# ESTRATÉGIAS DE BUSCA NÃO INFORMADA (Exaustiva ou Cega)

*Profa. Huei Diana Lee*

*Inteligência Artificial*

*CECE/UNIOESTE-FOZ*

# Busca em Espaço de Estados

- Problema bem formulado >>

  **estado final** deve ser "buscado"

- Usar um **método de busca** para determinar a **ordem** correta de aplicação dos operadores que levará do estado **inicial** ao estado **final**

# Busca em Espaço de Estados

- Processo de

  **geração** (estados possíveis) e

  **teste** (verificação se o objetivo está entre eles)

- Busca terminada com sucesso >>

  executar a solução

  (= conjunto ordenado de operadores a aplicar)

# Busca em Espaço de Estados

**Fronteira** do espaço de estados:

nós (estados) passíveis para expansão no momento

# Ex: viajar de Curitiba a Camboriú

**estado inicial** => Curitiba

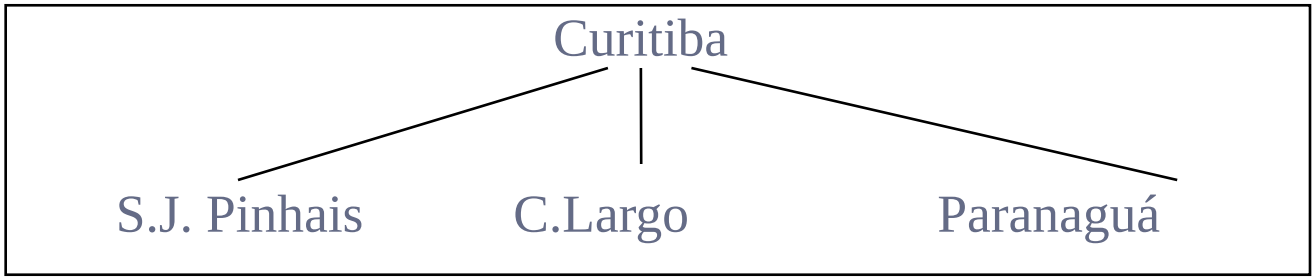
# Ex: viajar de Curitiba a Camboriú

**estado inicial** =>


**fronteira**

# Ex: viajar de Curitiba a Camboriú

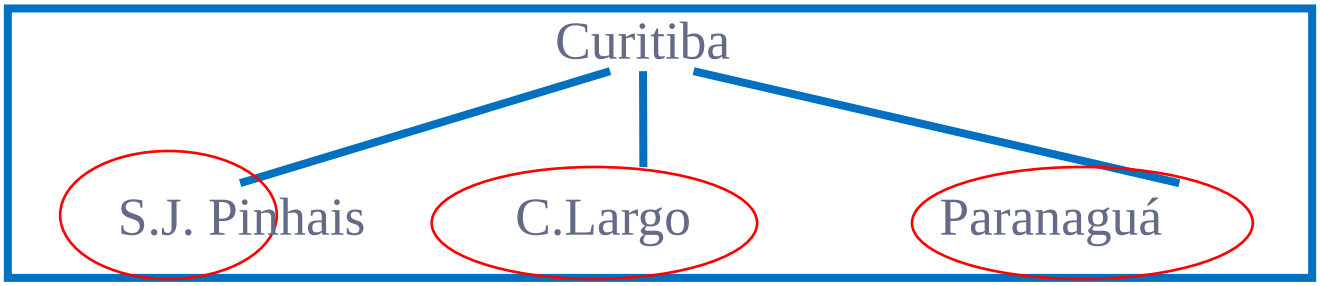
**estado inicial** =>

# Ex: viajar de Curitiba a Camboriú

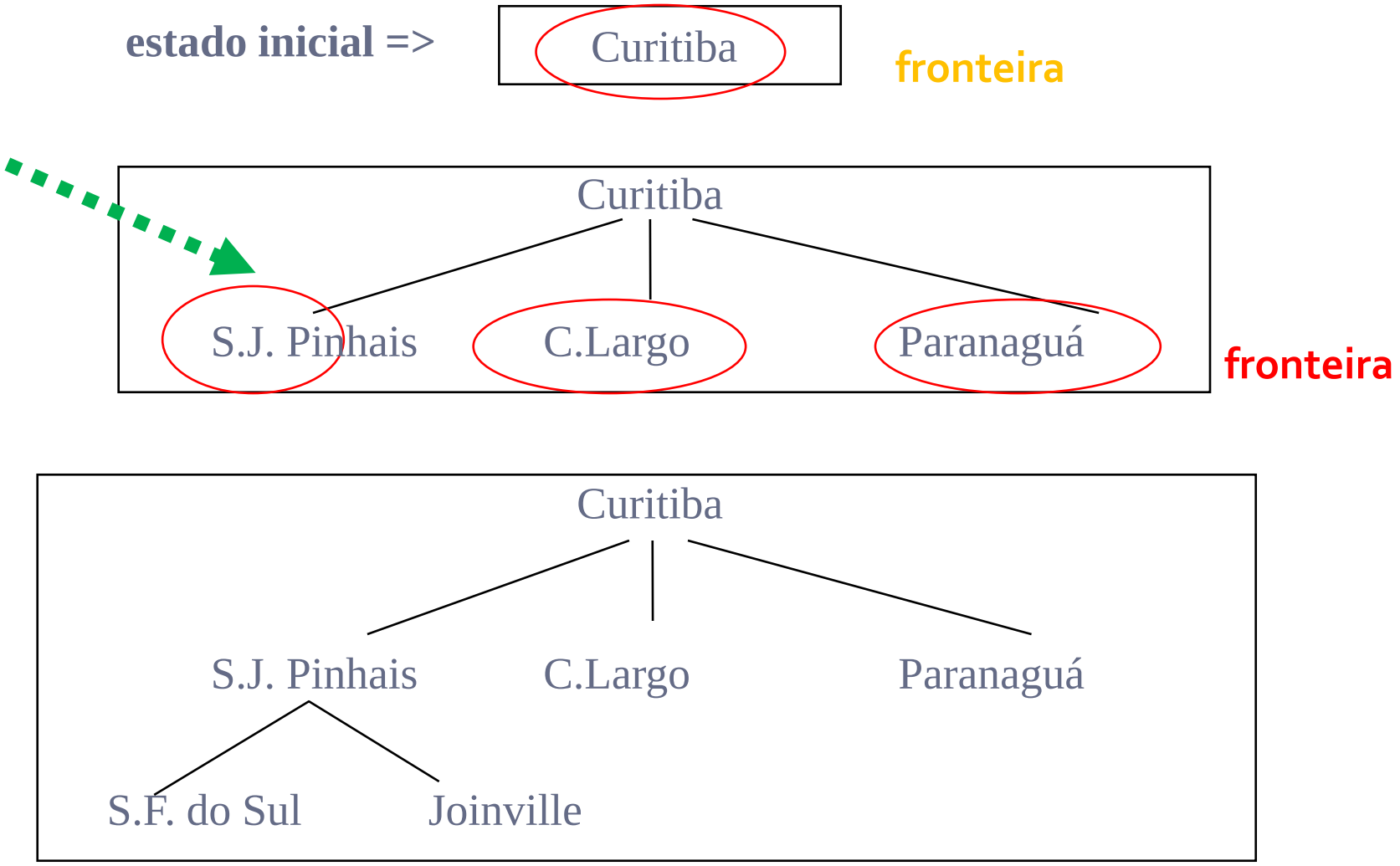
**estado inicial** => Curitiba

fronteira




fronteira

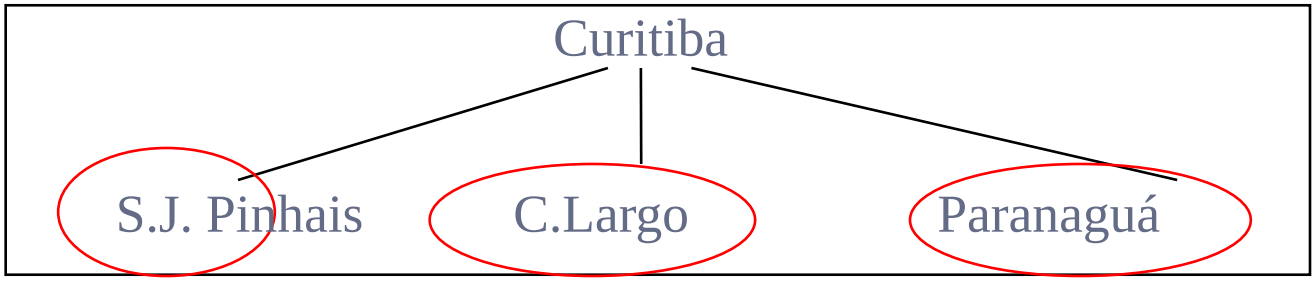
# Ex: viajar de Curitiba a Camboriú



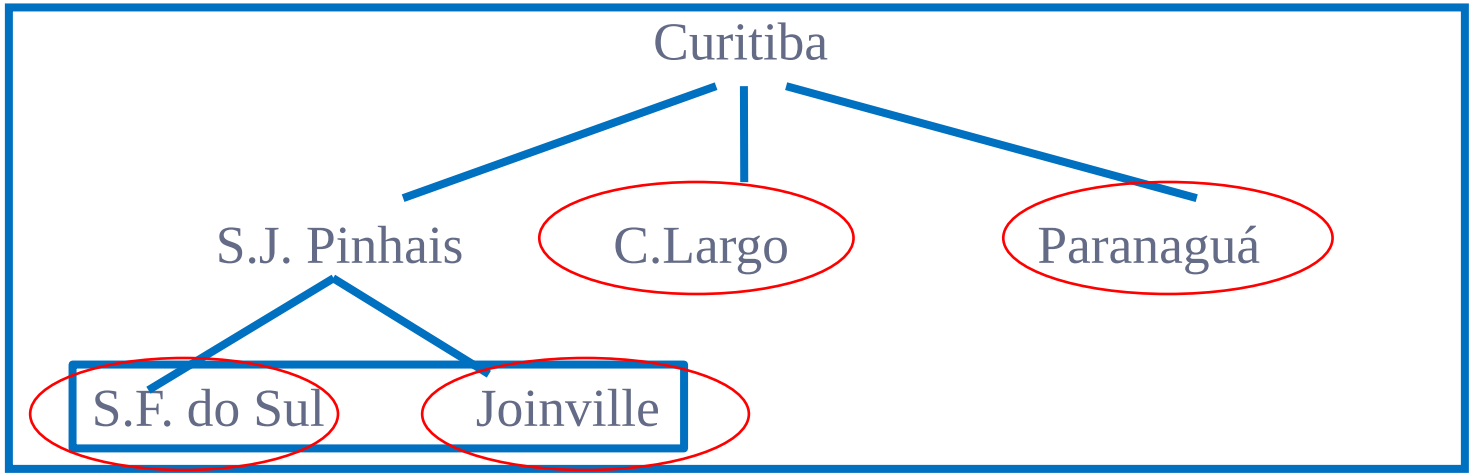
# Ex: viajar de Curitiba a Camboriú

**estado inicial** =>


fronteira




fronteira




fronteira

# Busca em Espaço de Estados:

## Geração e Teste - Implementação (Descrição Informal)

```
função BUSCA(problema, estratégia)
     retorna uma solução ou falha

     iniciar a árvore de busca usando um estado inicial
        de problema
     repita
        se não existe nenhum candidato para expansão
           (fronteira vazia)
        então retornar falha
     escolher um nó-folha para expansão de acordo com a
        estratégia
     se o nó contém um estado objetivo
        então retornar a solução correspondente
     senão expandir o nó e adicionar os nós resultantes
        à árvore de busca
```

# Busca em Espaço de Estados:

## Considerações sobre Implementação

- **Espaços de Estados**: podem ser representados como uma árvore onde os estados são nós e as operações são arcos
- Os nós da árvore podem guardar mais informação do que apenas o estado
- Estrutura de dados com pelo menos cinco componentes:
  - estado correspondente
  - seu nó pai
  - operador aplicado para gerar o nó (a partir do pai)
  - nós-filhos
  - profundidade do nó
  - custo do nó (desde a raiz)
  - outros

# Busca em Espaço de Estados:

## Implementação

**Função-Insere:** controla a ordem de inserção de nós na fronteira do espaço de estados

```
função Busca-Genérica (problema, Função-Insere)
        retorna uma solução ou falha
  fronteira ← Lista(Nó(Estado-Inicial[problema]))
  loop do
    se fronteira está vazia então retorna falha
    nó ← Remove-Primeiro(fronteira)
    se Teste-Término[problema] aplicado a
        Estado[nó] = sucesso
         então retorna nó
    fronteira ←
        Função-Insere(fronteira,Operadores[problema,
    nó])
  end
```

# Métodos de Busca

## Busca exaustiva ou cega (não informada)

- Não se sabe qual o melhor nó da fronteira a ser expandido, isto é, o menor custo de caminho desse nó até um nó final (objetivo)

## Busca heurística (informada)

- Estima-se qual o melhor nó da fronteira a ser expandido com base em funções heurísticas (conhecimento)

# Busca Cega

## Estratégias para determinar a ordem de ramificação dos nós:

- Busca em profundidade e profundidade limitada
- Busca com aprofundamento iterativo
- Busca em largura
- Busca de custo uniforme

## Direção da ramificação:

- Do estado inicial para um estado final
- De um estado final para o estado inicial
- Busca bidirecional

# Critérios de Avaliação das Estratégias de Busca

- **Completa**?
  - a estratégia sempre encontra uma solução quando existe alguma?
- **Ótima**?
  - a estratégia encontra a melhor solução quando existem soluções diferentes? (menor caminho ou menor custo)
- **Custo de tempo**?
  - quanto "tempo" gasta para encontrar uma solução?
- **Custo de memória**?
  - quanta "memória" é necessária para realizar a busca?
- **Complexidade**:
  - b: fator de ramificação
  - d: profundidade do nó objetivo
  - m: comprimento máximo

# Busca em Profundidade

- Ordem de ramificação dos nós:

  Sempre expande o nó no nível mais **profundo** da árvore:

  - nó raiz
  - primeiro nó de profundidade 1
  - primeiro nó de profundidade 2, e assim por diante

- Variante com **retrocesso** ou ***backtracking*** :

  Quando um nó final não é solução (ou é, mas se quer encontrar todas as soluções), o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço de estados

# Busca em Profundidade

**Algoritmo:**

**função** Busca-em-Profundidade(problema)
retorna uma solução ou falha
Busca-Genérica (problema, ***Insere-no-Começo***)

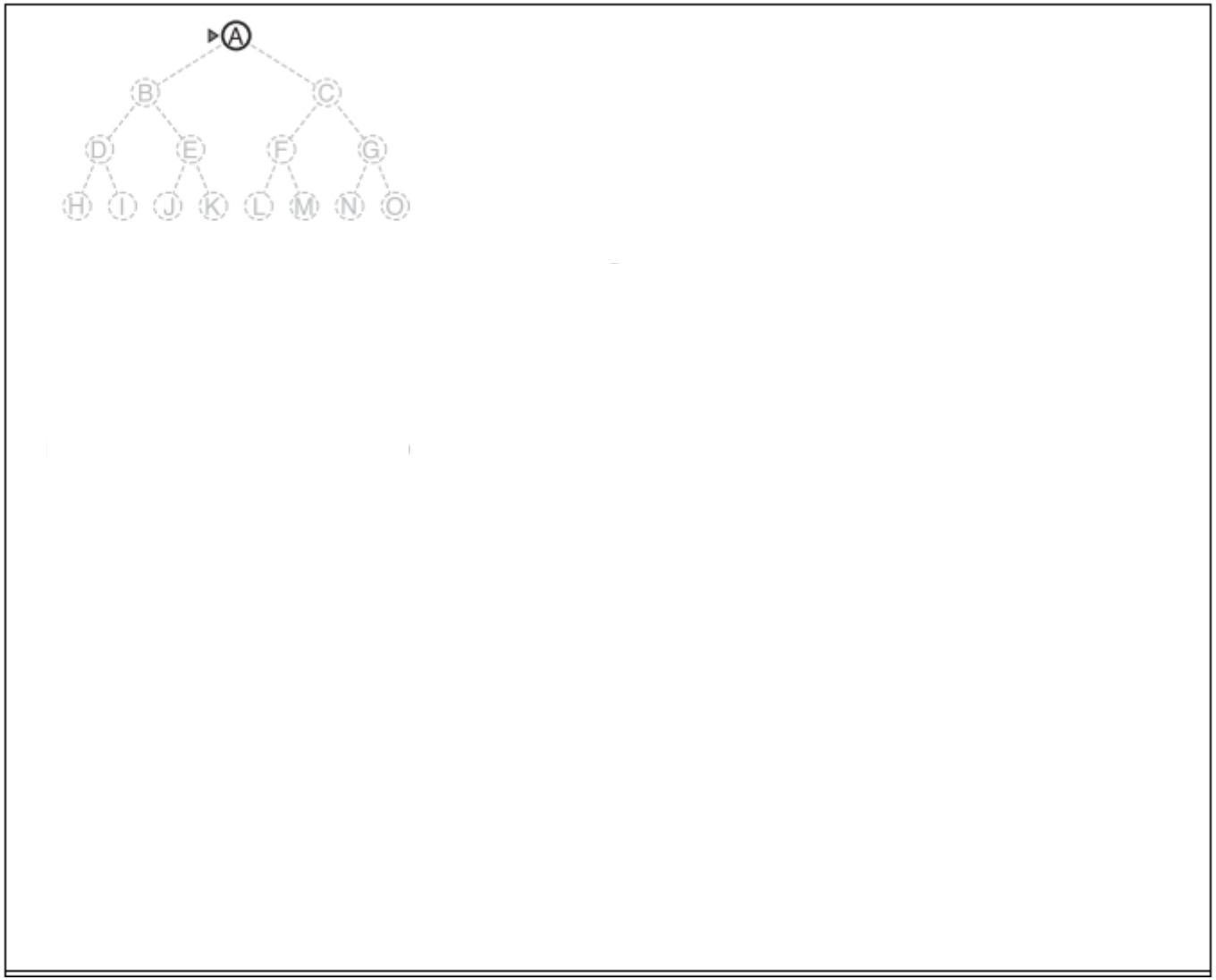
***Insere-no-Começo***: que tipo de lista?



OBS: Algoritmo de Busca-Genérica s13

# Busca em Profundidade

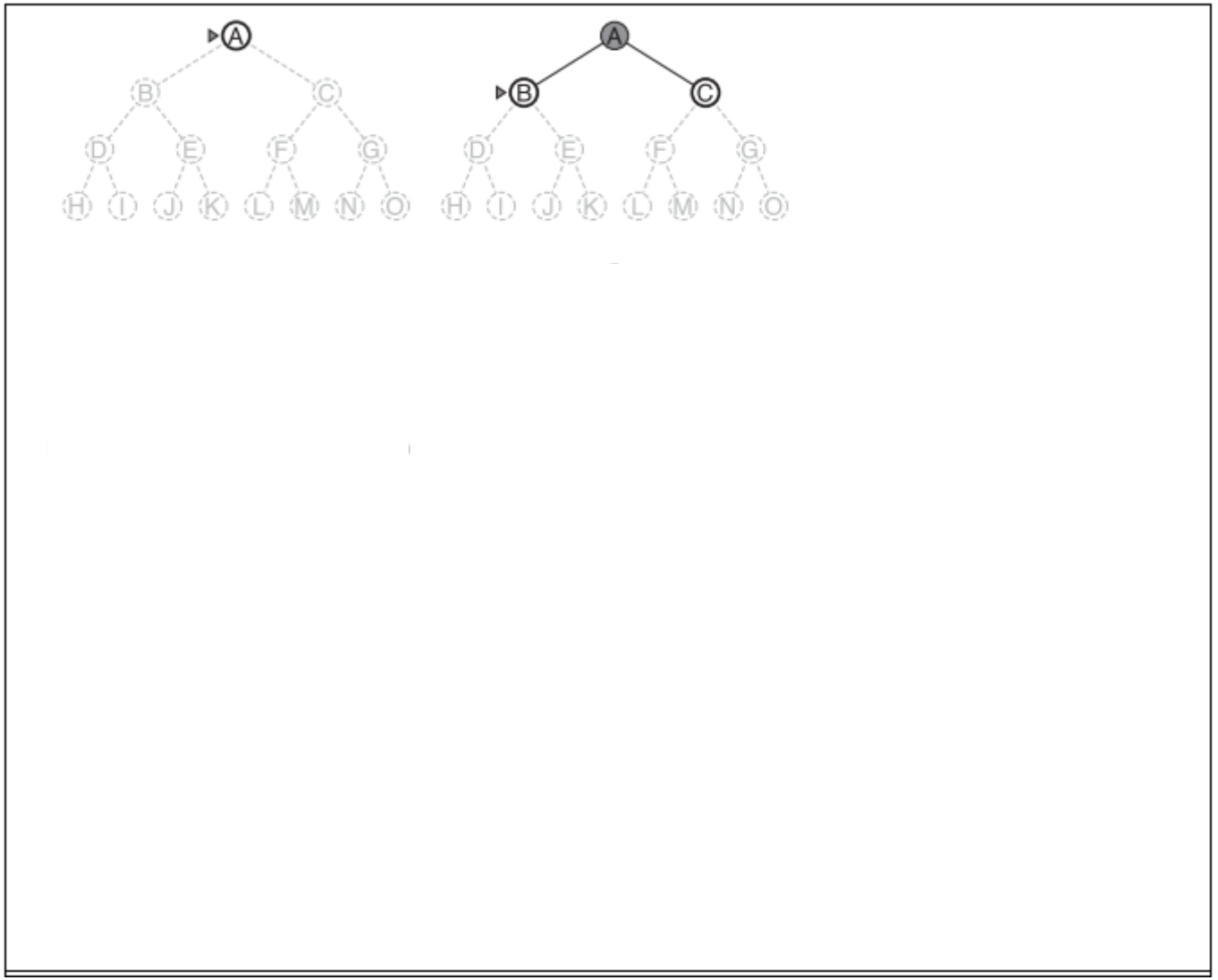
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

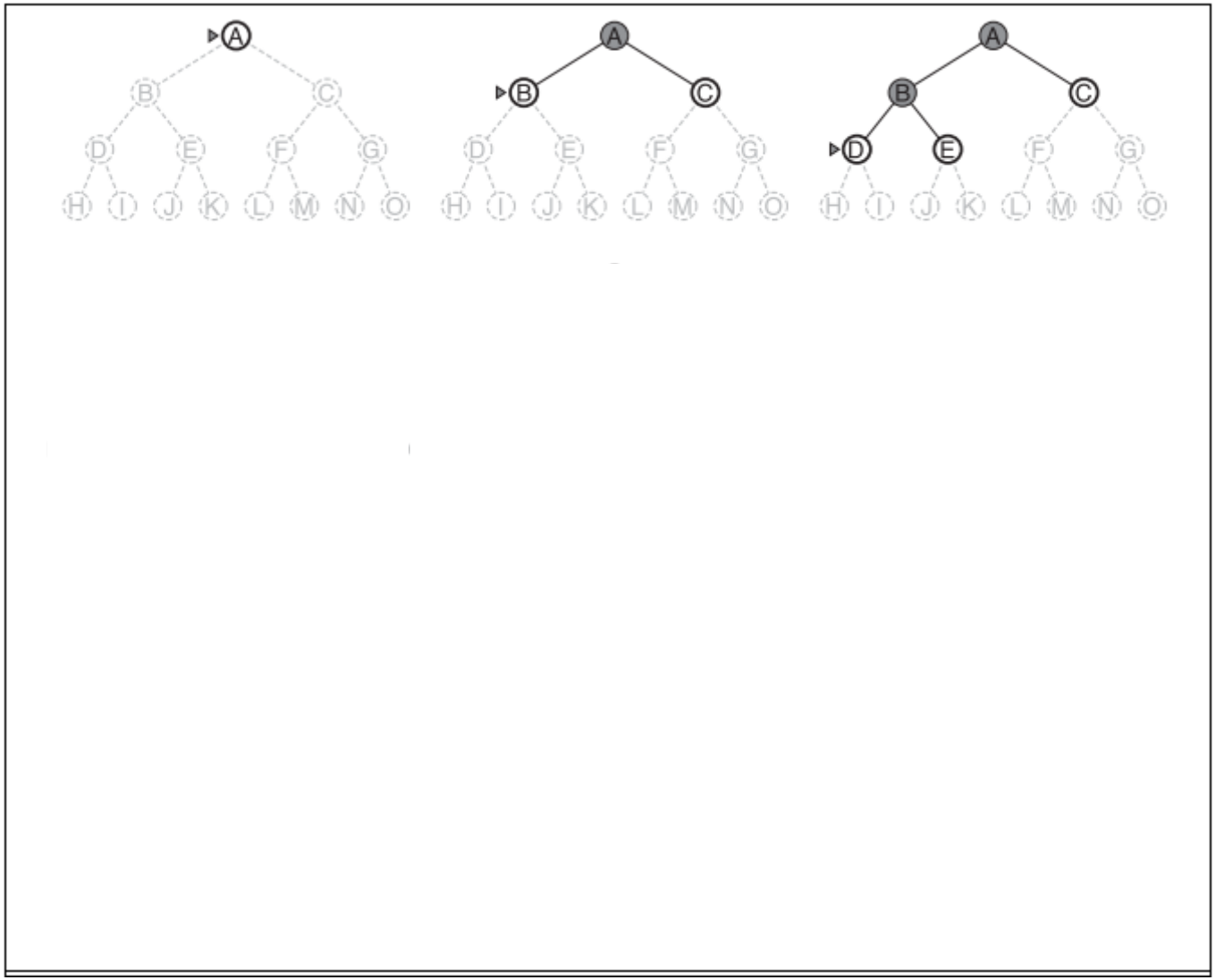
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

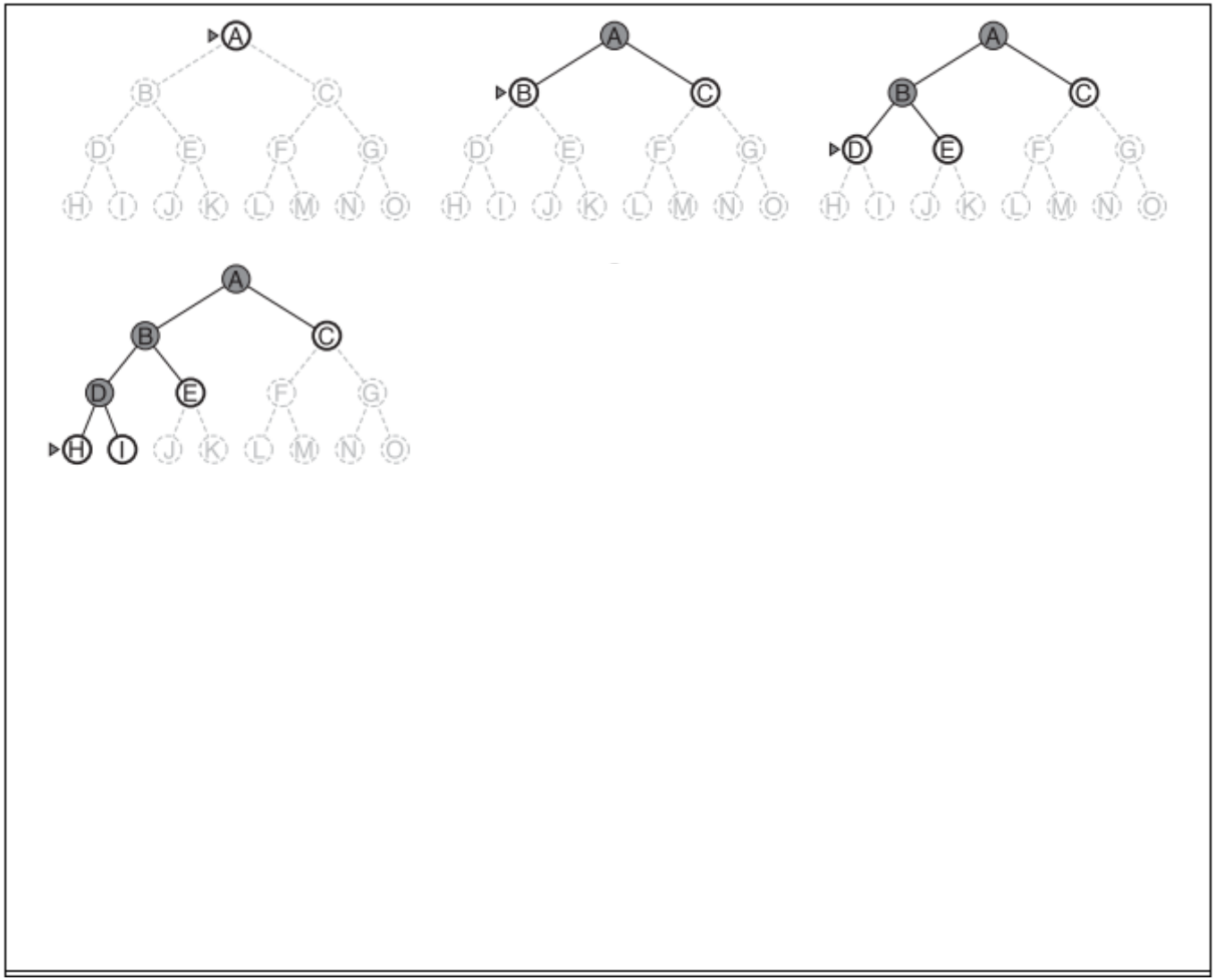
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

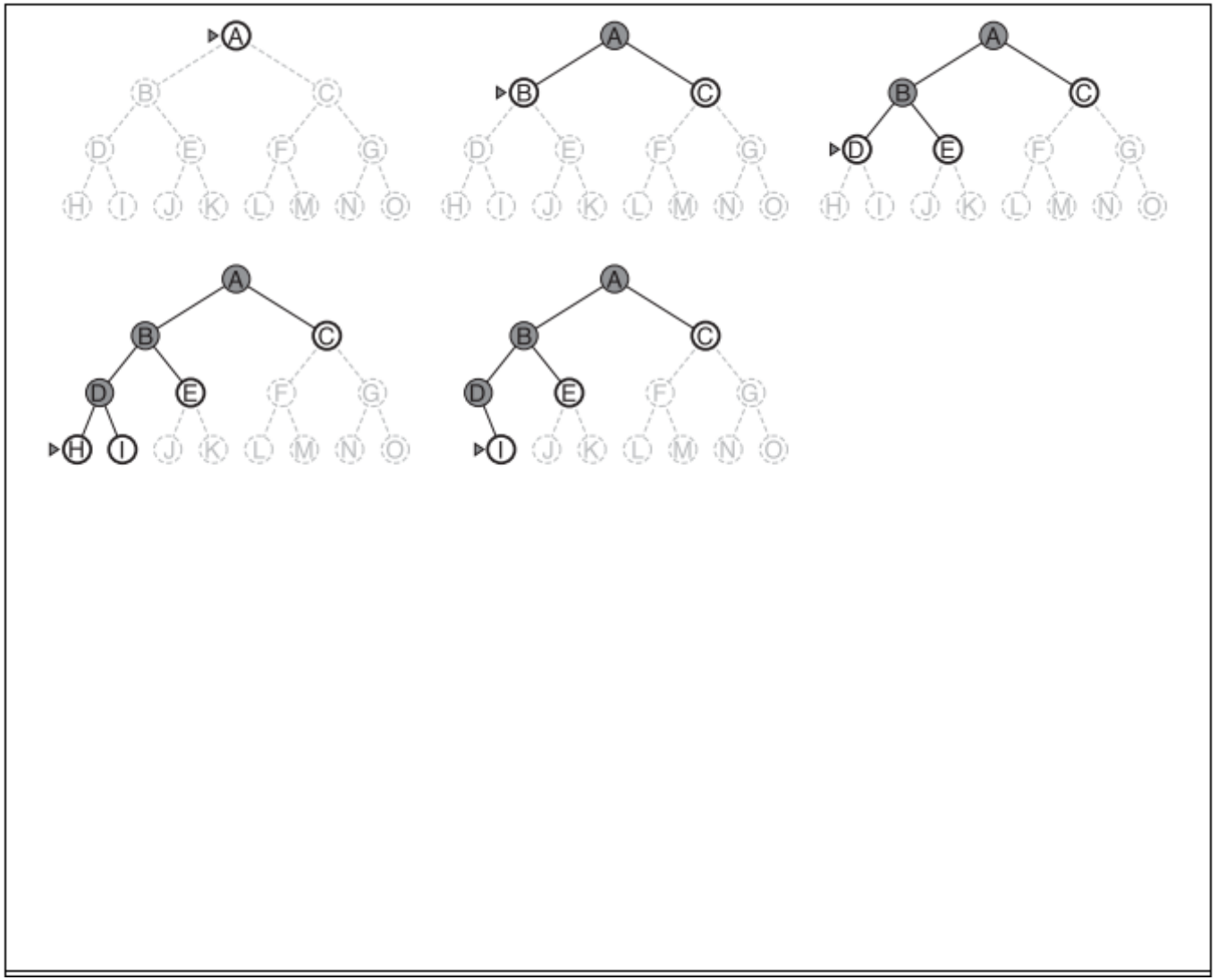
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

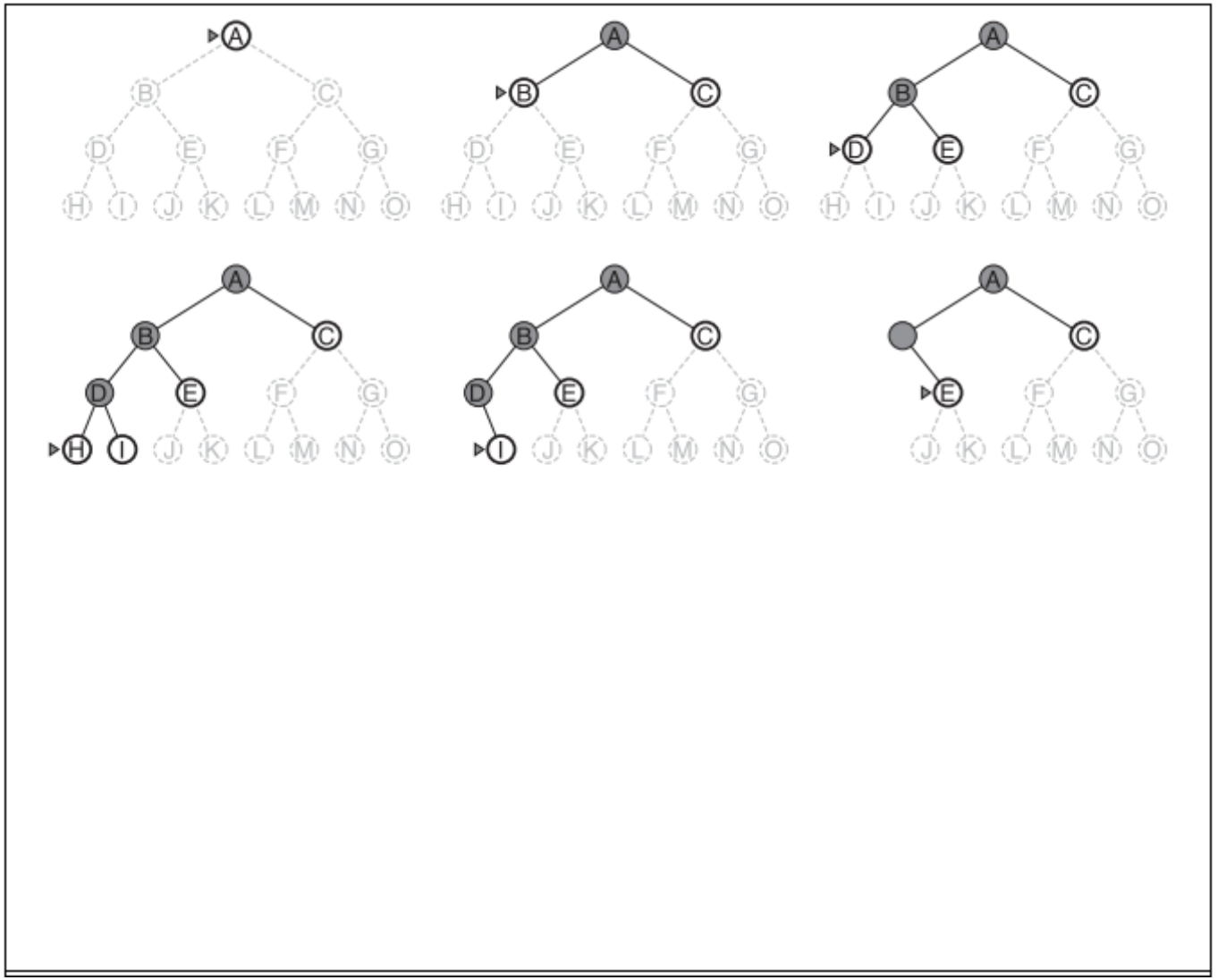
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

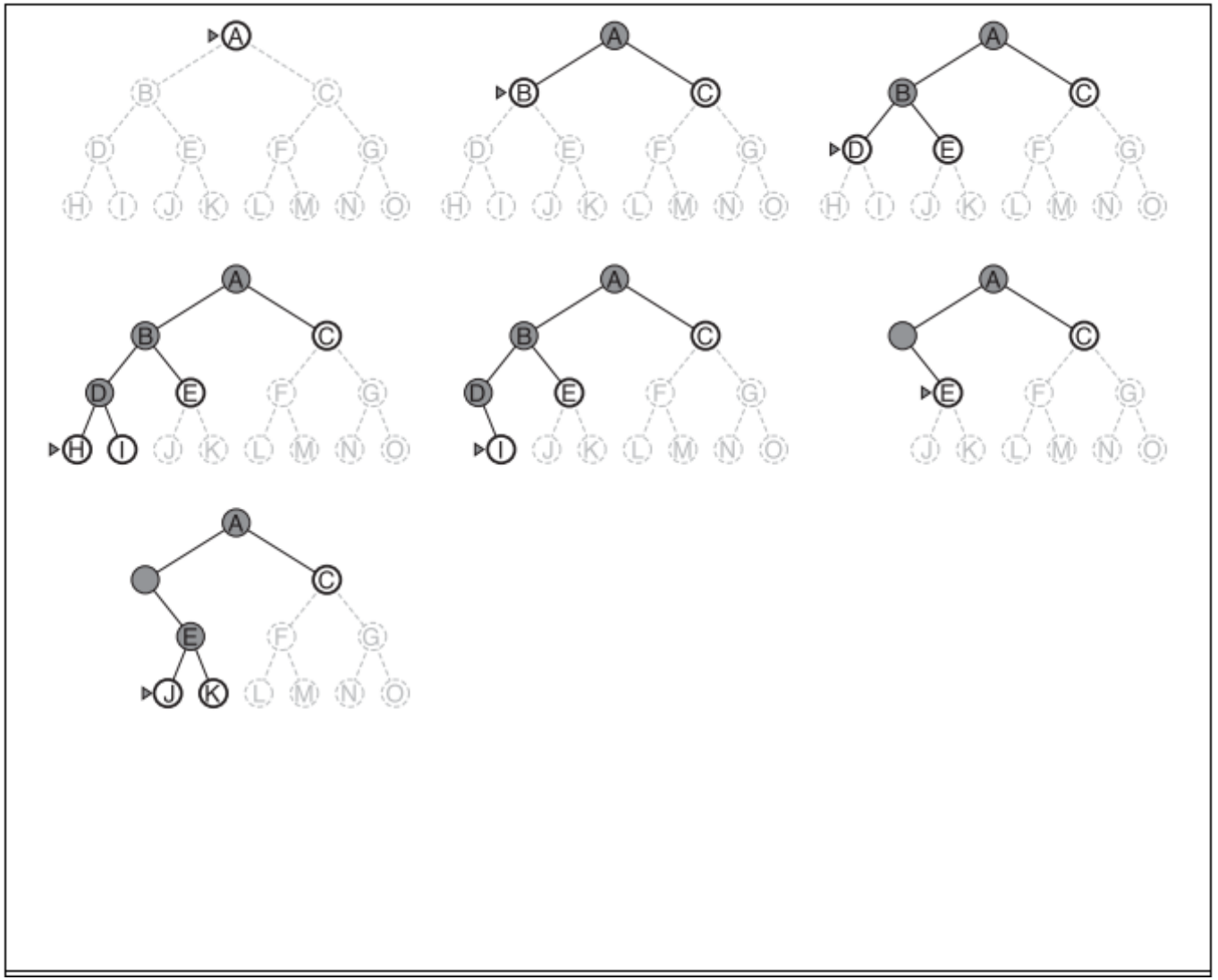
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

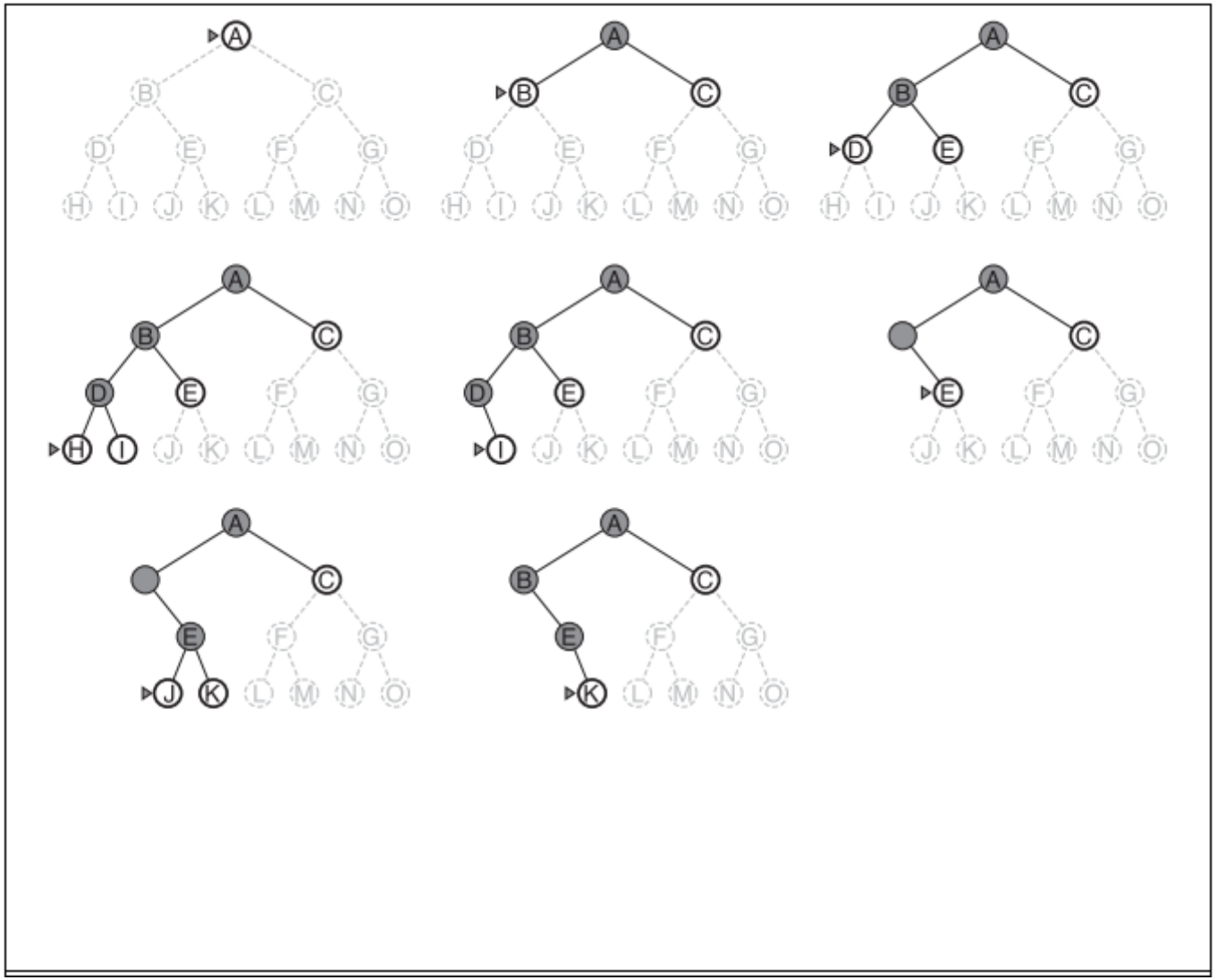
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

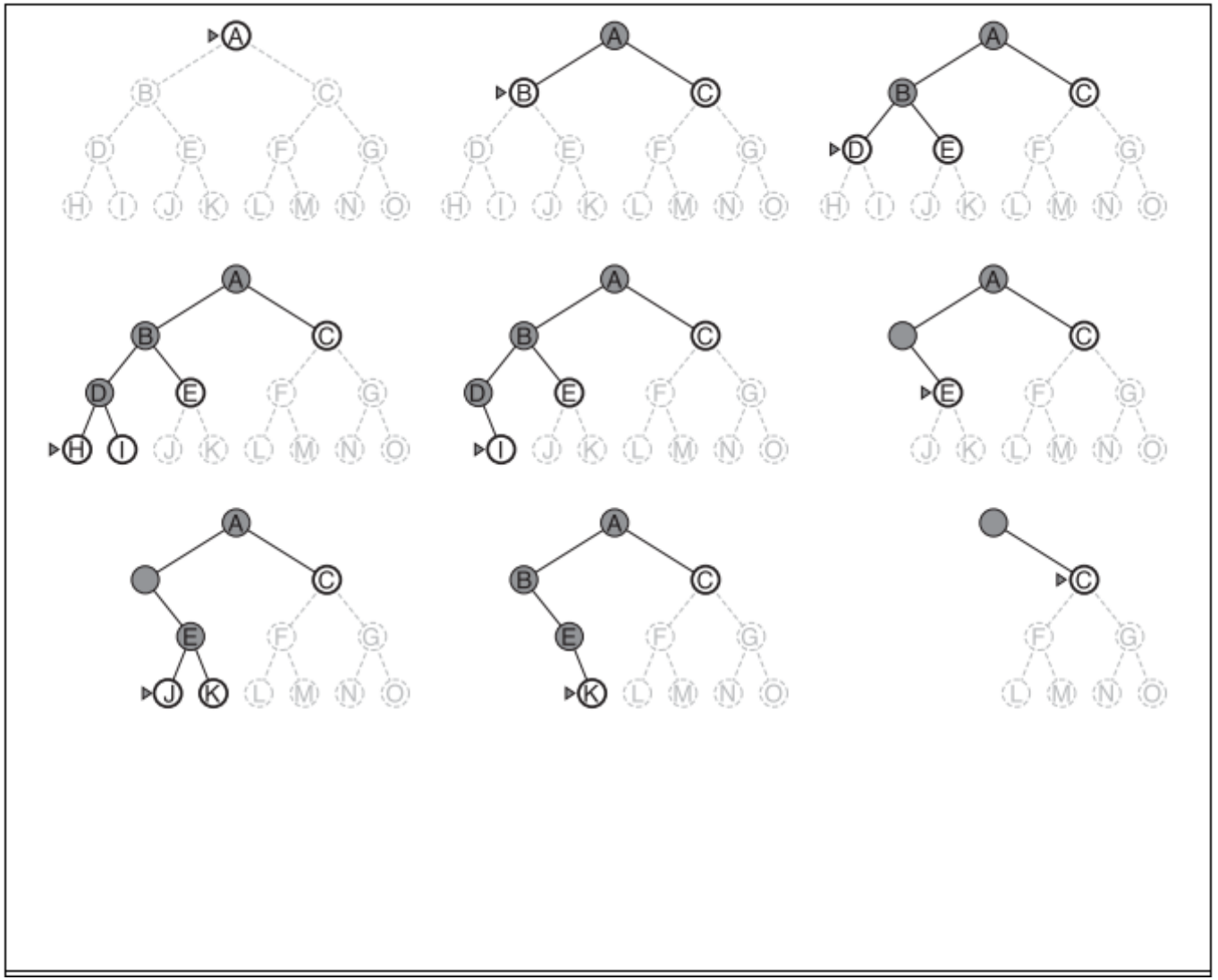
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

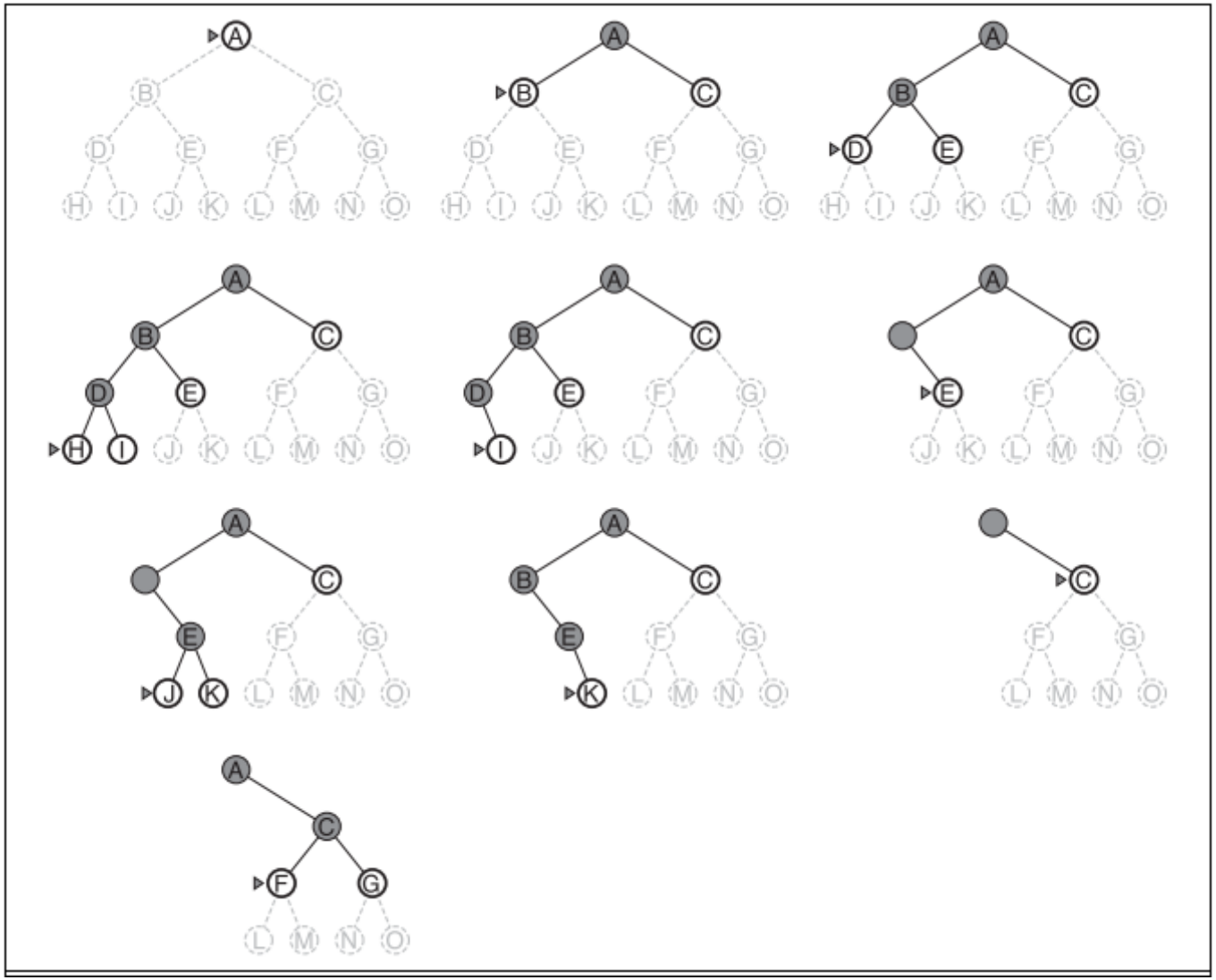
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

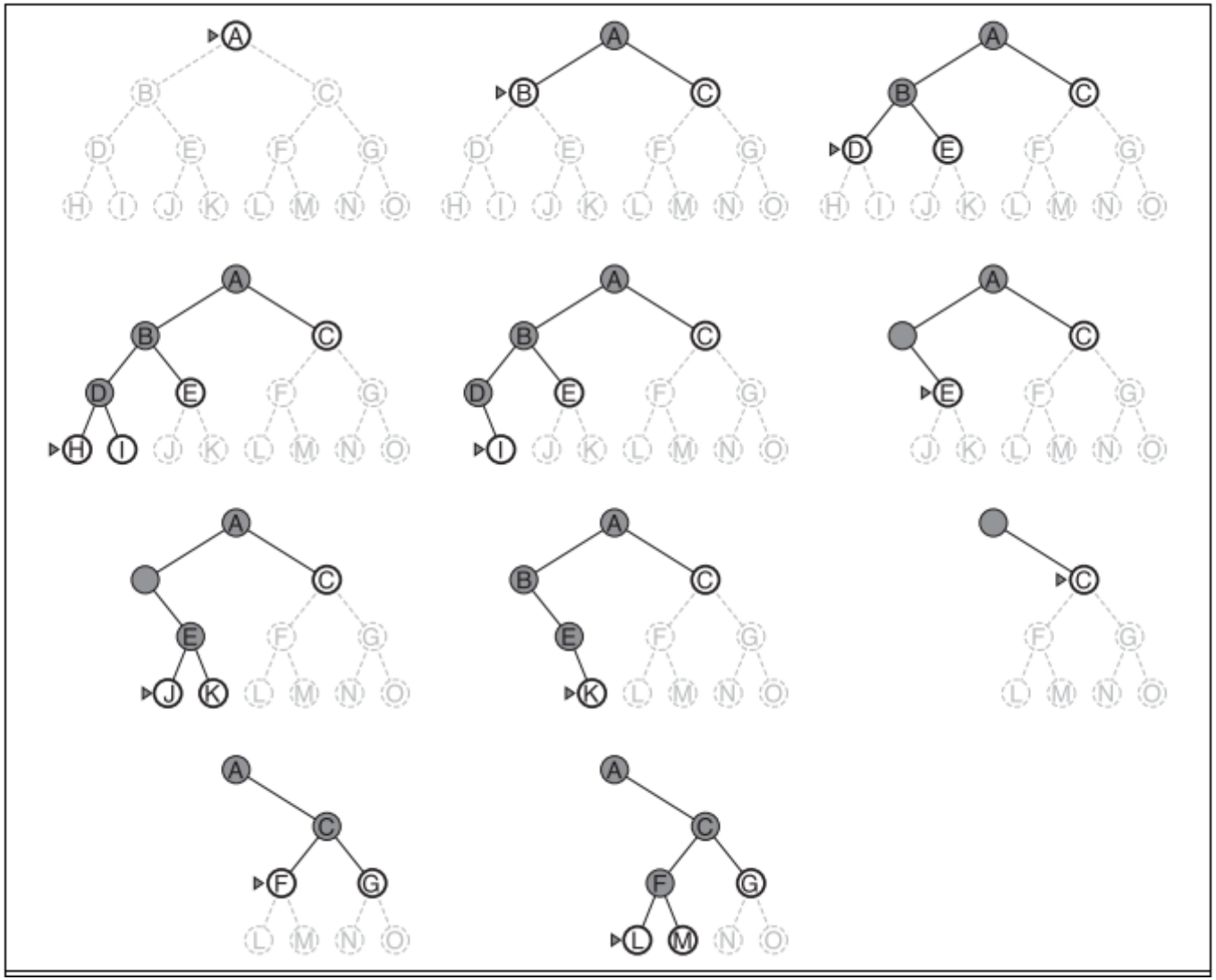
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

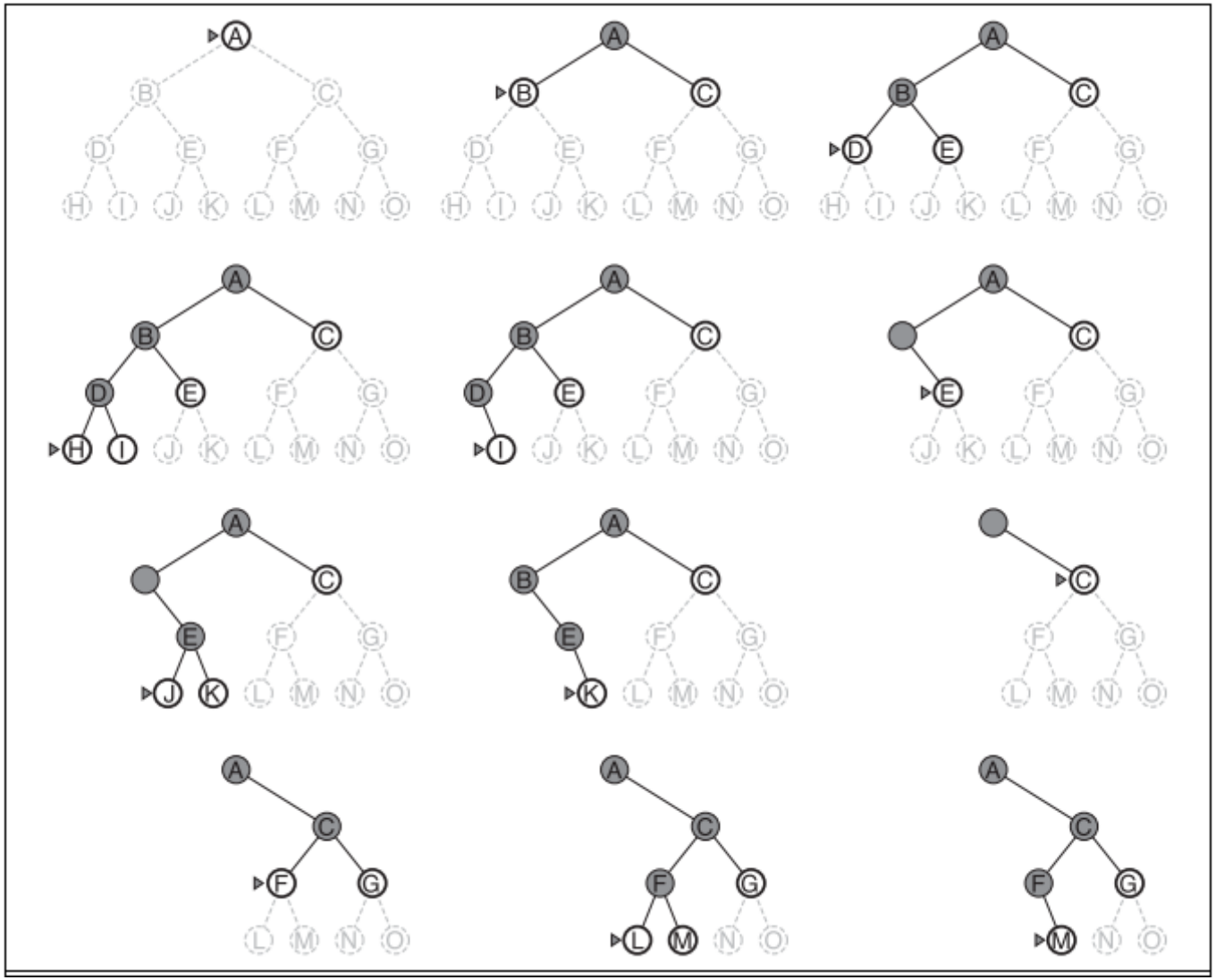
## com Retrocesso




(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade

## com Retrocesso



(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade (Prolog)

```
% resolva(No,Solucao) Solucao é um caminho acíclico (na
  ordem reversa) entre nó inicial No e nó final
resolva(No,Solucao) :-
      depthFirst([],No,Solucao).

% depthFirst(Caminho,No,Solucao) estende o caminho
  [No|Caminho]
% até um nó final obtendo Solucao
depthFirst(Caminho,No,[No|Caminho]) :-
      final(No).
depthFirst(Caminho,No,S) :-
      s(No,No1),
      \+ pertence(No1,Caminho),          % evita um ciclo
      depthFirst([No|Caminho],No1,S).

pertence(E,[E|_]).
pertence(E,[_|T]) :-
      pertence(E,T).
```

# Busca em Profundidade

- Não é completa nem é ótima:

  Esta estratégia deve ser evitada quando as árvores geradas são muito profundas ou geram caminhos infinitos

- Custo de memória:

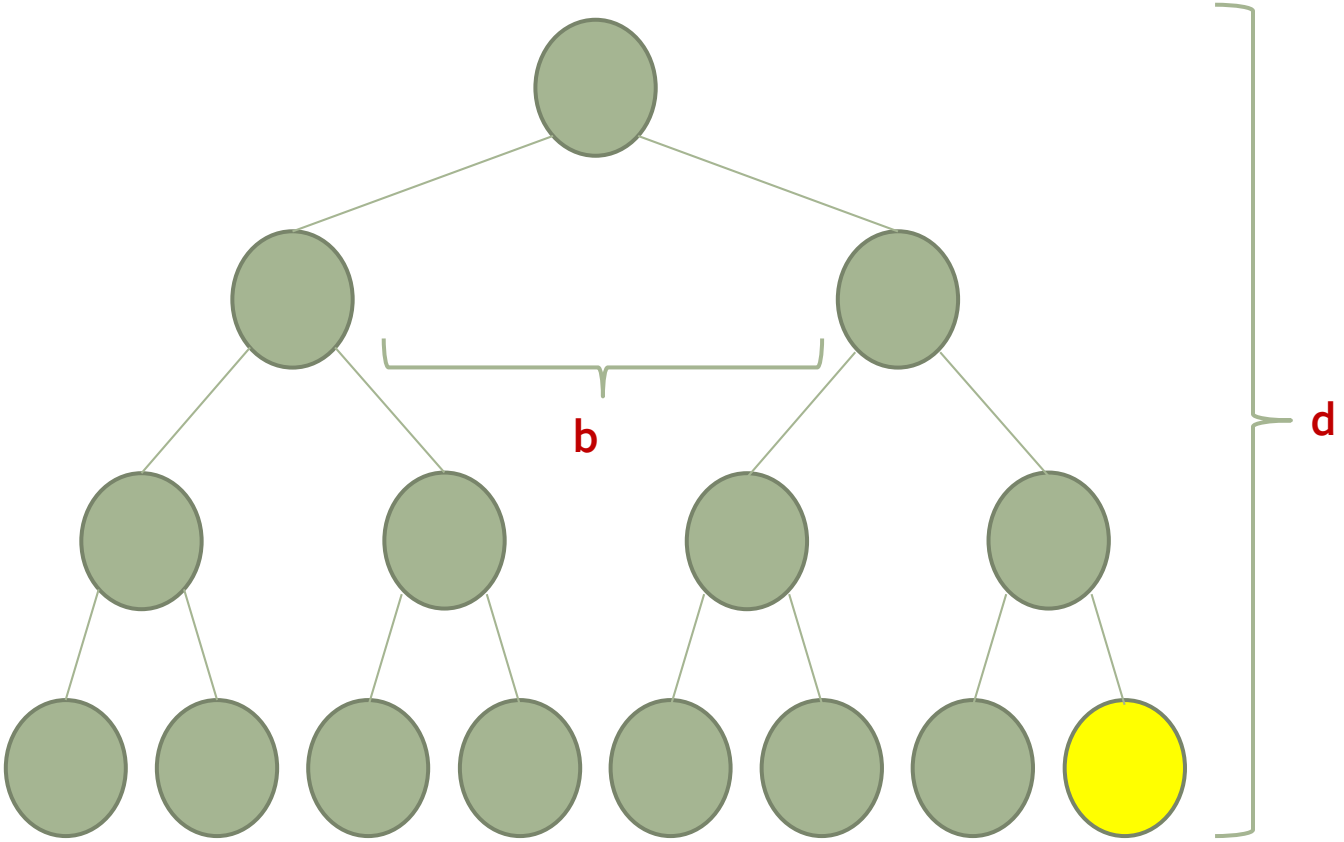
  Necessita armazenar apenas O(b*d) nós para um espaço de estados com fator de ramificação b e profundidade d, onde "m pode ser maior que d" (profundidade da 1a. solução)

- Custo de tempo:

  $O(b^m)$, no pior caso

- Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura

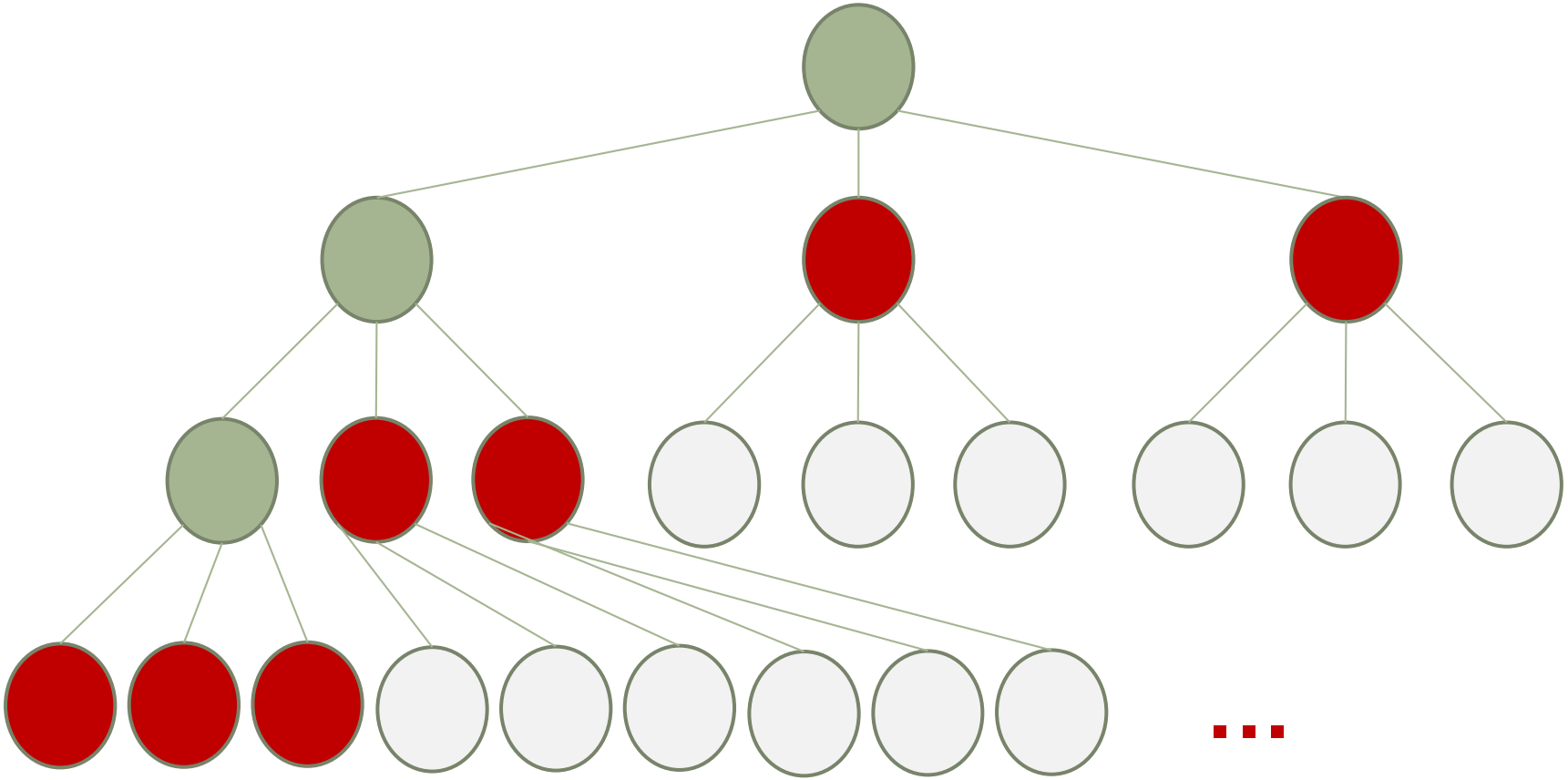
# Busca em Profundidade (Tempo)




Pior caso:

Complexidade de Tempo = $b^d + b^{d-1} + b^{d-2} + \ldots + b^0 =$ **$O(b^d)$,** com d=m

# Busca em Profundidade (Espaço)



Com d=3 e b=3 => 7 nós vermelhos
Em geral = ((b - 1) * d) + 1, pior caso: d = m
Complexidade de Espaço = **O(b*m)**

# Busca em Largura

- Busca em profundidade x Busca em largura: escolhe primeiro visitar aqueles nós mais próximos do nó inicial

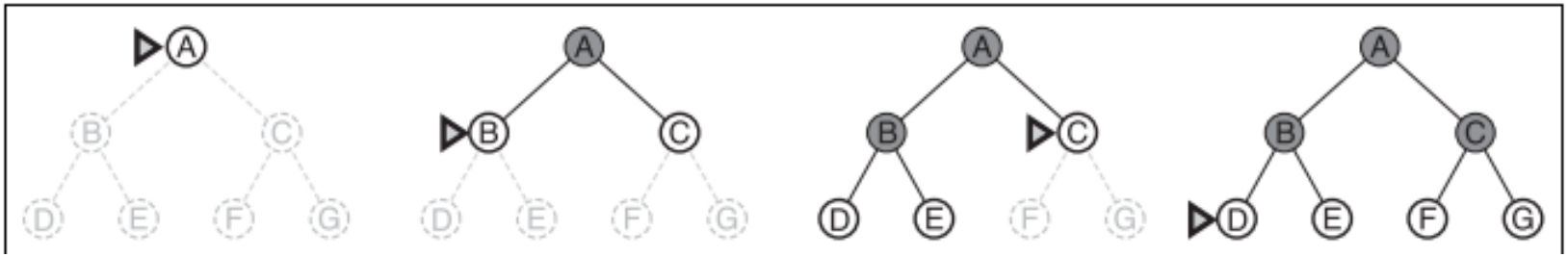
- Algoritmo não é tão simples:
  - necessário controlar um conjunto de nós candidatos alternativos
  - não apenas um único, como na busca em profundidade

# Busca em Largura

- O conjunto de nós é todo o nível inferior da árvore de busca
- Além disso, só o conjunto é insuficiente se o caminho da solução também for necessário
- Assim, ao invés de manter um nó candidato, é necessário manter um conjunto de caminhos candidatos

# Busca em Largura

- Ordem de ramificação dos nós:
  - Nó raiz
  - Todos os nós de profundidade 1
  - Todos os nós de profundidade 2
  - Assim por diante




(Fonte: Russel & Norvig, 2010)

# Busca em Largura

**Algoritmo:**

```
função Busca-em-Largura (problema)
    retorna uma solução ou falha
    Busca-Genérica (problema,
                    Insere-no-Fim)
```

***Insere-no-Fim***: estratégia de lista?

# Busca em Largura

- Esta estratégia é completa
- É ótima ?
  - Sempre encontra a solução mais "rasa"
  - No entanto, nem sempre é a solução de menor custo de caminho, caso os operadores tenham valores diferentes
    - Exemplo: ir para uma cidade D passando por B e C pode ser mais perto do que passando só por E
- Em outras palavras, é ótima se custo de caminho cresce com a profundidade do nó
- Isso ocorre quando todos os operadores têm o mesmo custo (=1)

# Busca em Largura

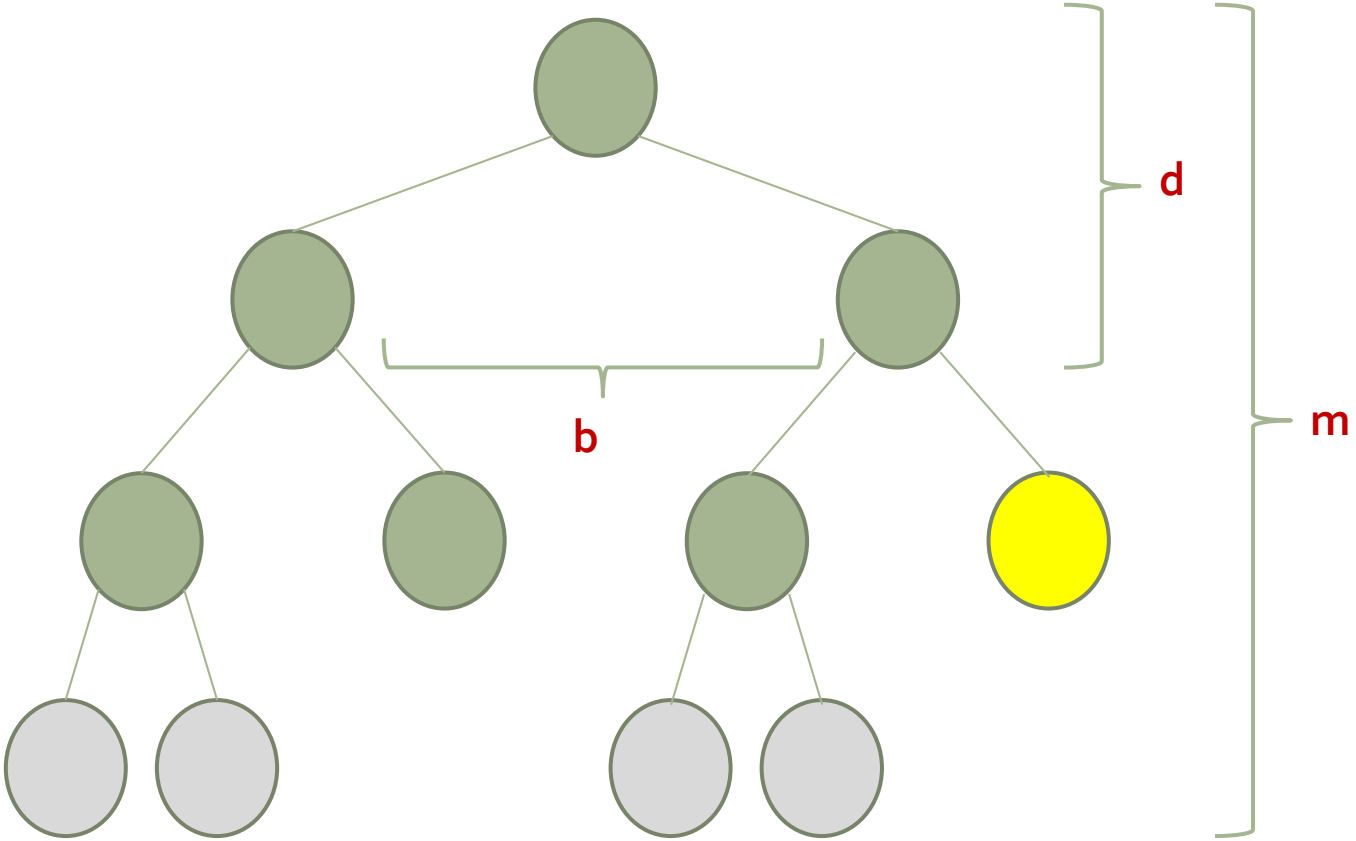
- Custo de tempo:
  - Considerando o fator de ramificação do problema = b, e a primeira solução para o problema está no nível d
  - Então o número máximo de nós gerados até se encontrar a solução é

$$1 + b + b^2 + b^3 + \ldots + b^d$$

  - Custo exponencial = $O(b^d)$
- Custo de memória:
  - Problema: a fronteira do espaço de estados deve permanecer na memória = $O(b^d)$
  - Logo, busca em largura só retorna bons resultados quando a profundidade da árvore de busca é relativamente pequena

# Busca em Largura

(Tempo e Espaço)




Complexidades de Tempo e Espaço: **$O(b^d)$**

# Busca em Largura

## (Tempo e Espaço)

| 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 |

**Figure 3.13** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

(Fonte: Russel & Norvig, 2010)

# Busca em Profundidade Limitada

- Um limite máximo (L) de profundidade para os caminhos gerados
- L >= d, onde L é o limite de profundidade e d é a profundidade da primeira solução do problema
- Evita o problema de caminhos muito longos ou infinitos impondo
- Espaço: O(b*L)
- Tempo: $O(b^L)$

- **PROBLEMA?**

# Busca em Profundidade Limitada

- Não se tem previamente um limite razoável

- Se o limite for muito pequeno (menor que qualquer caminho até uma solução) ?

  então a busca falha

- Se o limite for muito grande?

  a busca se torna muito complexa

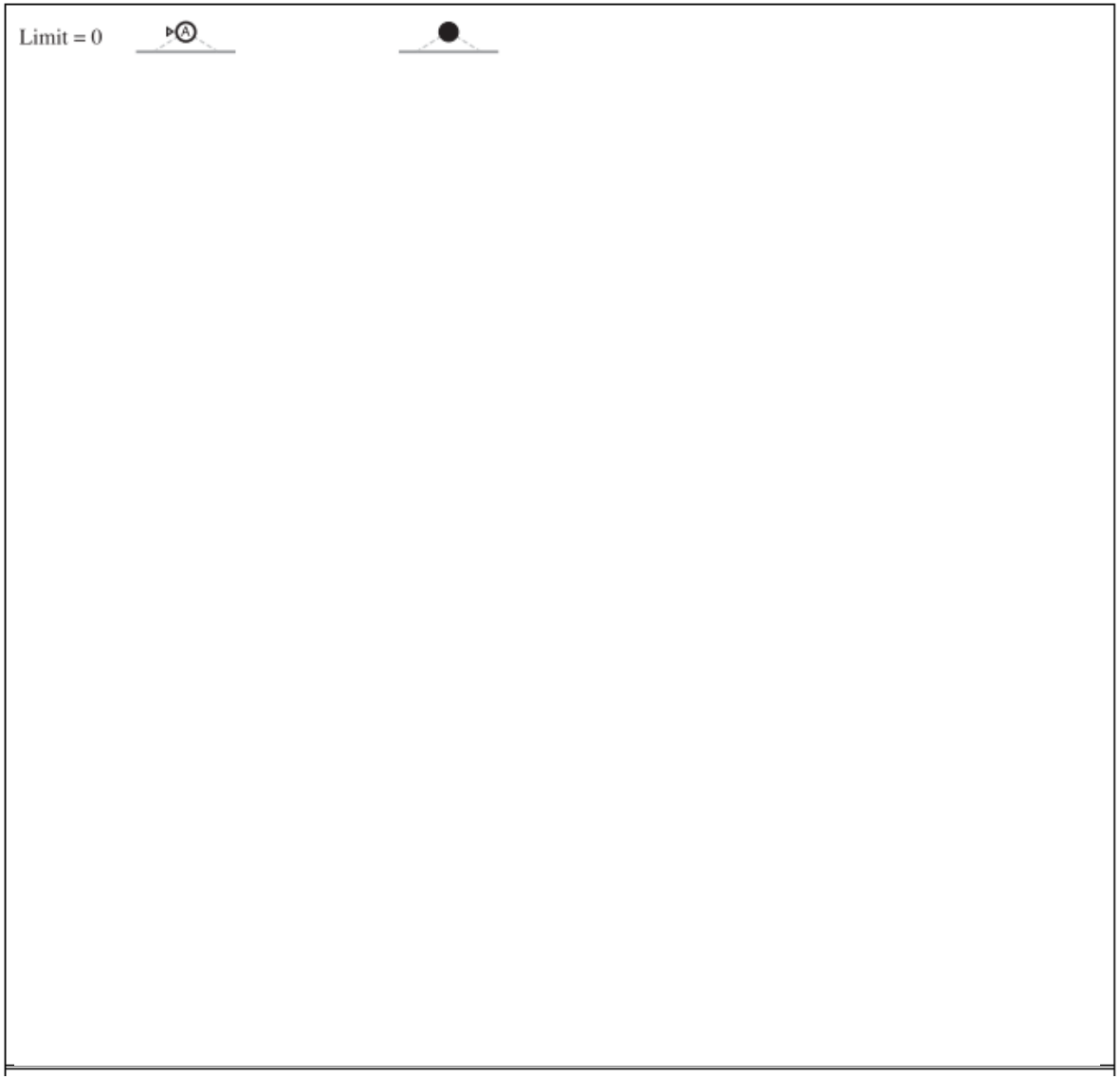
# Busca com Aprofundamento Iterativo

- Tentam-se limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - Fixa-se profundidade = i, executa busca
  - Se não chegou a um objetivo, recomeça busca com profundidade = i + n (n qualquer)
- Tempo de busca piora, porém melhora o custo de memória! Mesmo assim, pode ser boa alternativa

# Busca com Aprofundamento Iterativo

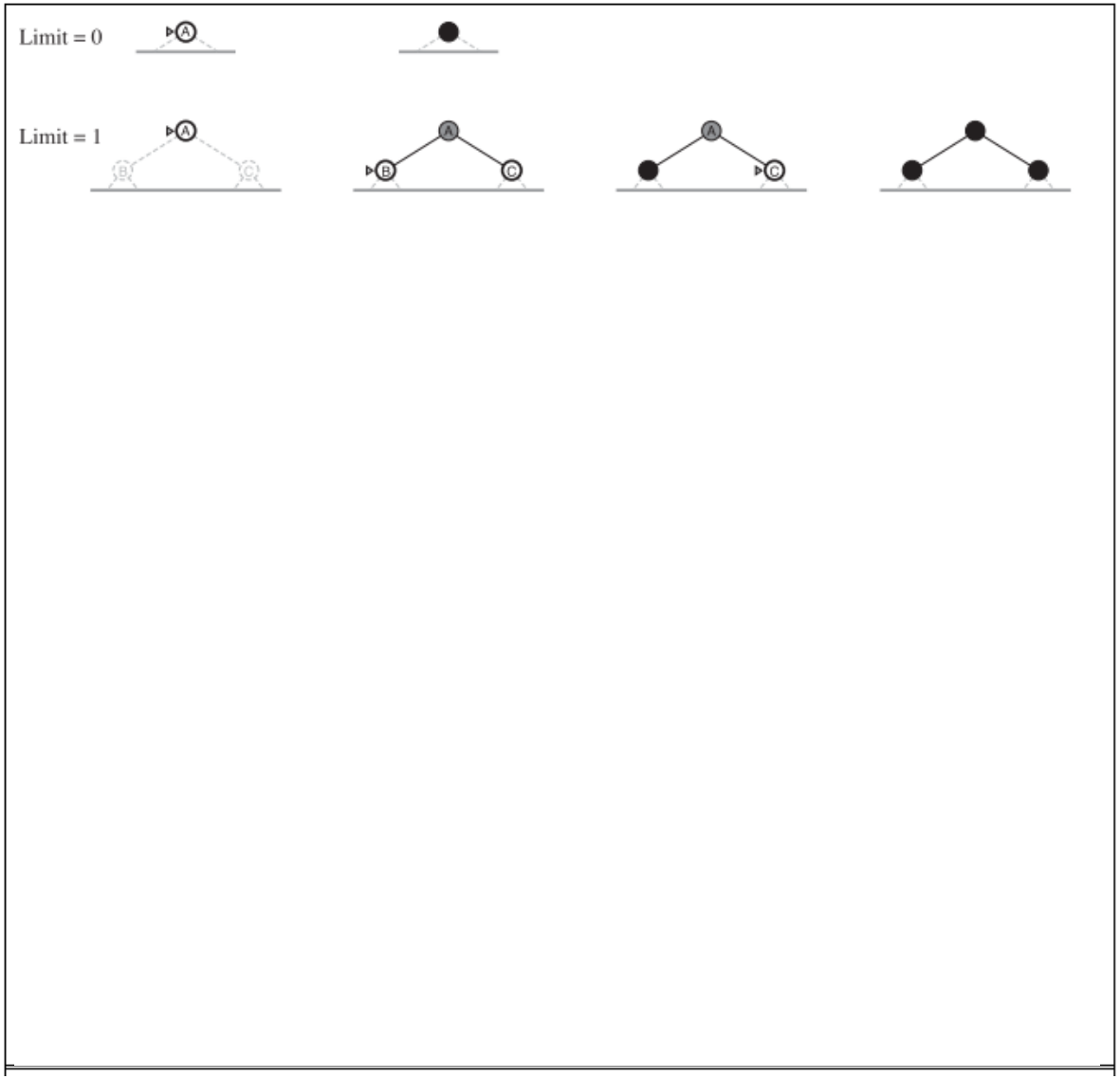
- Combina as vantagens de busca em largura com busca em profundidade
  - É ótima e completa
  - Com n = 1 e operadores com custos iguais
  - Custo de memória: O(b*d)
  - Custo de tempo: $O(b^d)$
- Bons resultados quando o espaço de estados é grande e de profundidade desconhecida

# Busca com Aprofundamento Iterativo



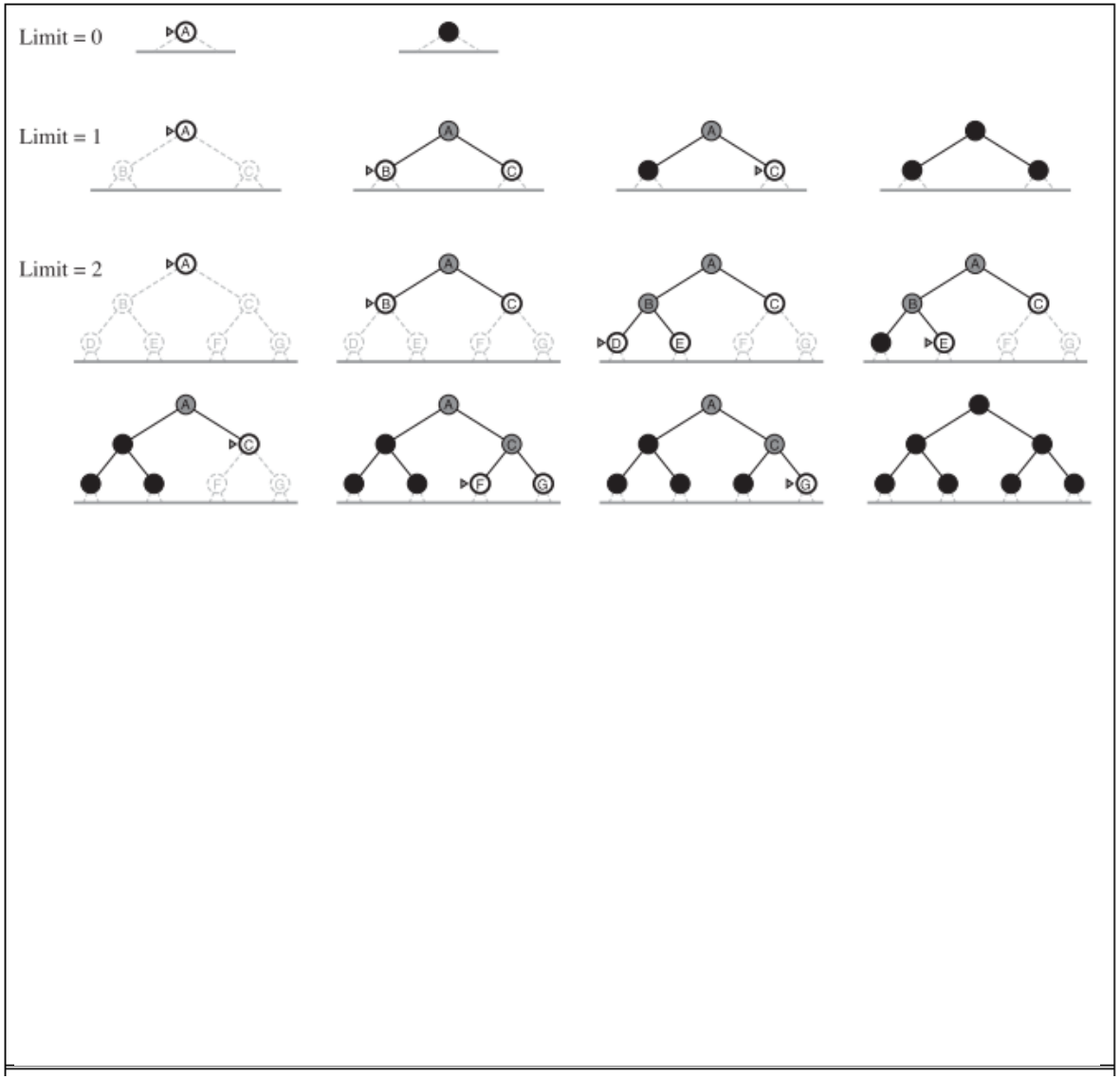

(Fonte: Russel & Norvig, 2010)

# Busca com Aprofundamento Iterativo



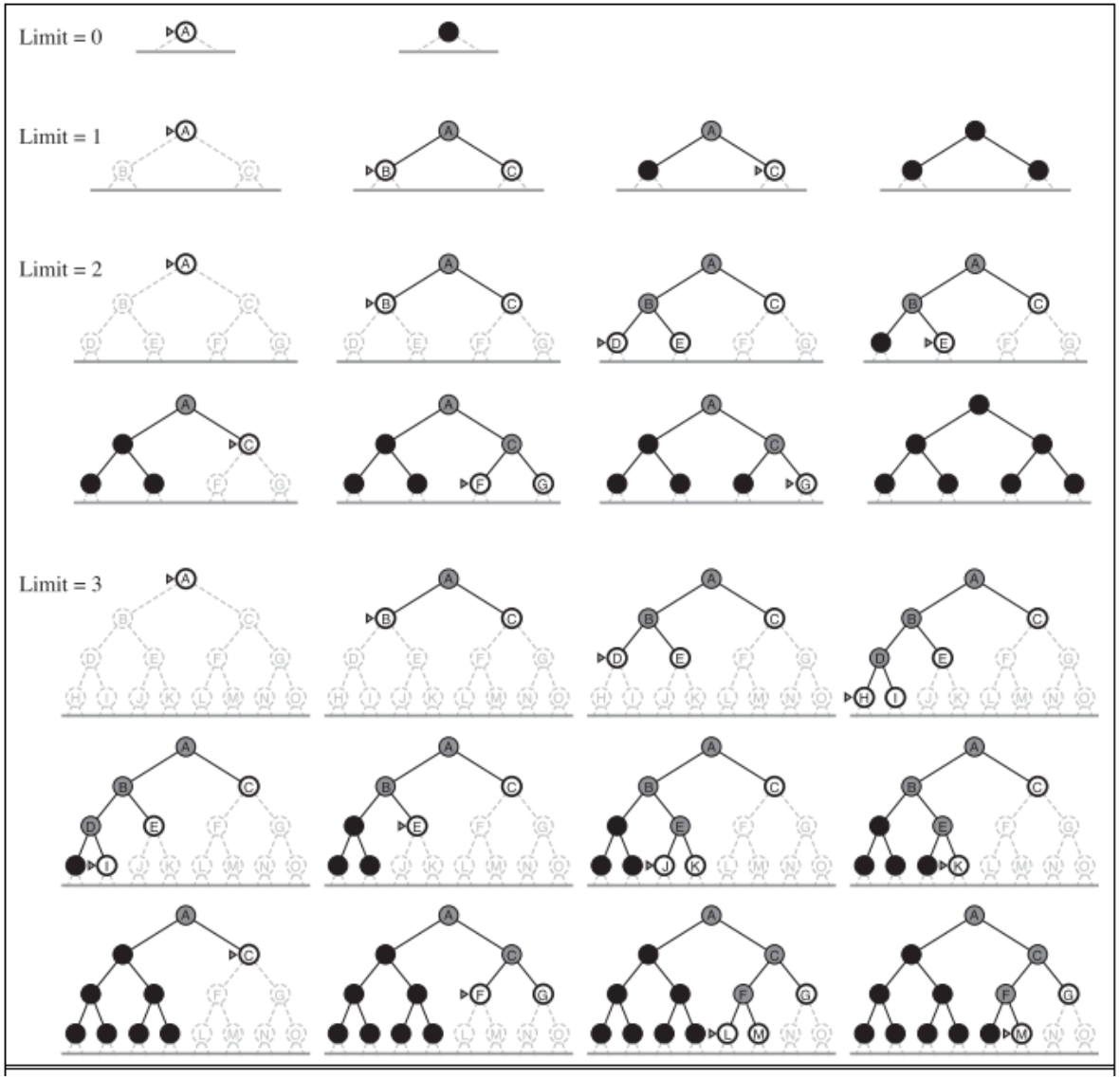

(Fonte: Russel & Norvig, 2010)

# Busca com Aprofundamento Iterativo




(Fonte: Russel & Norvig, 2010)

# Busca com Aprofundamento Iterativo




(Fonte: Russel & Norvig, 2010)

# Busca de Custo Uniforme

(Dijkstra's Search)

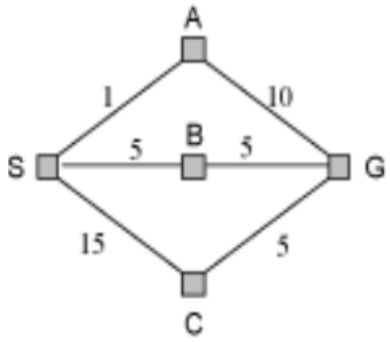
- Extensão da busca em largura:
  - Expande-se o nó da fronteira com menor custo de caminho até o momento
  - Cada operador pode ter um custo associado diferente, medido pela função **g(n)** que é resultante do custo do caminho da origem ao nó n
- Na busca em largura: g(n) = profundidade (n)

**Algoritmo:**

```
função Busca-de-Custo-Uniforme (problema)
    retorna uma solução ou falha
    Busca-Genérica (problema,
                    Insere-Ordem-Crescente)
```

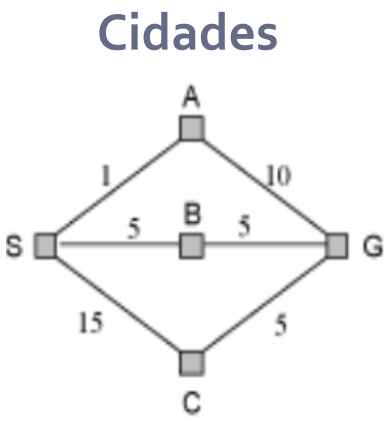
# Busca de Custo Uniforme

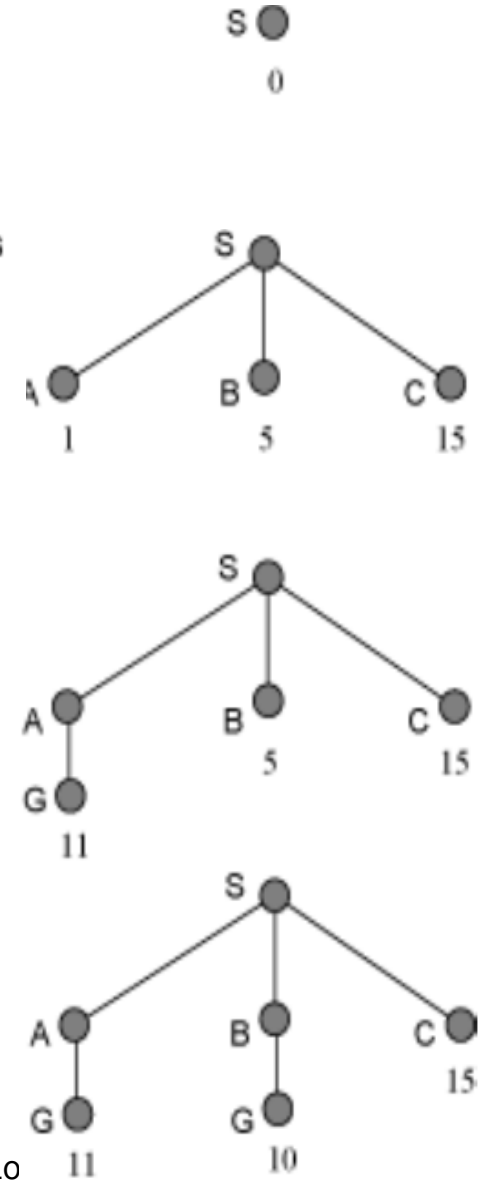
## Cidades




(Fonte: Russel & Norvig, 2010)

# Busca de Custo Uniforme




**F = {S}**
S é o estado objetivo? Caso não, expande-o e guarda seus filhos A, B e C ordenadamente na fronteira

**F = {A, B, C}**
A é o estado objetivo? Caso não, expande-o e guarda seu filho GA ordenadamente

obs.: o algoritmo de geração e teste guarda na fronteira todos os nós gerados, testando se um nó é o objetivo apenas quando ele é retirado da lista!

**F= {B, GA, C}**
B é o estado objetivo? Caso não, expande-o e guarda seu filho GB ordenadamente

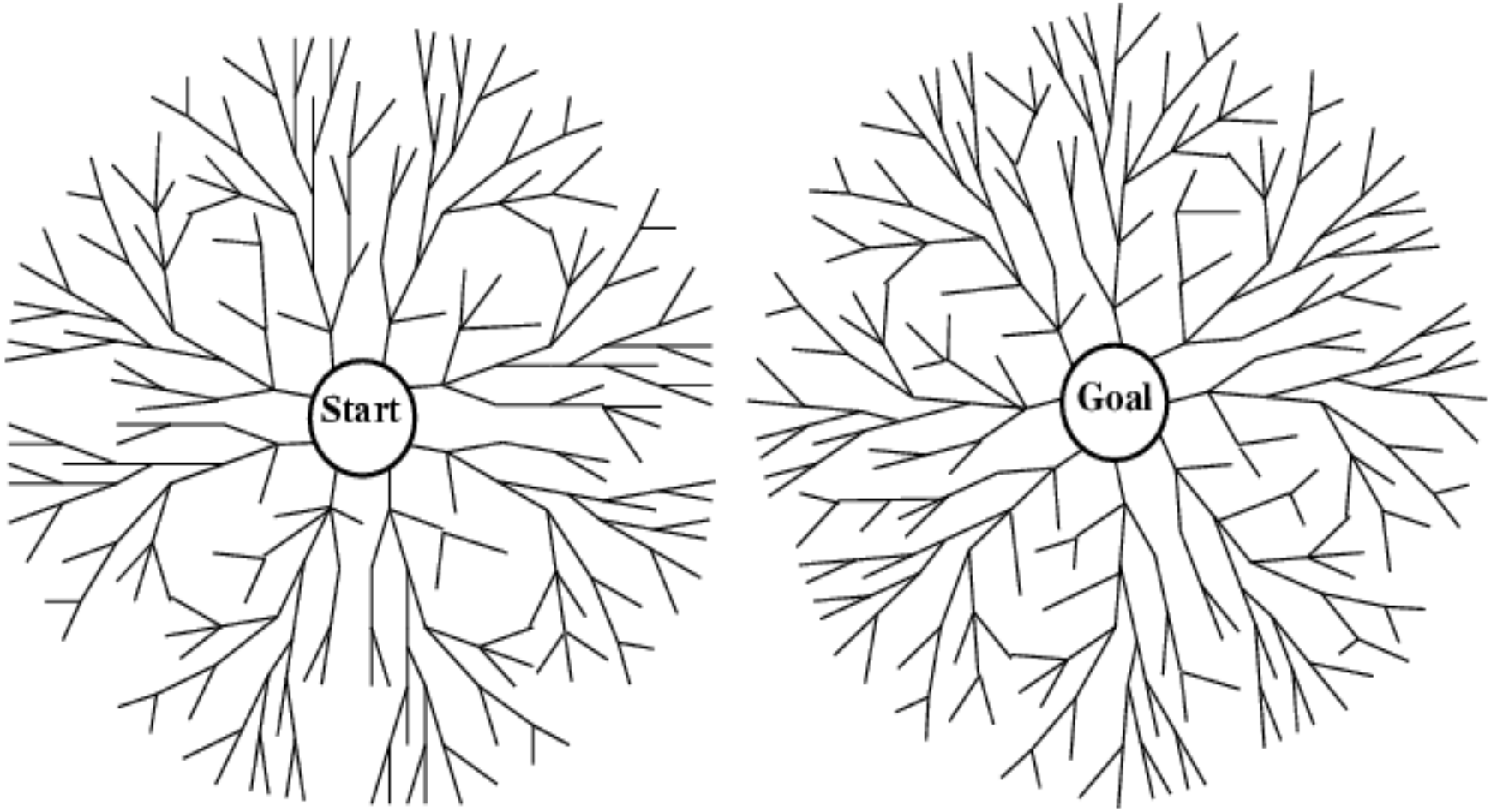
**F= {GB, GA, C}**
GB é o estado objetivo? Sim e para!

(Fonte: Russel & Norvig, 2010

# Busca de Custo Uniforme

- Esta estratégia é completa
- É ótima se
  - g (sucessor(n)) >= g (n)
    - custo de caminho no mesmo caminho não decresce
    - i.e., não tem operadores com custo negativo
  - Caso contrário, teríamos que expandir todo o espaço de estados em busca da melhor solução
    - Exemplo: seria necessário expandir também o nó C do exemplo, pois o próximo operador poderia ter custo associado = -13, por exemplo, gerando um caminho mais barato do que através de B
- Custo de tempo e de memória: teoricamente, igual ao da Busca em Largura

# Busca Bidirecional




(Fonte: Russel & Norvig, 2010)

# Complexidade dos Algoritmos de Busca

- b = número de caminhos alternativos/fator de bifurcação/ramificação (*branching* factor)
- m = profundidade da solução
- d = profundidade máxima da árvore de busca
- L= limite de profundidade

| | Tempo | Espaço | Completa? (encontra uma solução quando ela existe) | Ótima? (solução mais curta garantida) |
|---|---|---|---|---|
| Profundidade | $O(b^m)$ | O(bm) | Sim (espaços finitos)<br>Não (espaços infinitos) | Não |
| Profundidade limitada | $O(b^L)$ | O(bL) | Sim se L ≥ d | Não |
| Profundidade iterativa | $O(b^d)$ | O(bd) | Sim | Sim |
| Largura | $O(b^d)$ | $O(b^d)$ | Sim | Sim |
| Bidirecional | $O(b^{d/2})$ | $O(b^{d/2})$ | Sim | Sim |

# Evitar Geração de Estados Repetidos

- Problema geral em busca
  - Expandir estados presentes em caminhos já explorados
- É inevitável quando existem operadores reversíveis
  - Exemplos: encontrar rotas, canibais e missionários, 8-números, entre outros
  - A árvore de busca é potencialmente infinita
- Três soluções com diferentes níveis de eficácia e custo de implementação...

# Evitar Estados Repetidos: soluções

1. Não retornar ao estado "pai"
2. Não retornar a um ancestral
3. Não gerar qualquer estado que já tenha sido criado antes (em qualquer ramo)
   - requer que todos os estados gerados permaneçam na memória: custo $O(b^d)$
   - pode ser implementado mais eficientemente com *hash tables*

# Referências


- Material Didático Prof. Huei Diana Lee – Unioeste.
- Material Didático Prof. José Augusto Baranauskas – USP Ribeirão Preto.
- Material Didático Prof. Marcílio Souto – UFRN.
- Russel, S. e Norvig, P. Artificial Intelligence: A modern approach, Prentice Hall, 2010.
- Outras referências indicadas no curso.