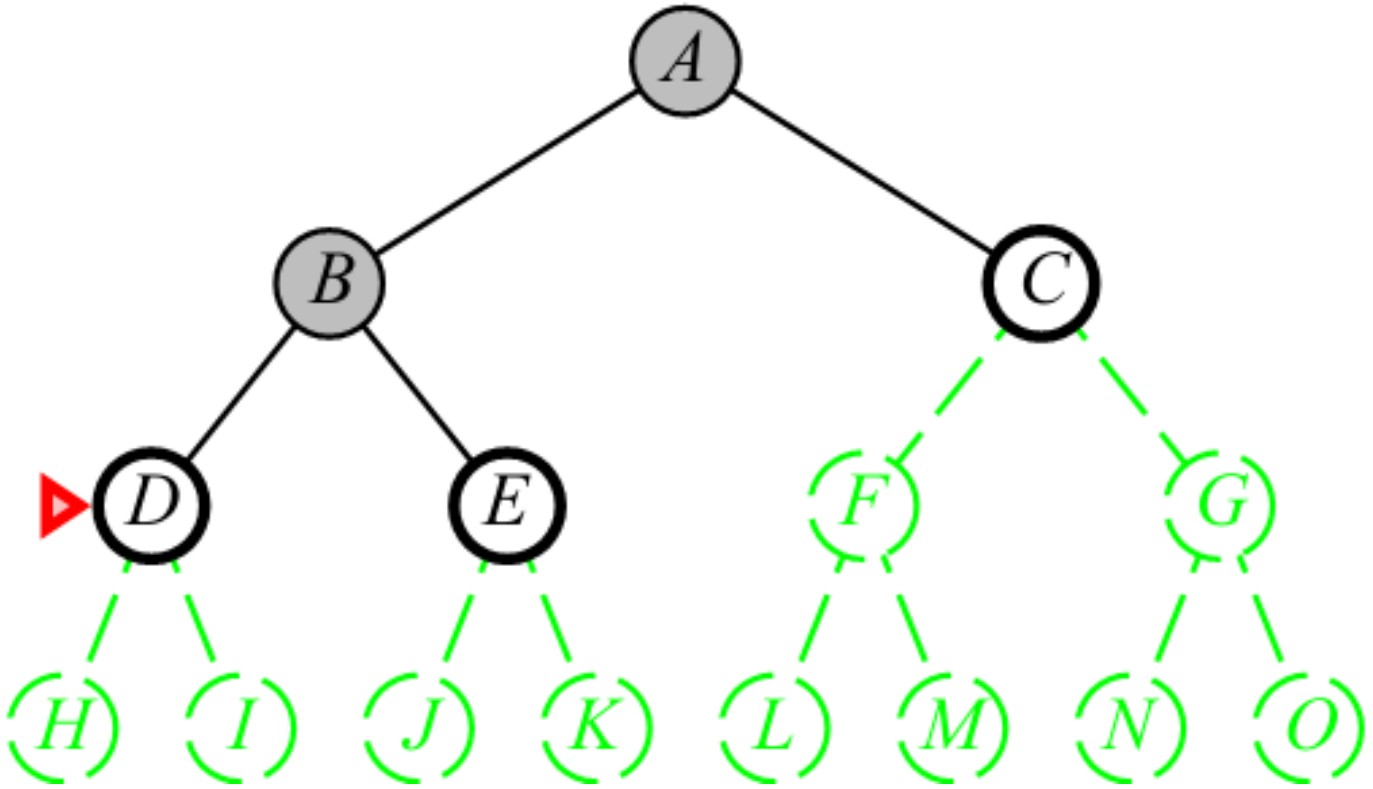
Busca em profundidade



B
C

BASE
FRONTEIRA

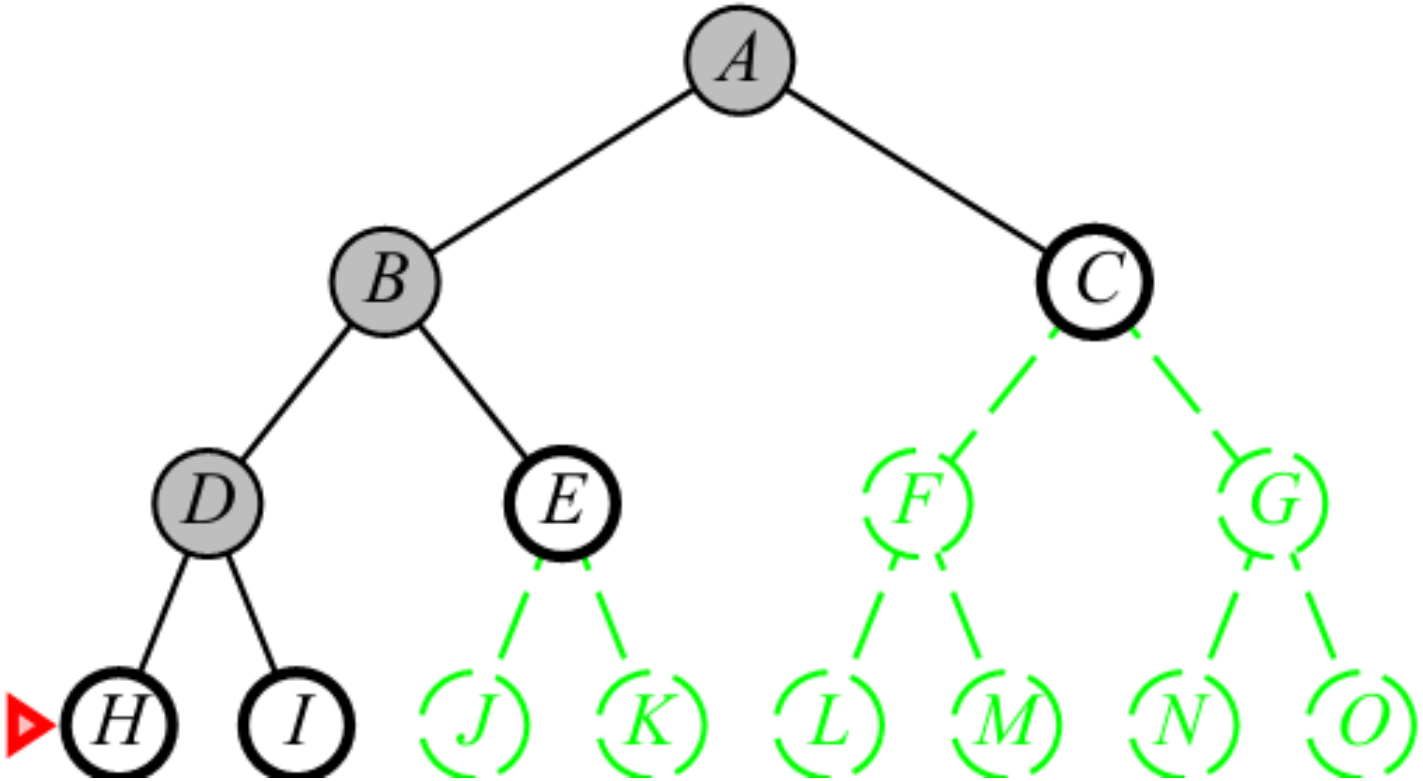
Busca em profundidade



D
E
C

BASE
FRONTEIRA

Busca em profundidade

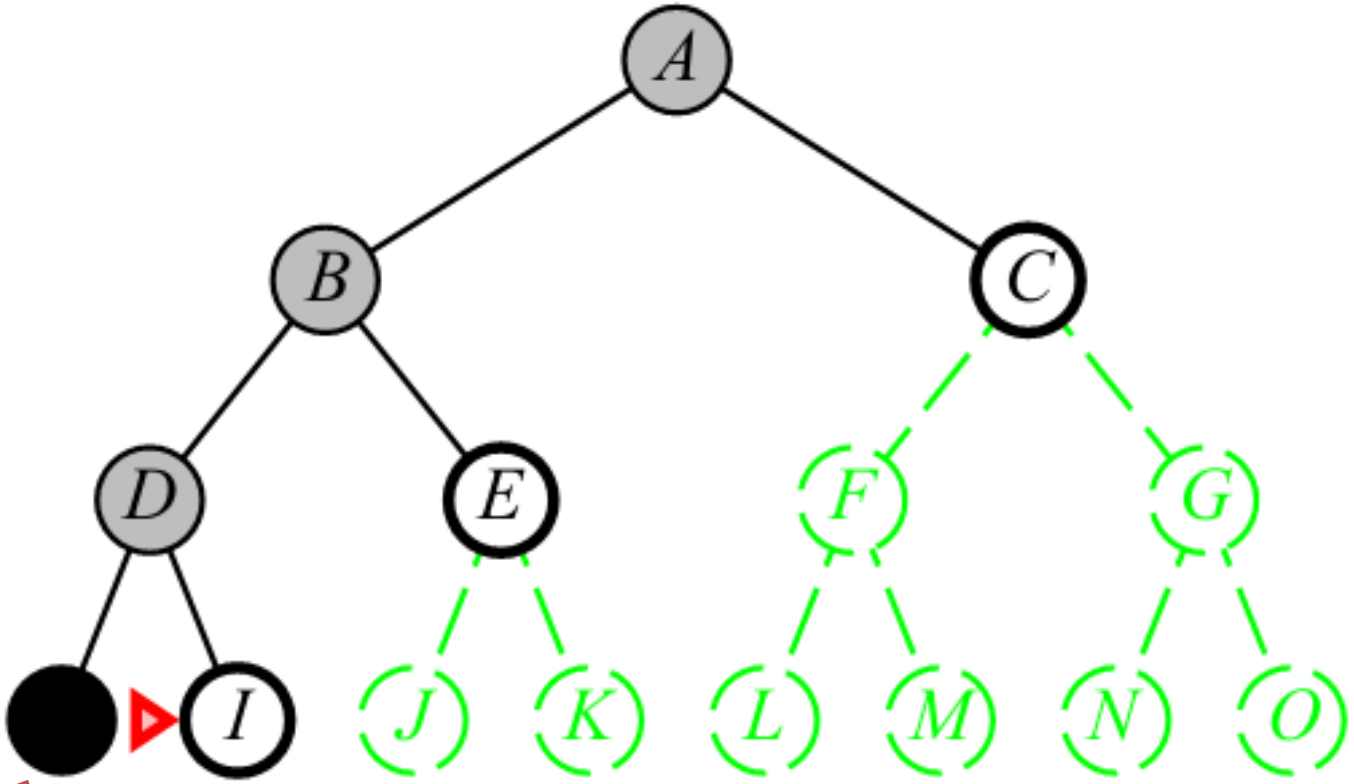


H
I
E
C

BASE
FRONTEIRA

Situação de maior ocupação de memória:
(3 nós na fronteira + 3 já explorados)
1 + 2 + 2 + 2 = 1 + 3.2 = 1 + m.b

Busca em profundidade

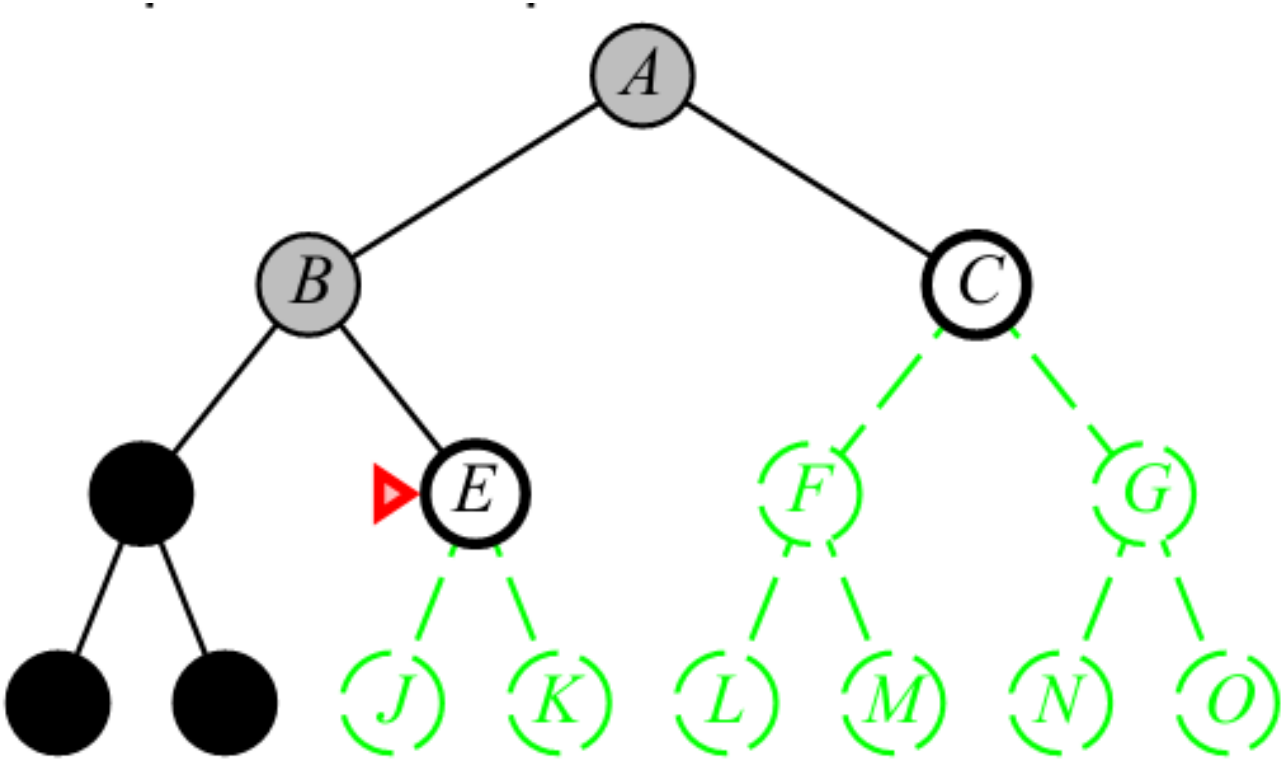


I
E
C

BASE
FRONTEIRA

**nó pode ser removido da memória uma vez que todos seus descendentes tenham sido explorados*

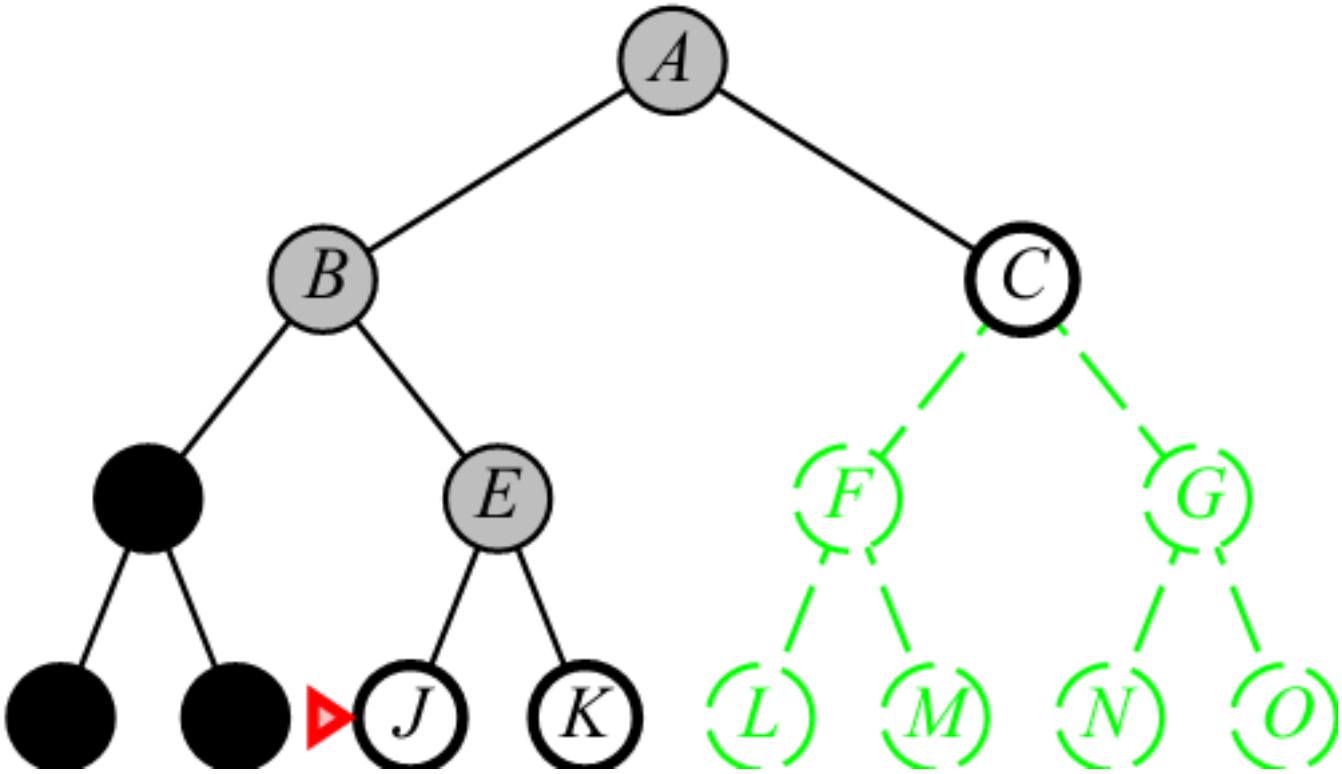
Busca em profundidade



E
C

BASE
FRONTEIRA

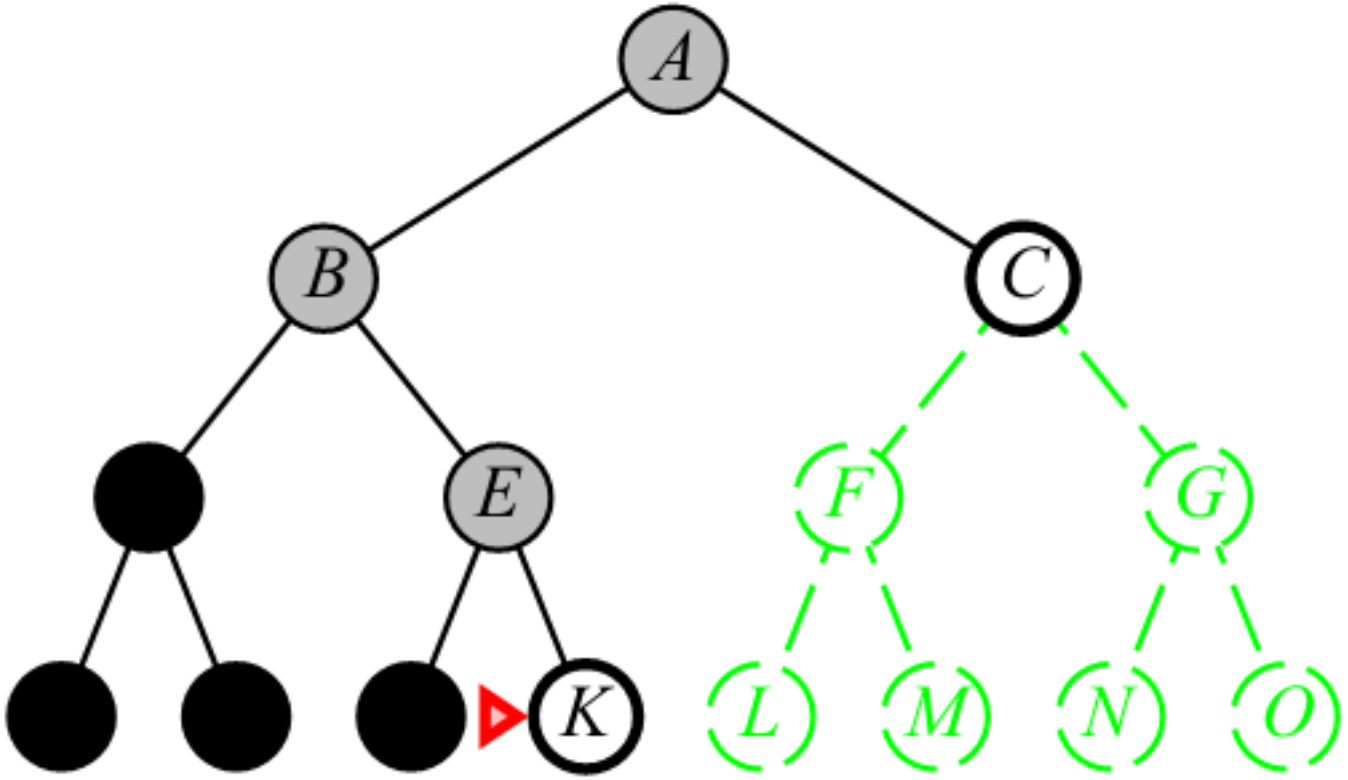
BUSCA EM PROFUNDIDADE



J
K
C

BASE
FRONTEIRA

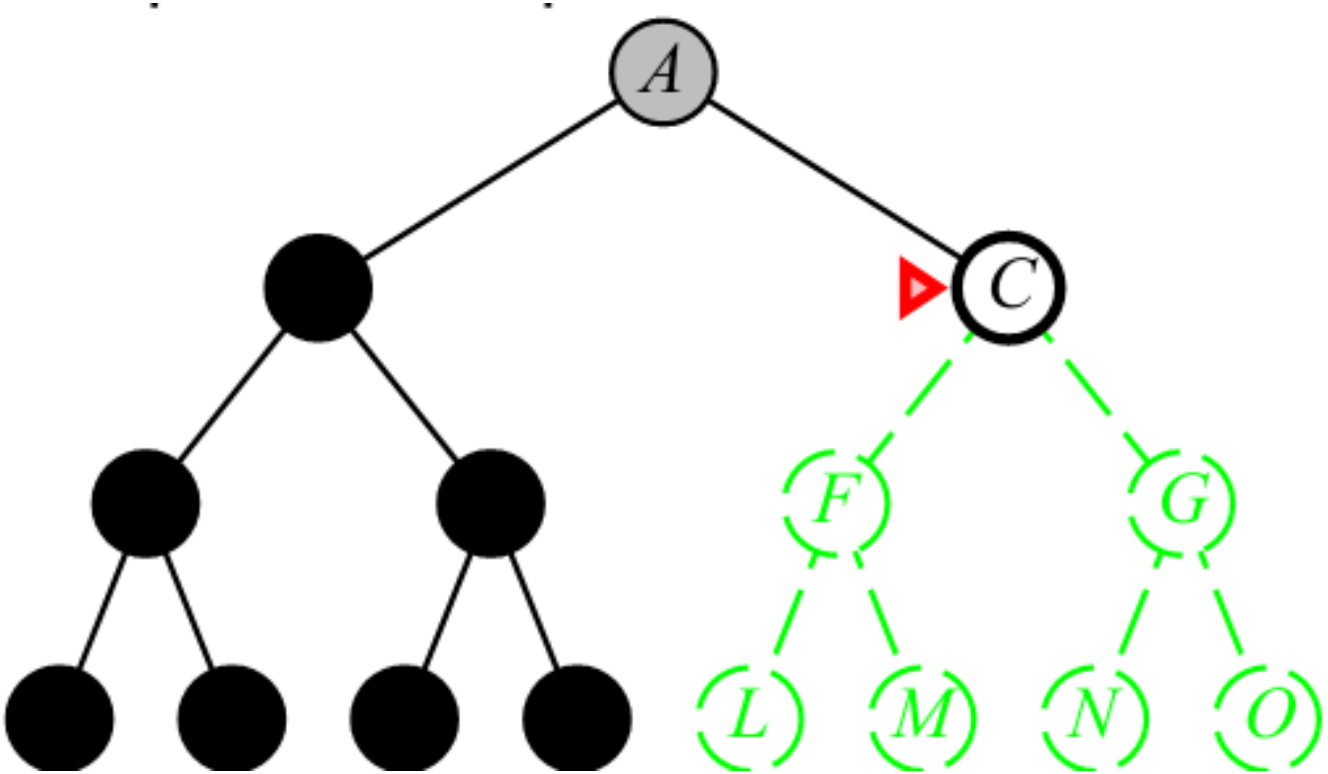
Busca em profundidade



K
C

BASE
FRONTEIRA

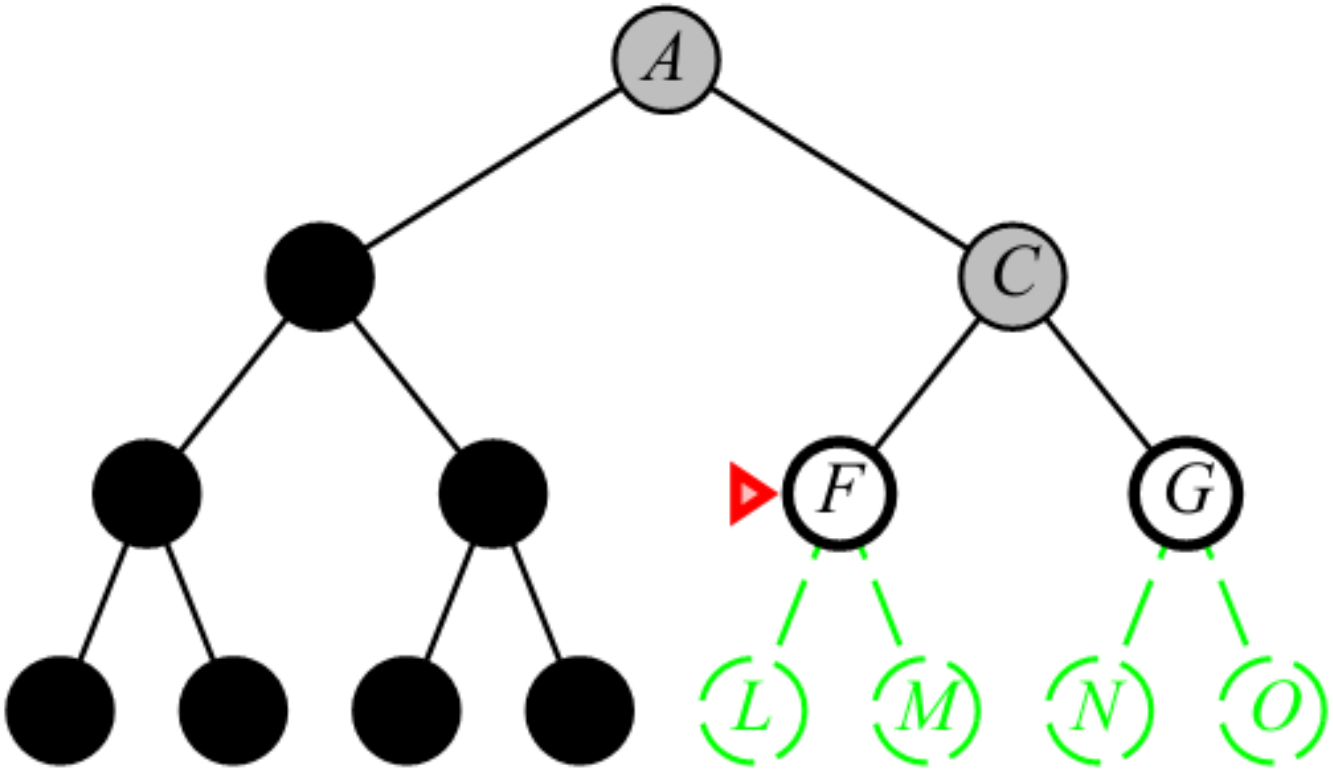
Busca em profundidade



C

BASE
FRONTEIRA

Busca em profundidade



Avaliação da busca em profundidade

Critério	Avaliação
Completo	Sim, se o espaço de estados for finito
Otimalidade	Não, pois retorna a 1a. Solução encontrada
Espacial	O(b.m)
Tempo	$O(b^m)$ Se a solução estiver no primeiro ramo então é linear em relação à m (pode ser infinito)

b: ramificação máxima entre todos os nós
m: tamanho máximo entre todos os caminhos no espaço de estados

Avaliação da busca em profundidade

- A baixa complexidade espacial da busca em profundidade com busca em árvore fez com que fosse aplicado em
  - Satisfação de restrições (Constraint satisfaction)
  - Satisfabilidade em lógica Prop. (Propositional satisfability)
  - Programação lógica (Logic programming)

BUSCA EM PROFUNDIDADE LIMITADA (DEPTH-LIMITED)

Resolução de problemas por meio de buscas

Busca em profundidade limitada

- Tenta amenizar o problema da busca em profundidade em espaços de estado de tamanho infinito
- Impõe um limite ℓ para a máxima profundidade a ser expandida
- Nós na profundidade ℓ são tratados como se não tivessem sucessores

Busca em profundidade limitada

- Problemas deste artifício:
  - Se $\ell < d$ a solução não será encontrada
  - e, portanto, é fonte de incompletude

  - Se $\ell > d$ não encontra o ótimo

- O problema é determinar o valor de ℓ!
  - Um modo é pegar o *tamanho máximo de caminho entre dois estados quaisquer n*o espaço de estados do grafo: **16**
  - Porém, sabemos que qualquer cidade pode ser alcançada a partir de qualquer outra em no máximo 9 passos (*diâmetro do espaço de estados*): **9**

- **d**: profundidade do nó objetivo mais raso

Busca em profundidade limitada

- Complexidade de tempo: $O(b^{\ell})$
- Complexidade de espaço: $O(b.\ell)$

Busca em profundidade limitada

```
function DEPTH-LIMITED-SEARCH(problem,limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INTIAL-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
```

BUSCA EM APROFUNDAMENTO ITERATIVO
(ITERATIVE DEEPENING SEARCH)

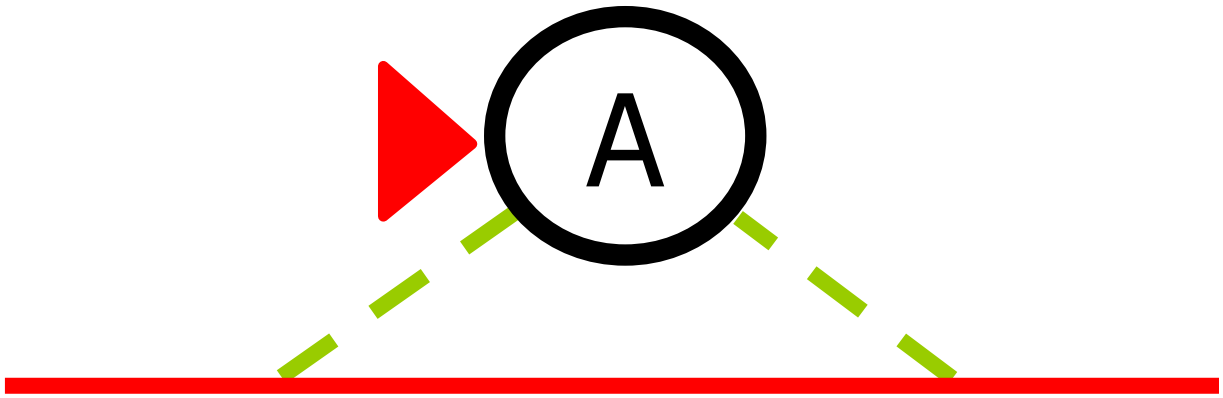
Resolução de problemas por meio de buscas

Busca de Aprofundamento Iterativo

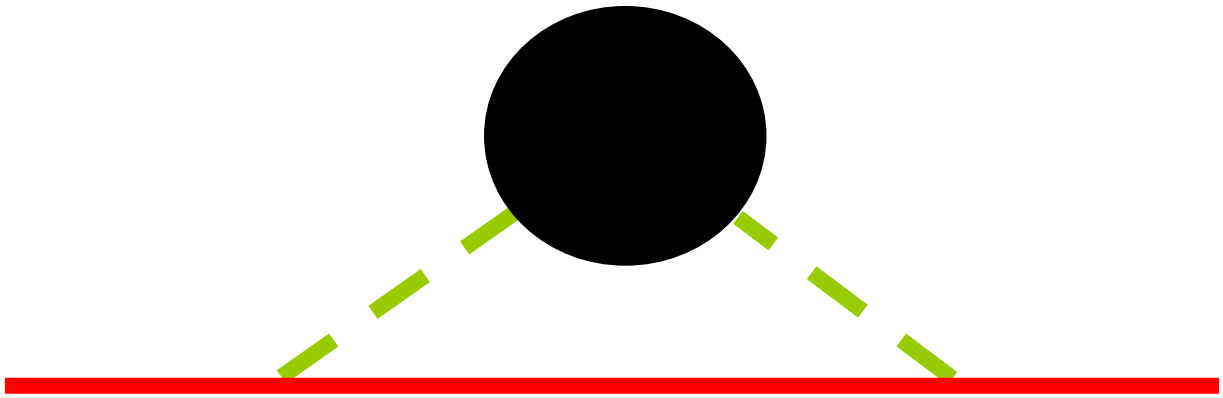
- Estratégia utilizada em conjunto com busca em profundidade para encontrar o melhor limite **l**
  - aumentar gradualmente *l* até encontrar um estado objetivo.

- Isto ocorre quando a profundidade alcançar **d**
  - (profundidade do objetivo mais raso)

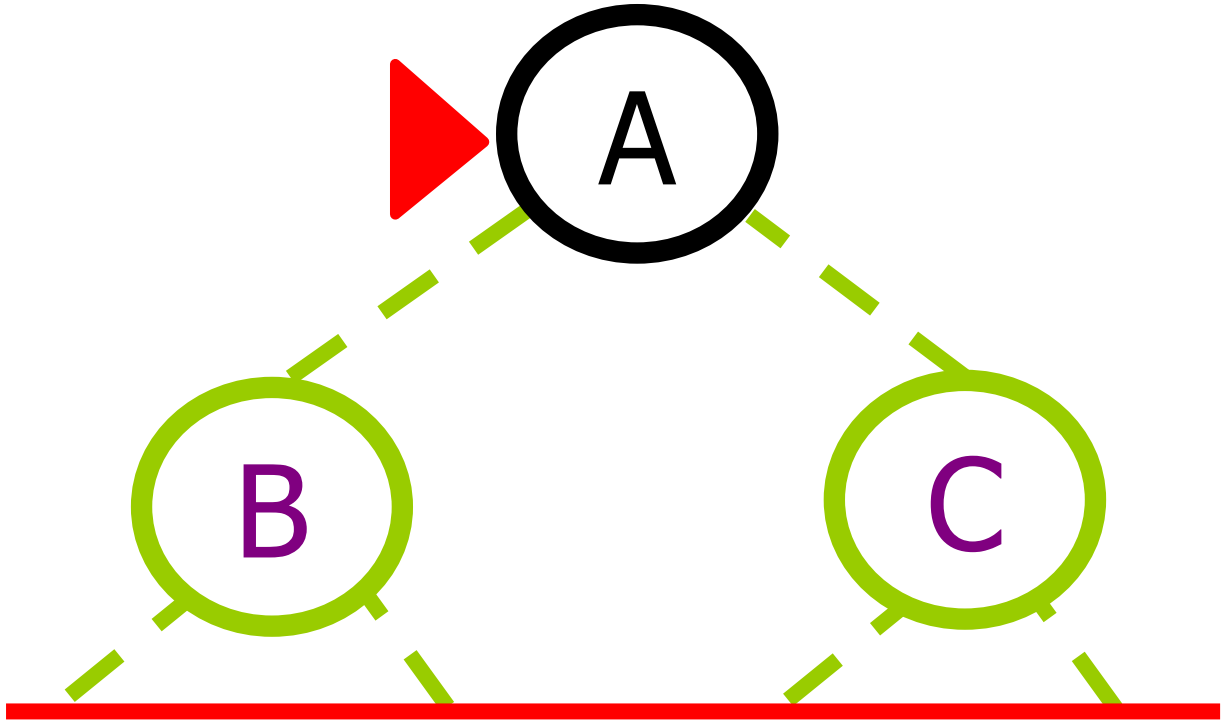
Busca de Aprofundamento Iterativo ℓ=0



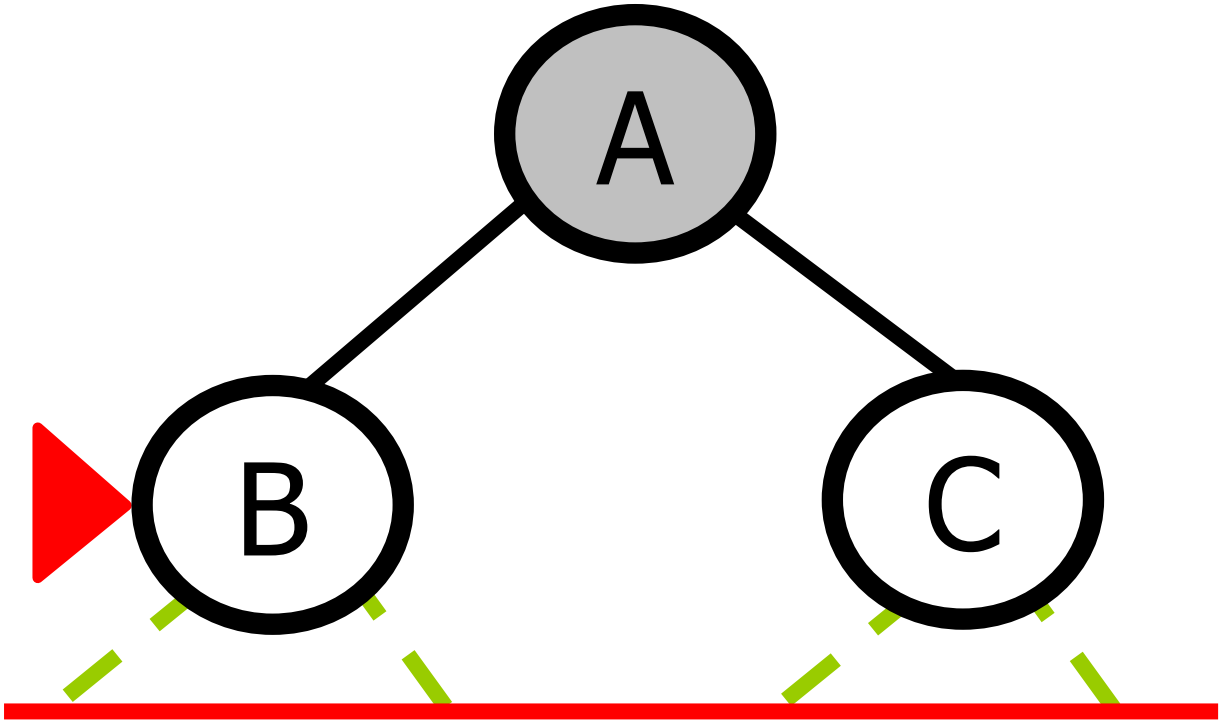
Busca de Aprofundamento Iterativo ℓ=0



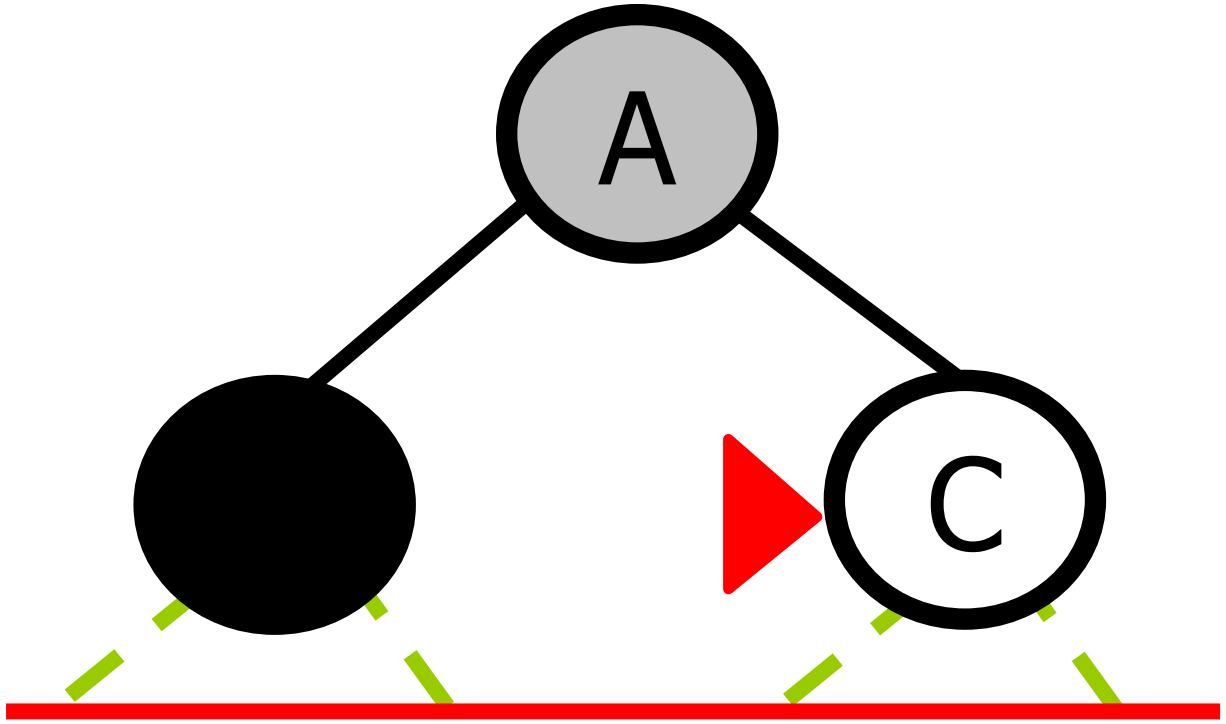
Busca de Aprofundamento Iterativo ℓ=1



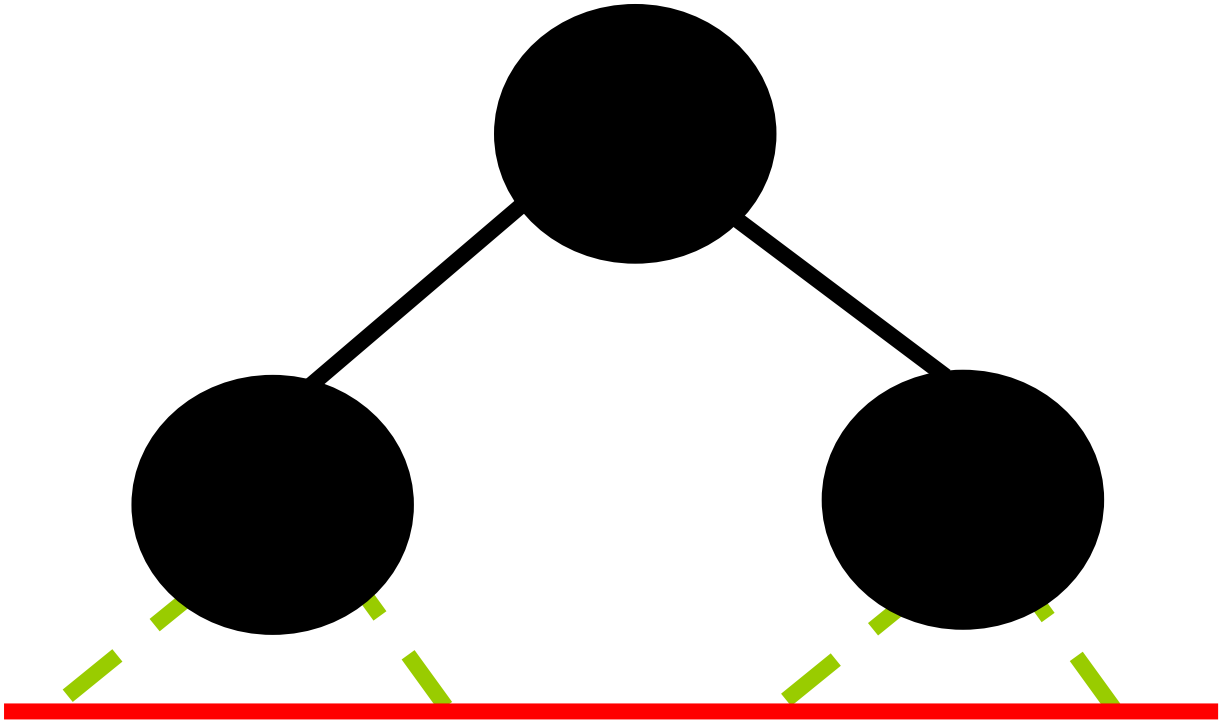
Busca de Aprofundamento Iterativo ℓ=1



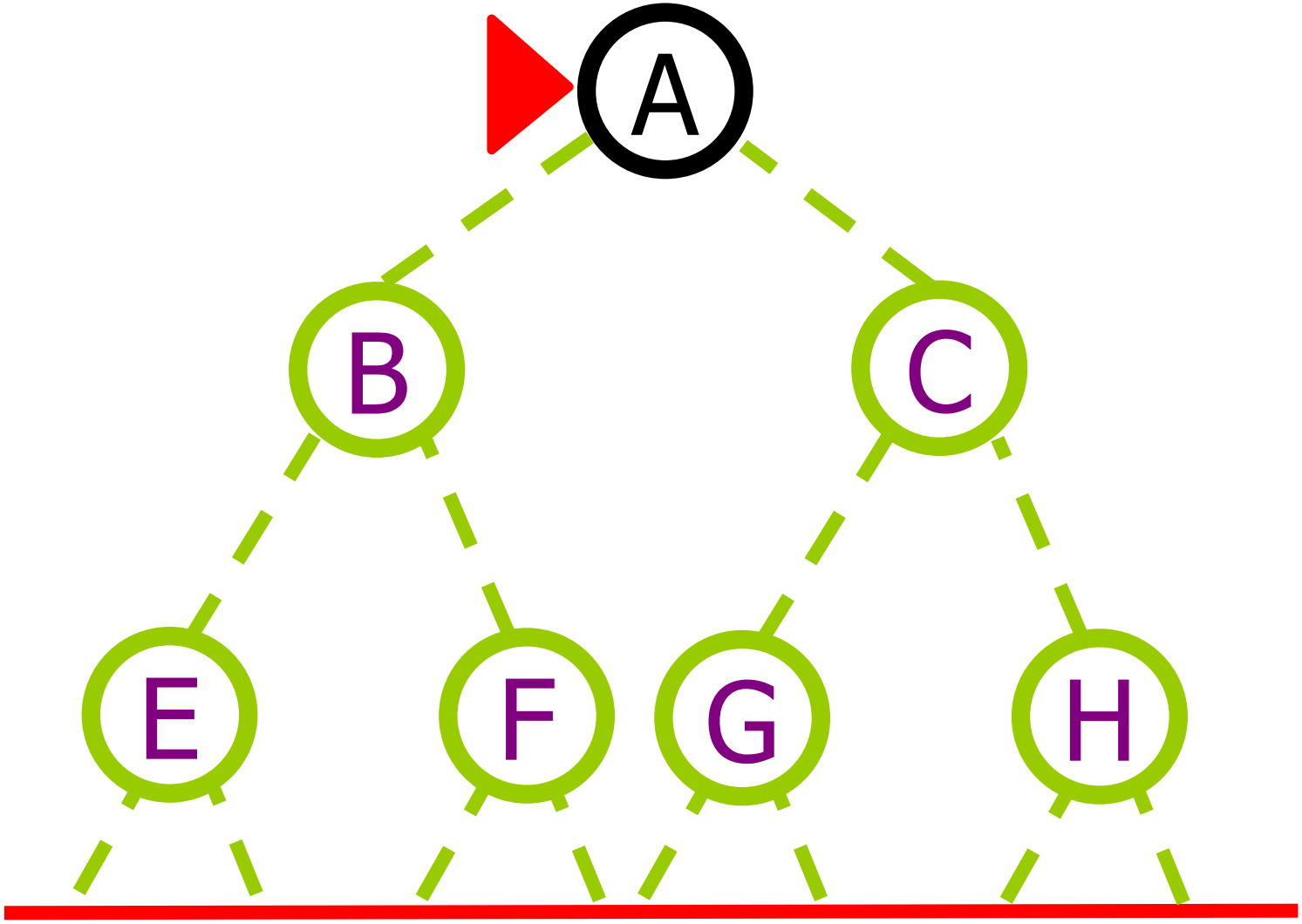
Busca de Aprofundamento Iterativo ℓ=1



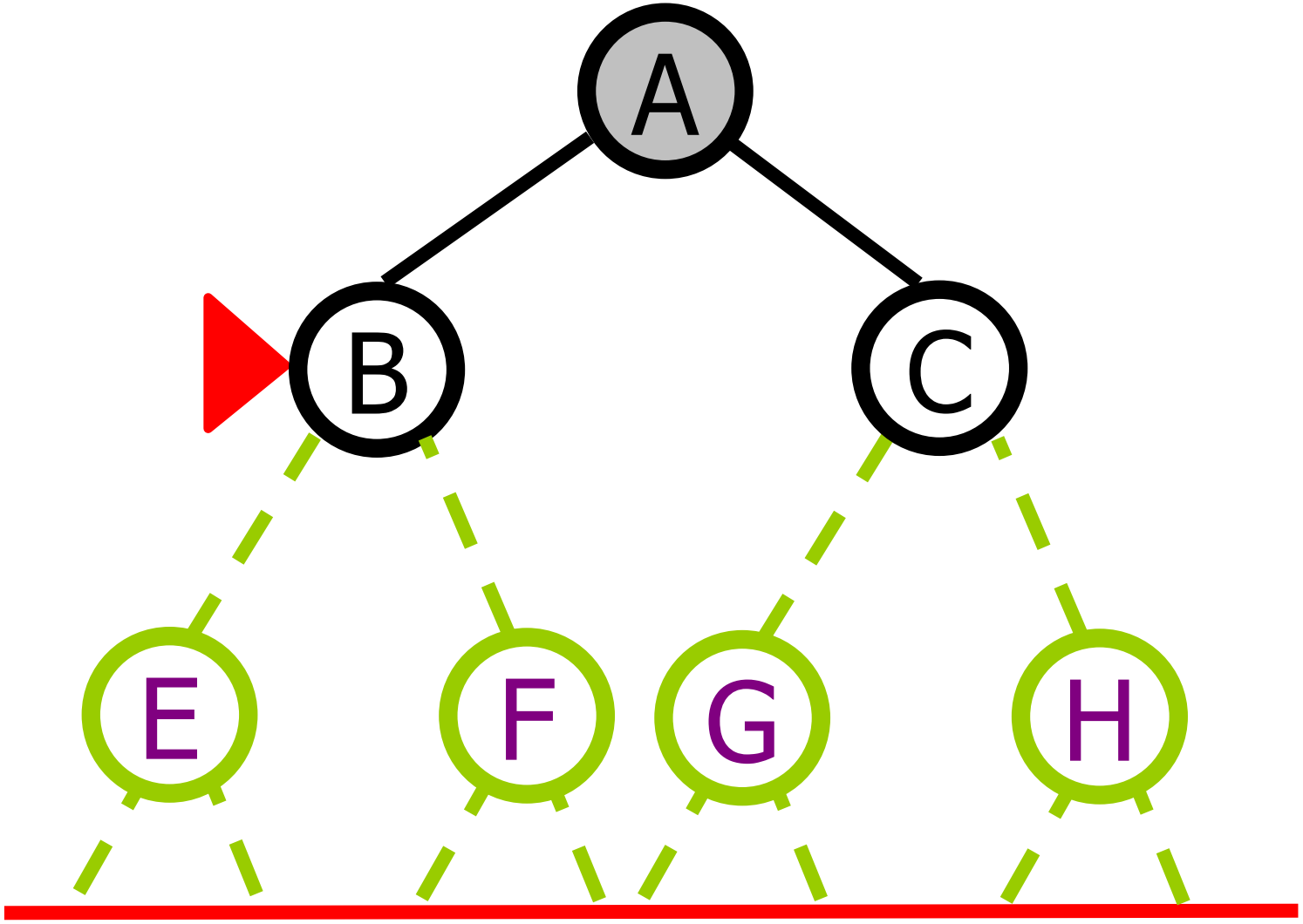
Busca de Aprofundamento Iterativo ℓ=1



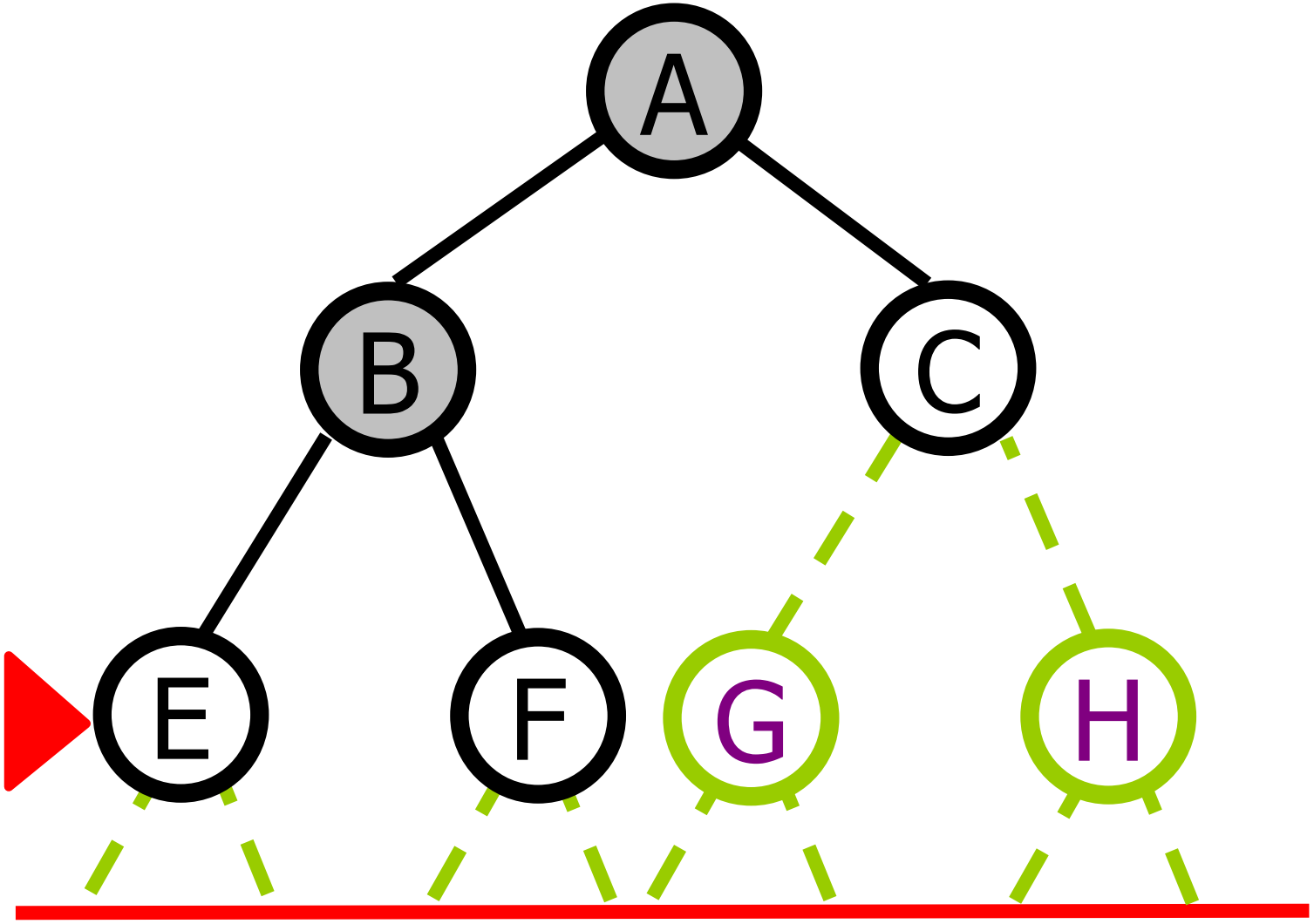
Busca de Aprofundamento Iterativo ℓ=2



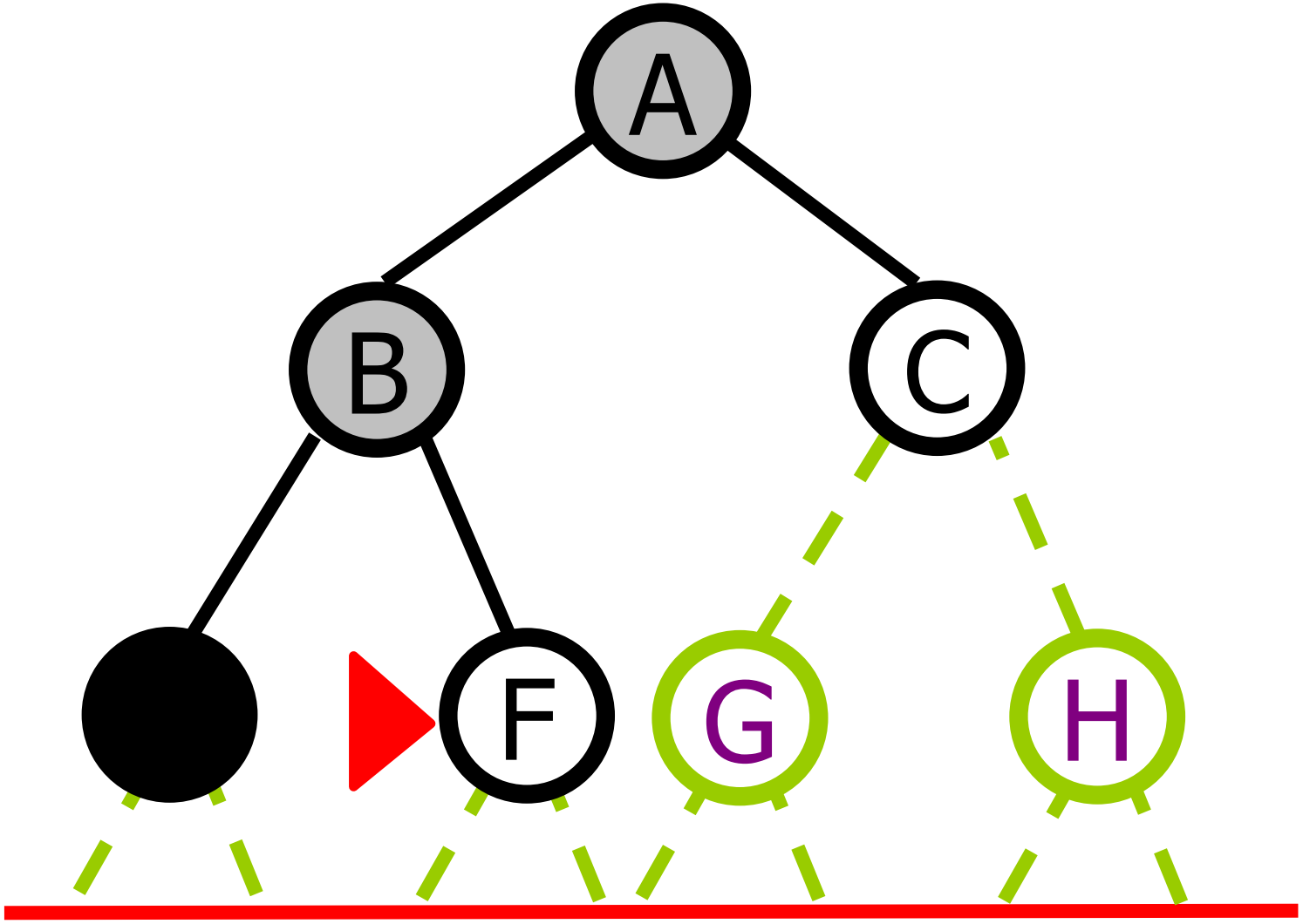
Busca de Aprofundamento Iterativo ℓ=2



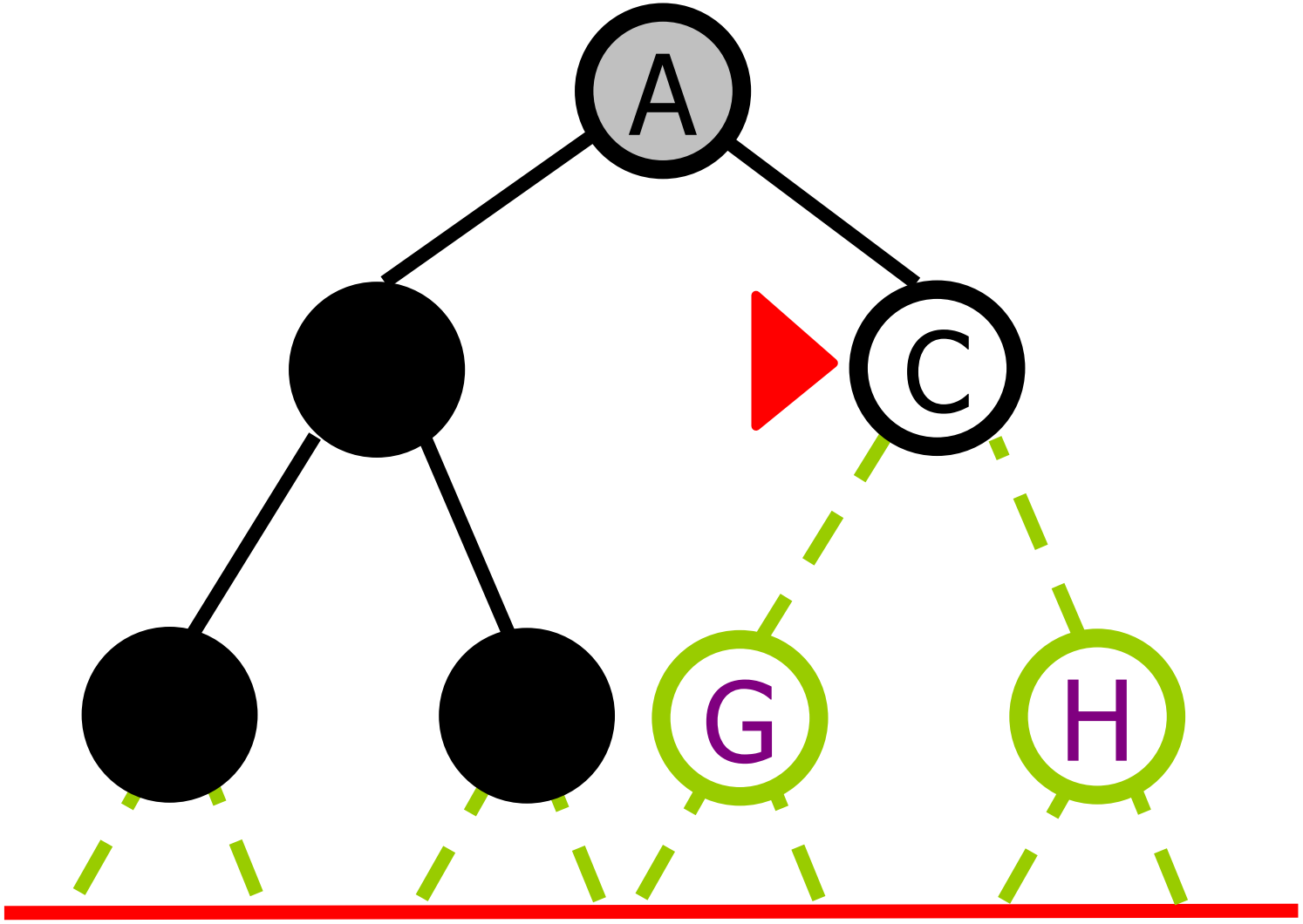
Busca de Aprofundamento Iterativo ℓ=2



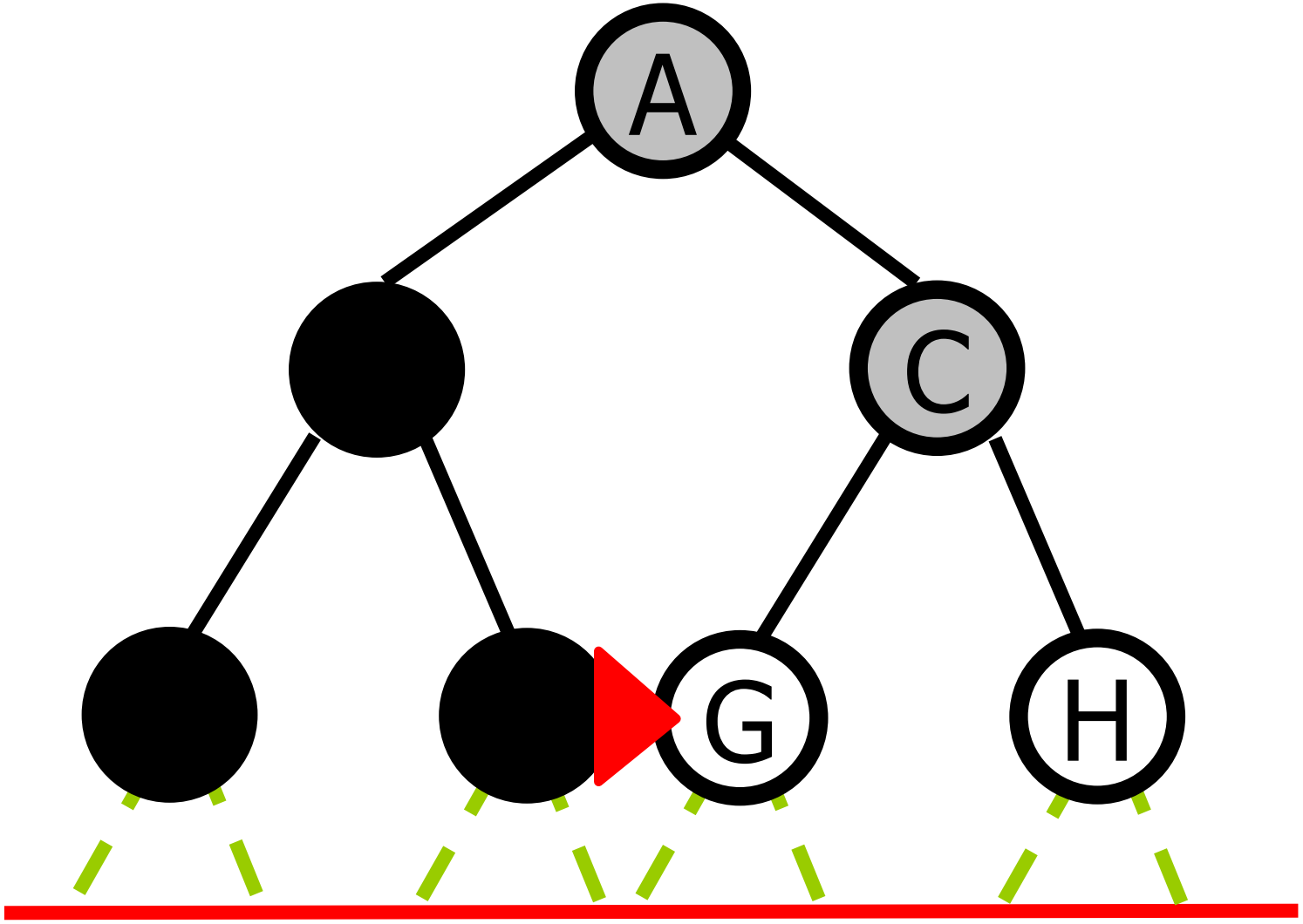
Busca de Aprofundamento Iterativo ℓ=2



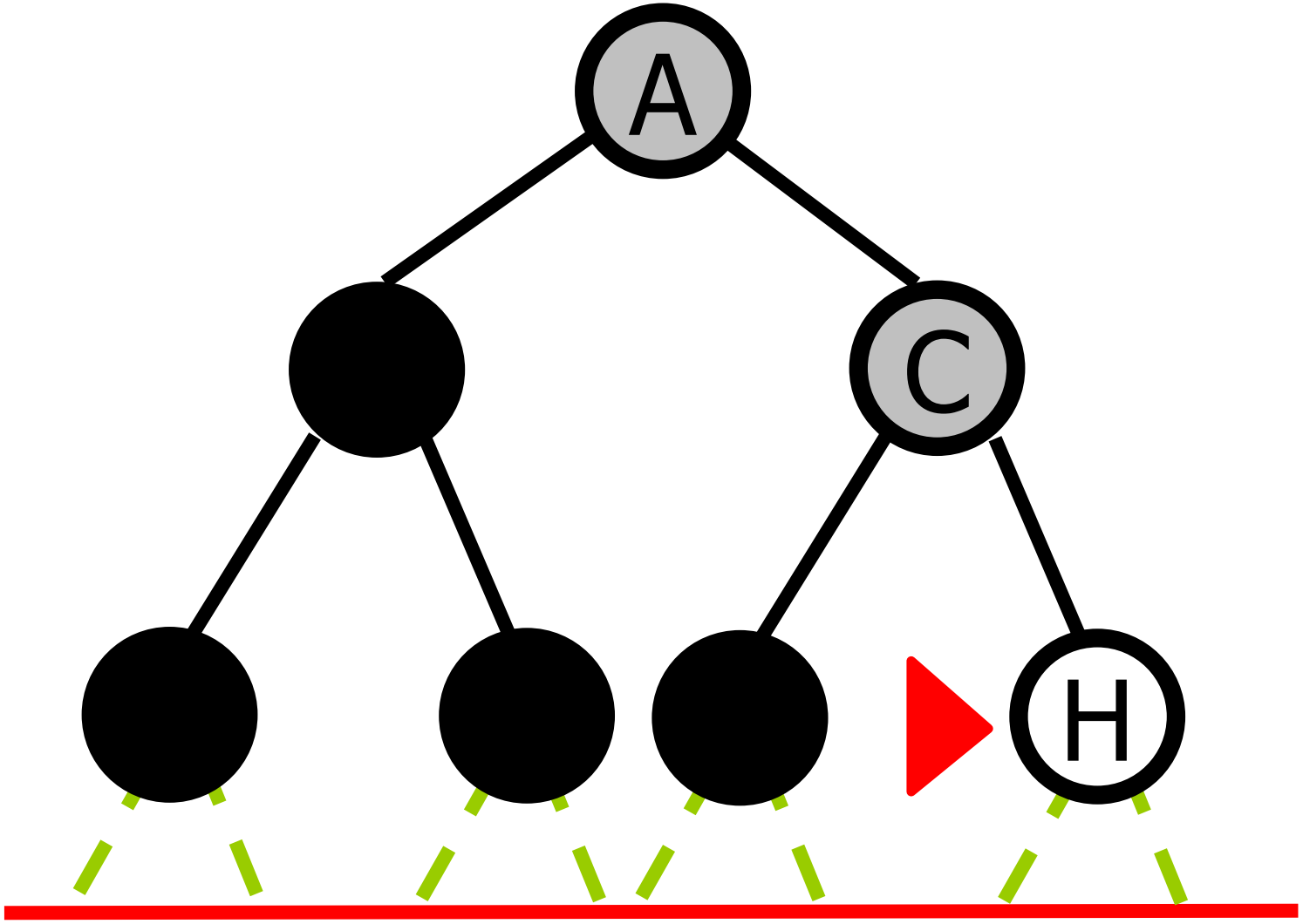
Busca de Aprofundamento Iterativo ℓ=2



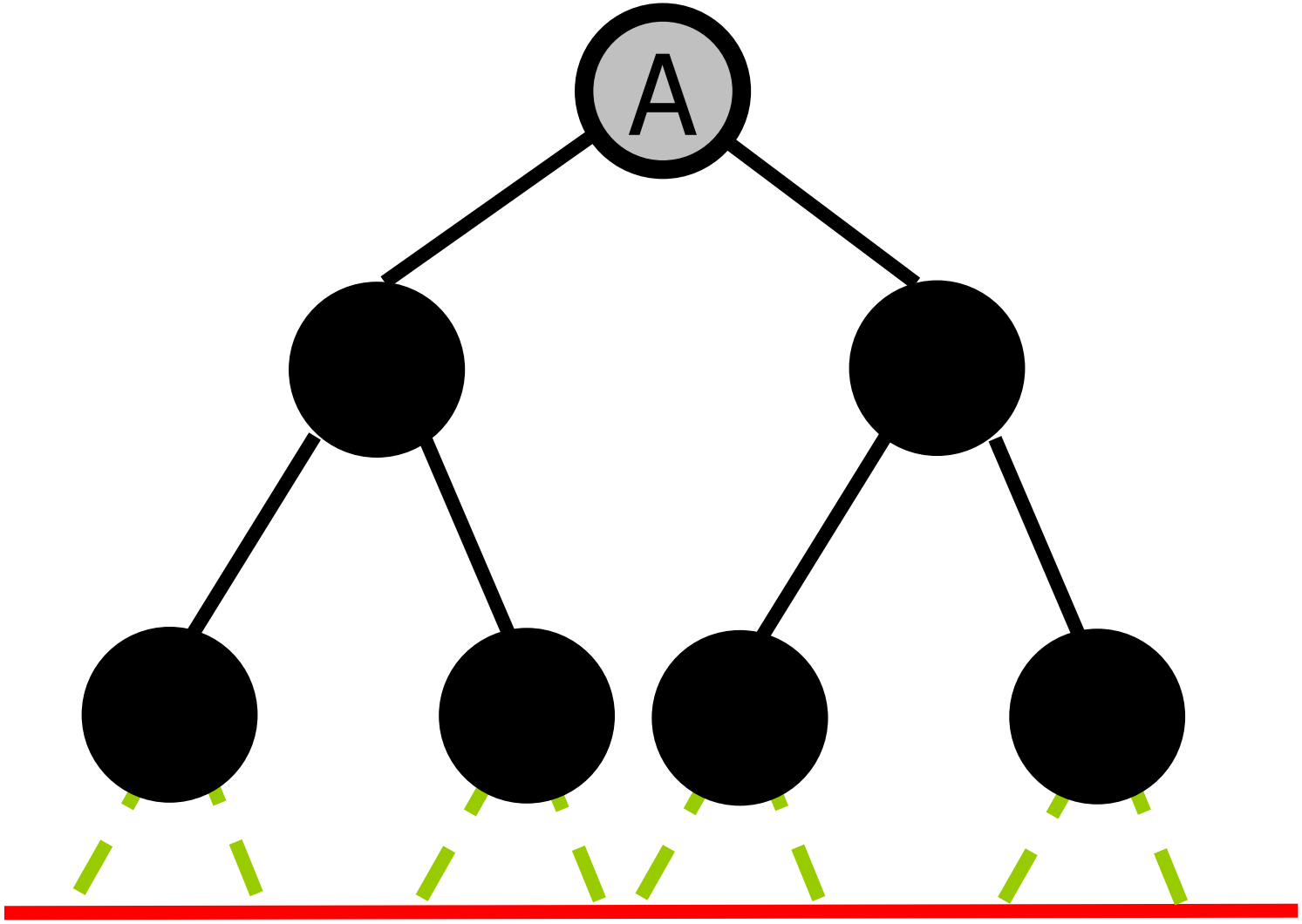
Busca de Aprofundamento Iterativo ℓ=2



Busca de Aprofundamento Iterativo ℓ=2

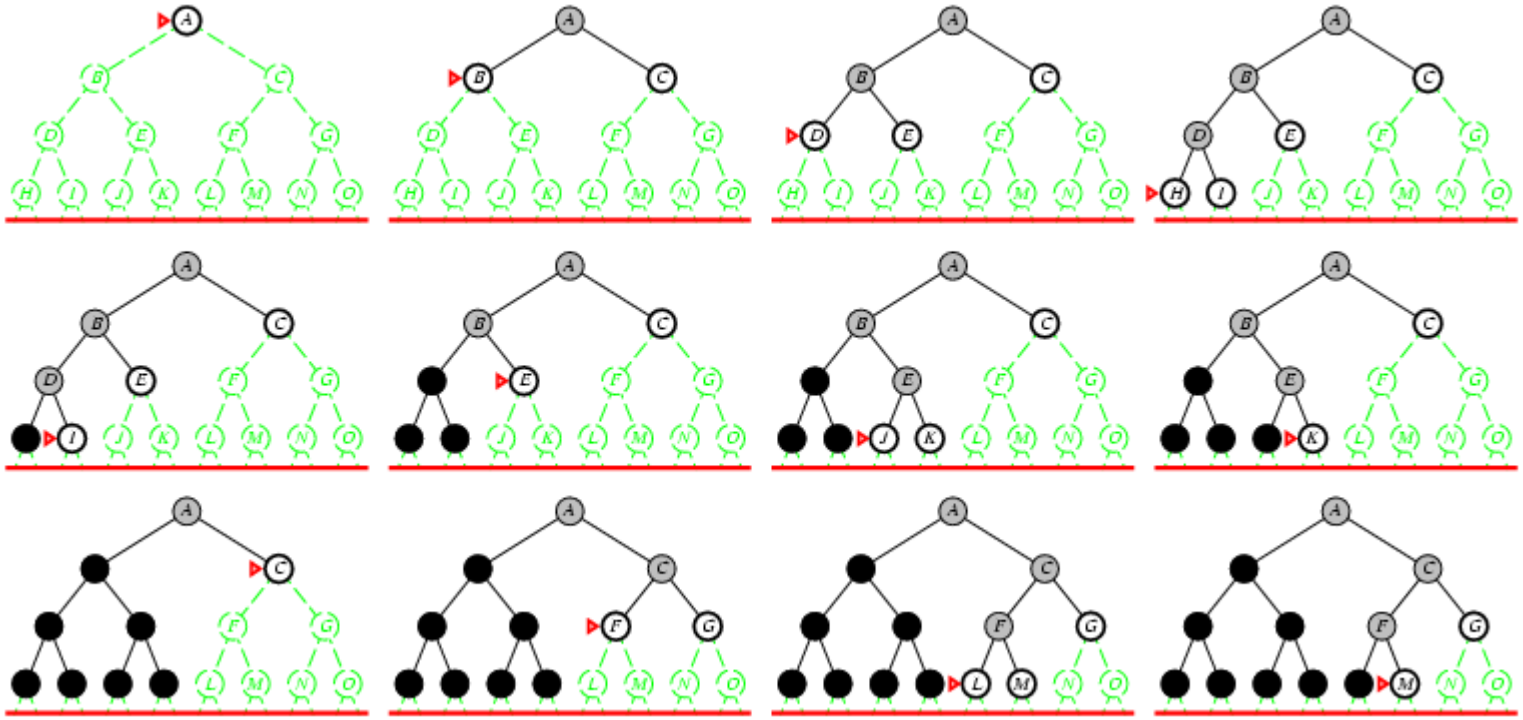


Busca de Aprofundamento Iterativo ℓ=2



Busca de Aprofundamento Iterativo ℓ=3

Limite l=3



Busca em Aprofundamento Iterativo

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

Avaliação da Busca de Aprofundamento Iterativo

Critério	Avaliação
Completo	Sim, quando b for finito = busca em largura
Otimalidade	Sim, se custos dos ações forem iguais
Tempo	$O(b^d)$ = busca em largura
Espacial	O(b.d) = busca em profundidade

Indicado quando o espaço de estados é grande e a profundidade da solução não é conhecida.