

# Tipos de Busca (Busca Cega)

Capítulo 3 – Russell & Norvig

# Ao final desta aula a gente deve saber:

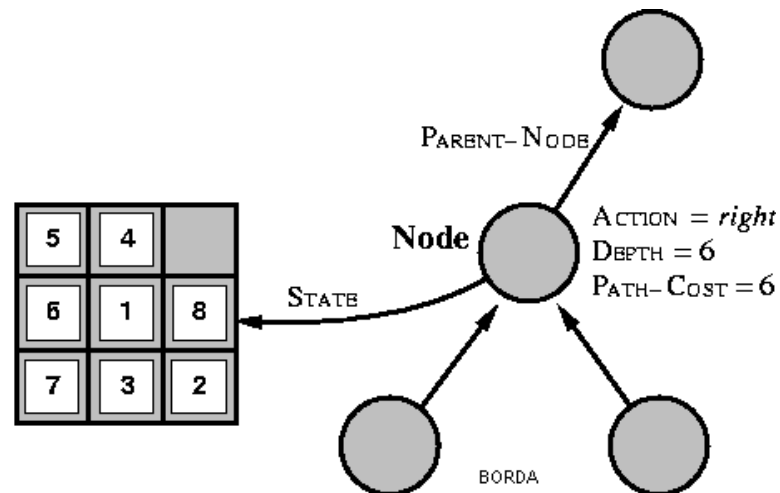
- Conhecer as várias estratégias de realizar busca não-informada (Busca Cega)
- Determinar que estratégia se aplica melhor ao problema que queremos solucionar
- Evitar a geração de estados repetidos.

## Aula passada (Busca de soluções)

- Percorrer o espaço de estados a partir de uma ***árvore de busca***;
- *Expandir* o estado atual aplicando a função sucessor, *gerando* novos estados;
- Busca: seguir um caminho, deixando os outros para depois;
- *A estratégia de busca* determina qual caminho seguir.

# Aula passada (Implementação do algoritmo)

- Os nós da fronteira devem guardar mais informação do que apenas o estado:
  - Na verdade nós são uma **estrutura de dados** com 5 componentes:
    - o estado (configuração) correspondente ao nó atual
    - o seu nó pai – **ou o caminho inteiro para não precisar de operações extras**
    - a ação aplicada ao pai para gerar o nó – **verifica de onde veio para evitar loops**
    - o custo do nó desde a raiz (  $g(n)$  )
    - a profundidade do nó – **se guardar o caminho não precisa!**



# Aula passada (Busca em Espaço de Estados)

Função **Busca-Genérica** (*problema formulado*, Função-Inserir)

retorna **uma solução** ou **falha**

*fronteira*  $\leftarrow$  Estado-Inicial (*problema*)

loop do

se *fronteira* está vazia então retorna **falha**

*nó*  $\leftarrow$  Remove-Primeiro (*fronteira*)

se Teste-Término (*problema*, *nó*) tiver sucesso

então **retorna** *nó*

*fronteira*  $\leftarrow$  Função-Inserir (*fronteira*, Ações (*nó*) )

end

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

# Aula passada (Métodos de Busca)

- Busca exaustiva (cega)
  - Não sabe qual o melhor nó da fronteira a ser expandido
    - i.e., menor custo de caminho desse nó até um nó final (objetivo).
  - Estratégias de Busca (ordem de expansão dos nós):
    - Busca em largura
    - Busca em profundidade
- Busca heurística (informada)
  - Estima qual o melhor nó da fronteira a ser expandido com base em funções heurísticas => conhecimento

# Aula passada (Avaliação das estratégias de busca)

- Completude (completeza):
  - a estratégia **sempre** encontra uma solução quando existe alguma?
- Custo do tempo:
  - quanto **tempo** gasta para encontrar uma solução?
- Custo de memória:
  - quanta **memória** é necessária para realizar a busca?
- Qualidade/otimalidade (*optimality*):
  - a estratégia encontra **a melhor solução** quando existem soluções diferentes?
    - menor custo de caminho

# Outras análises de algoritmos de busca

- Fator de ramificação: **b** (número máximo de sucessores de qualquer nó);
- Profundidade do nó objetivo menos profundo: **d**
  - tempo: medido em termos do número de nós gerados durante a busca
  - espaço: número máximo de nós armazenados.



# Busca Cega (Exaustiva)

- Estratégias para determinar a ordem de expansão dos nós
  1. Busca em largura
  2. Busca de custo uniforme
  3. Busca em profundidade
  4. Busca com aprofundamento iterativo

# Busca em Largura

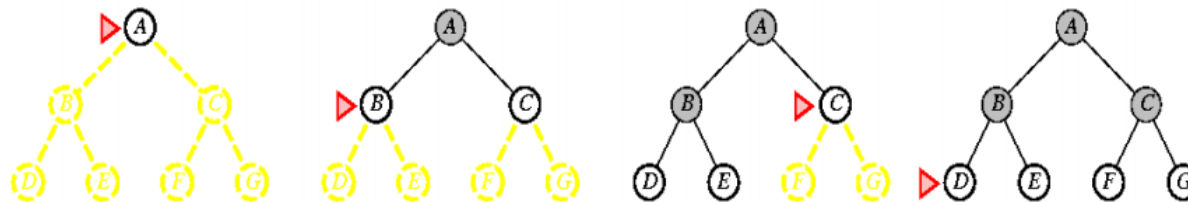
- O nó raiz é expandido primeiro e, em seguida, todos os sucessores dele, depois todos os sucessores desses nós
  - Ou seja, todos os nós em uma dada **profundidade** são expandidos antes de todos os nós do nível seguinte.
- Ordem de expansão dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc...

- Algoritmo:

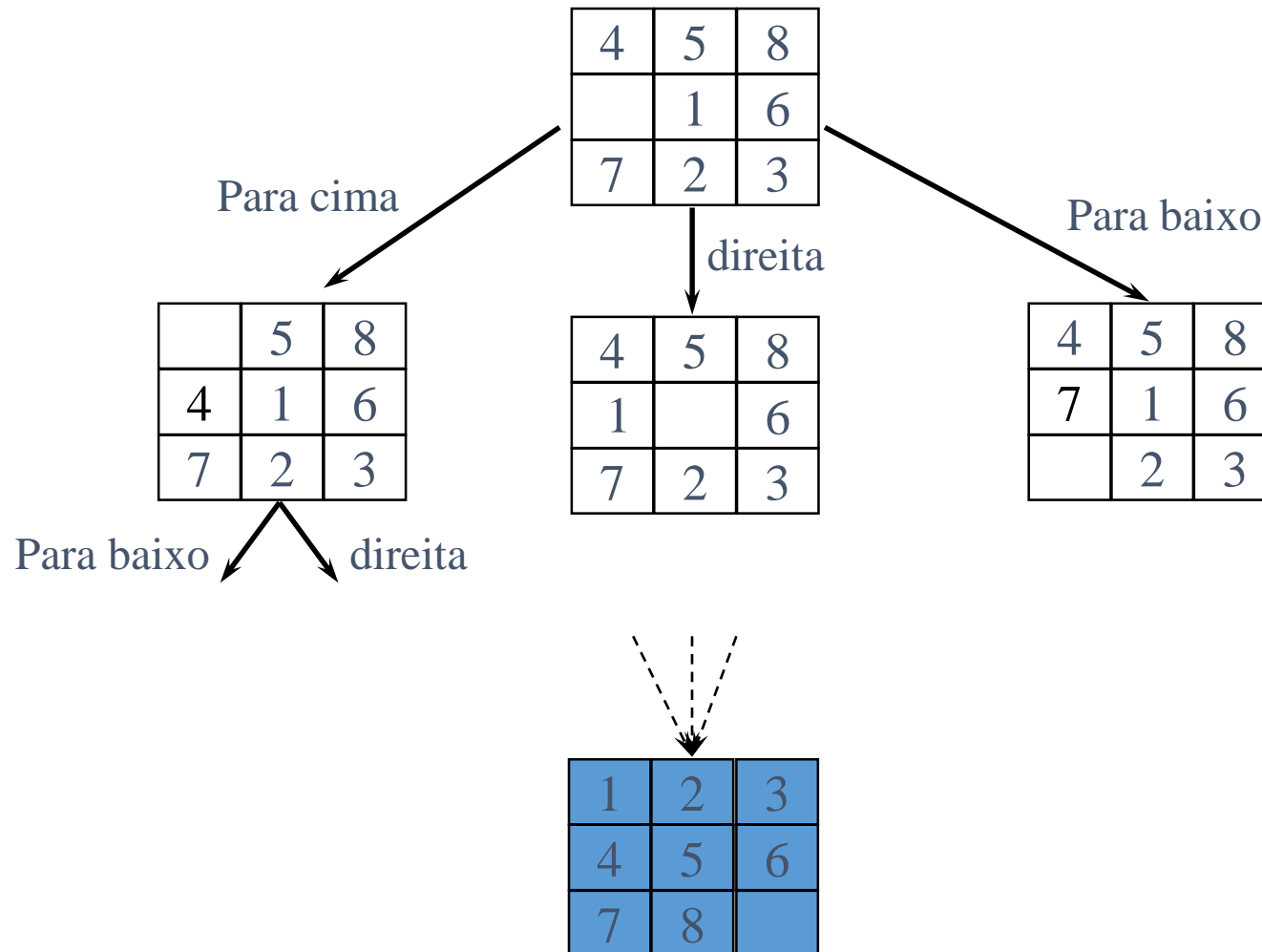
função **Busca-em-Largura** (*problema*)

retorna **uma solução ou falha**

Busca-Genérica (*problema*, Inserir-no-Fim)

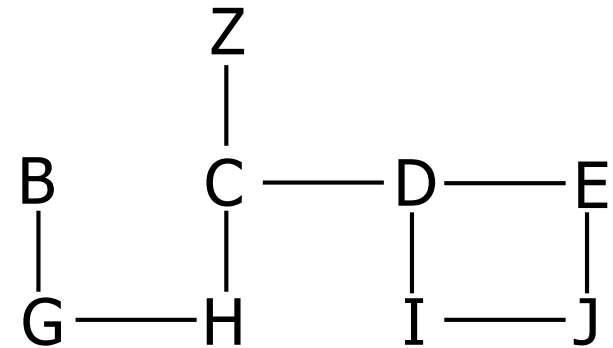


# Exemplo: Jogo dos 8 números



# Exemplo: Labirinto

- Estado inicial: Z
- Objetivo: B



# Busca em Largura

## Qualidade

- Esta estratégia é *completa*
- É *ótima* ?
  - Sempre encontra a solução mais “rasa”
    - que nem sempre é a solução de menor **custo de caminho**, caso os operadores tenham valores diferentes.
- É *ótima* se
  - $\forall n, n' \text{ profundidade}(n') \geq \text{profundidade}(n) \Rightarrow$   
 $\text{custo de caminho}(n') \geq \text{custo de caminho}(n).$ 
    - Em outras palavras, se a função **custo de caminho** é não-decrescente com a profundidade do nó.
    - A função de **custo de caminho** acumula o custo do caminho da origem ao nó atual.
  - Geralmente, isto só ocorre quando todos os operadores têm o mesmo custo (=1)

# Busca em Largura

## Custo

- Fator de expansão da árvore de busca:
  - número de nós gerados a partir de cada nó ( $b$ )
- Custo de tempo:
  - se o fator de expansão 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 =  $b + b^2 + b^3 + \dots + b^d$
  - **custo exponencial** =  $O(b^d)$ .
- Custo de memória:
  - a *fronteira* do espaço de estados deve permanecer na memória
  - é um problema mais crucial do que o tempo de execução da busca

# Busca em Largura

- Esta estratégia só dá bons resultados quando a *profundidade* da árvore de busca é *pequena*.
- Exemplo:
  - fator de expansão  $b = 10$
  - 1.000 nós gerados por segundo
  - cada nó ocupa 100 bytes

Profundidade	Nós	Tempo	Memória
0	1	1 milissegundo	100 bytes
2	111	0.1 segundo	11 quilobytes
4	11111	11 segundos	1 megabytes
6	$10^6$	18 minutos	111 megabytes
8	$10^8$	31 horas	11 gigabytes
10	$10^{10}$	128 dias	1 terabyte
12	$10^{12}$	35 anos	111 terabytes
14	$10^{14}$	3500 anos	11111 terabytes

# Busca de Custo Uniforme

- Modifica a busca em largura:
  - expande o nó da **fronteira com menor custo de caminho** na fronteira do espaço de estados
  - cada operador pode ter um custo associado diferente, medido pela função  $g(n)$ , para o nó  $n$ .
    - onde  $g(n)$  dá o custo do caminho da origem ao nó  $n$
- Na busca em largura:  **$g(n) = \text{profundidade}(n)$**
- Algoritmo:

função **Busca-de-Custo-Uniforme** (*problema*)

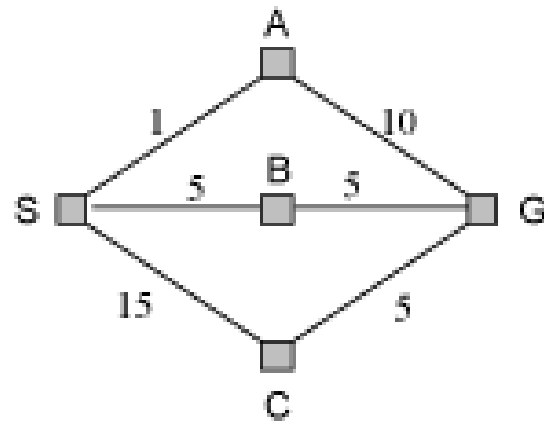
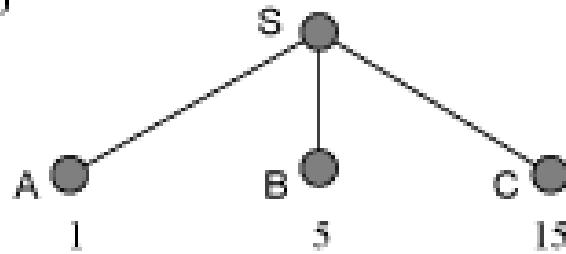
retorna **uma solução ou falha**

Busca-Genérica (*problema*, Inserir-Ordem-Crescente)

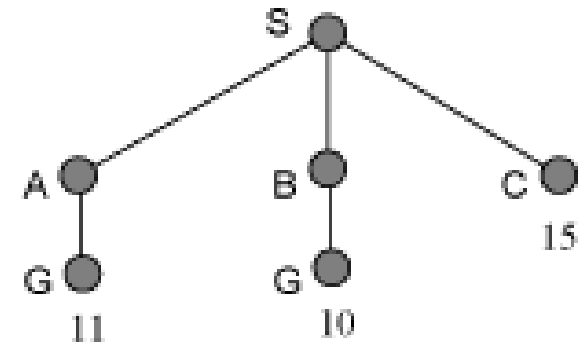
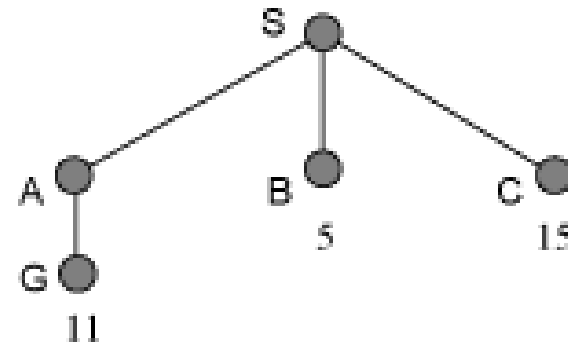


# Busca de Custo Uniforme

S ●  
0



(a)

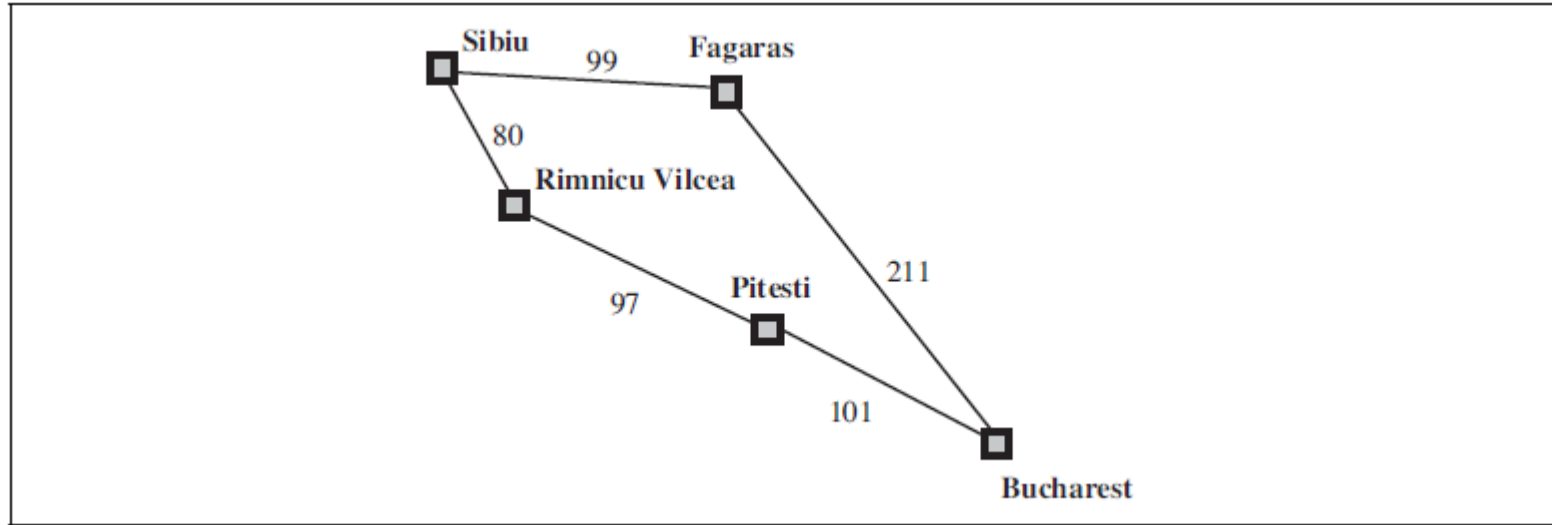


(b)

# Busca de Custo Uniforme

## Fronteira do exemplo anterior

- $F = \{S\}$ 
  - testa se  $S$  é o estado objetivo, expande-o e guarda seus filhos  $A$ ,  $B$  e  $C$  ordenadamente na fronteira
- $F = \{A, B, C\}$ 
  - testa  $A$ , expande-o e guarda seu filho  $G_A$  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!
- $F = \{B, G_A, C\}$ 
  - testa  $B$ , expande-o e guarda seu filho  $G_B$  ordenadamente
- $F = \{G_B, G_A, C\}$ 
  - testa  $G_B$  e para!



# Busca de Custo Uniforme

- É *completa*
  - Desde que o custo de cada passo exceda uma pequena constante positiva.
    - Caso contrário: algoritmo pode ficar travado em um loop infinito se existir um caminho com uma sequencia de ações de custo zero, por exemplo, sequencia de NoOp
- É *ótima* se
  - $g(\text{sucessor}(n)) \geq g(n)$ 
    - 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.
    - Ex. 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 em Profundidade

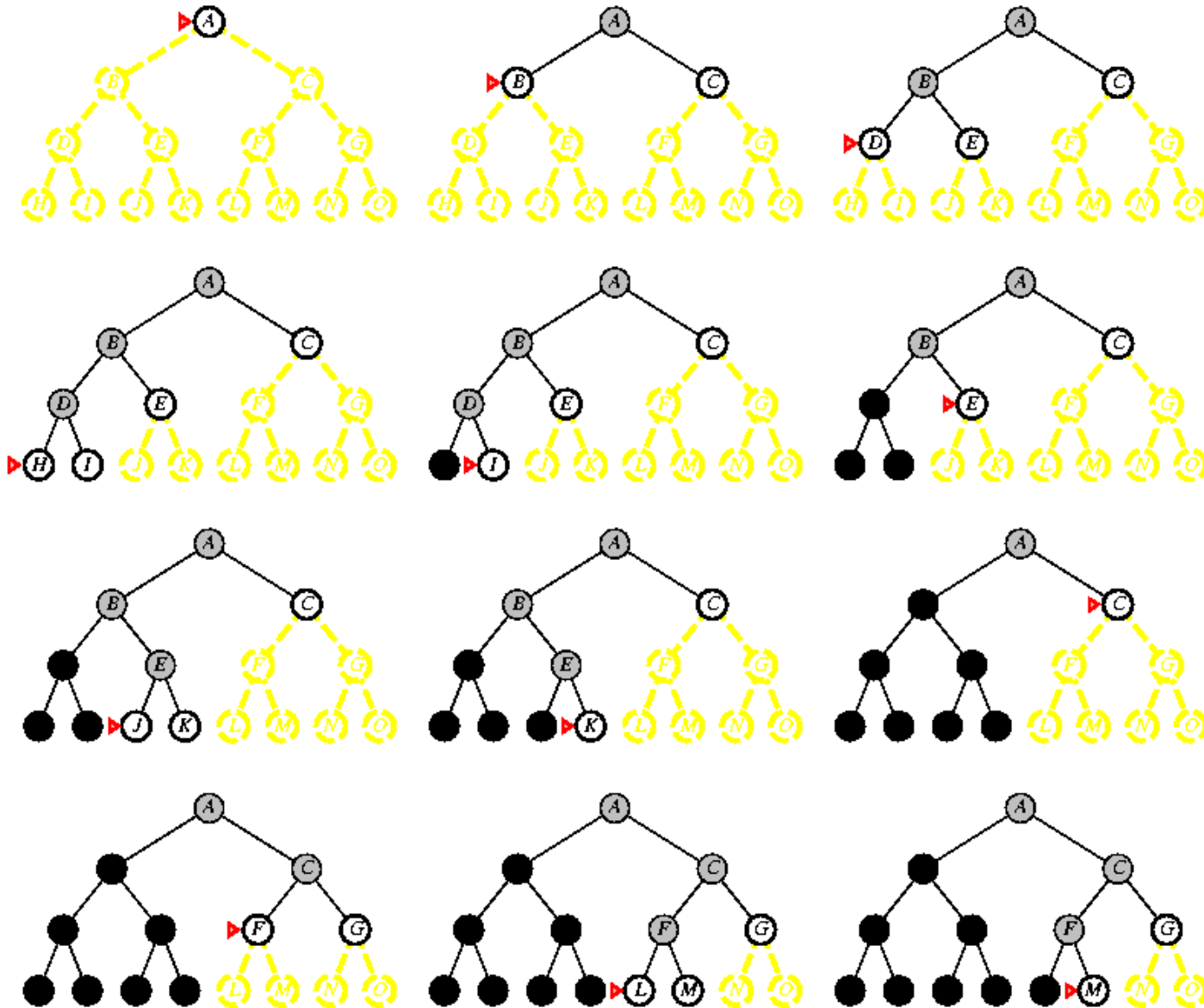
- Ordem de expansão dos nós:
  - sempre expande o nó no *nível mais profundo* da árvore:
    1. nó raiz
    2. primeiro nó de profundidade 1
    3. primeiro nó de profundidade 2, etc....
  - Quando um nó final não é solução, o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço de estados
- Algoritmo:

**função** Busca-em-Profundidade (*problema*)

retorna **uma solução ou falha**

Busca-Genérica (*problema*, Insere-no-Começo)

# Busca em Profundidade



# Busca em profundidade: análise

- Só precisa armazenar um único caminho da raiz até um nó folha, e os nós irmãos não expandidos;
- Nós cujos descendentes já foram completamente explorados podem ser retirados da memória;
- Logo para ramificação  $b$  e profundidade máxima  $m$ , a ***complexidade espacial*** é:  $O(bm)$

# Busca em Profundidade

- Esta estratégia **não é completa** nem é *ótima*.
  - Pode fazer uma escolha errada e ter que percorrer um caminho muito longo (as vezes infinito), quando uma opção diferente levaria a uma solução rapidamente (ex. nó C na fig. anterior);
- Custo de memória:
  - mantém na memória o caminho sendo expandido no momento, e os nós irmãos dos nós no caminho (para possibilitar o *backtracking*)
    - **necessita armazenar apenas  $b.m$  nós para um espaço de estados com fator de expansão  $b$  e profundidade  $m$ , onde  $m$  pode ser maior que  $d$  (profundidade da 1a. solução)**
- Custo de tempo:  **$O(b^m)$** , no pior caso.
- Observações:
  - Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.
  - Esta estratégia deve ser evitada quando as árvores geradas são muito *profundas* ou geram *caminhos infinitos*.



# Busca em profundidade limitada

- Para resolver o problema de busca em profundidade em árvores infinitas, um limite  $L$  restringe a busca. I.e., nós na profundidade  $L$  são tratados como se não tivessem sucessores.
- Resolve caminhos infinitos, porém adiciona mais incompleteza;
- Limites de profundidade podem ser conhecidos a priori:
  - ex. caminho mais longo no mapa da romênia tem  $L = 19$ , porém qqr cidade pode ser alcançada a partir de qqr outra em  $L = 9$ .
- Dois tipos de falhas terminais:
  - *falha*: nenhuma solução encontrada;
  - *corte*: nenhuma solução dentro de  $L$ ;

# Busca em profundidade limitada

- Evita o problema de caminhos muito longos ou infinitos impondo um limite máximo ( $l$ ) de profundidade para os caminhos gerados.
  - *É necessário que  $l \geq d$ , onde  $l$  é o limite de profundidade e  $d$  é a profundidade da primeira solução do problema*
- Resolve caminhos infinitos, porém adiciona mais incompletude
- Dois tipos de falhas terminais:
  - *falha*: nenhuma solução encontrada;
  - *corte*: nenhuma solução dentro de  $L$ ;
- Igual à Busca em Largura para  $i=1$  e  $n=1$

# Busca com Aprofundamento Iterativo

- Combina busca em profundidade com busca em largura;
- Faz busca em profundidade aumentando gradualmente o limite de profundidade;
- Método de busca preferido quando se tem espaço de busca grande e profundidade não conhecida;
- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - fixa profundidade =  $i$ , executa busca
  - se não chegou a um objetivo, recomeça busca com profundidade =  $i + n$  ( $n$  qualquer)
  - piora o tempo de busca, porém melhora o custo de memória!

# 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:
  - necessita armazenar apenas  $b \cdot d$  nós para um espaço de estados com fator de expansão  $b$  e limite de profundidade  $d$
- Custo de tempo:
  - $O(b^d)$
- Bons resultados quando o espaço de estados é *grande* e de *profundidade desconhecida*.

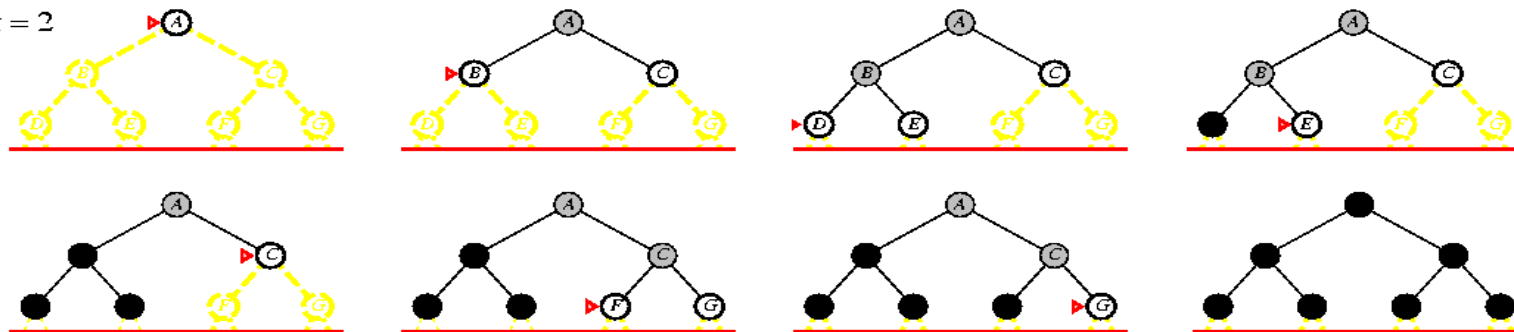
Limit = 0



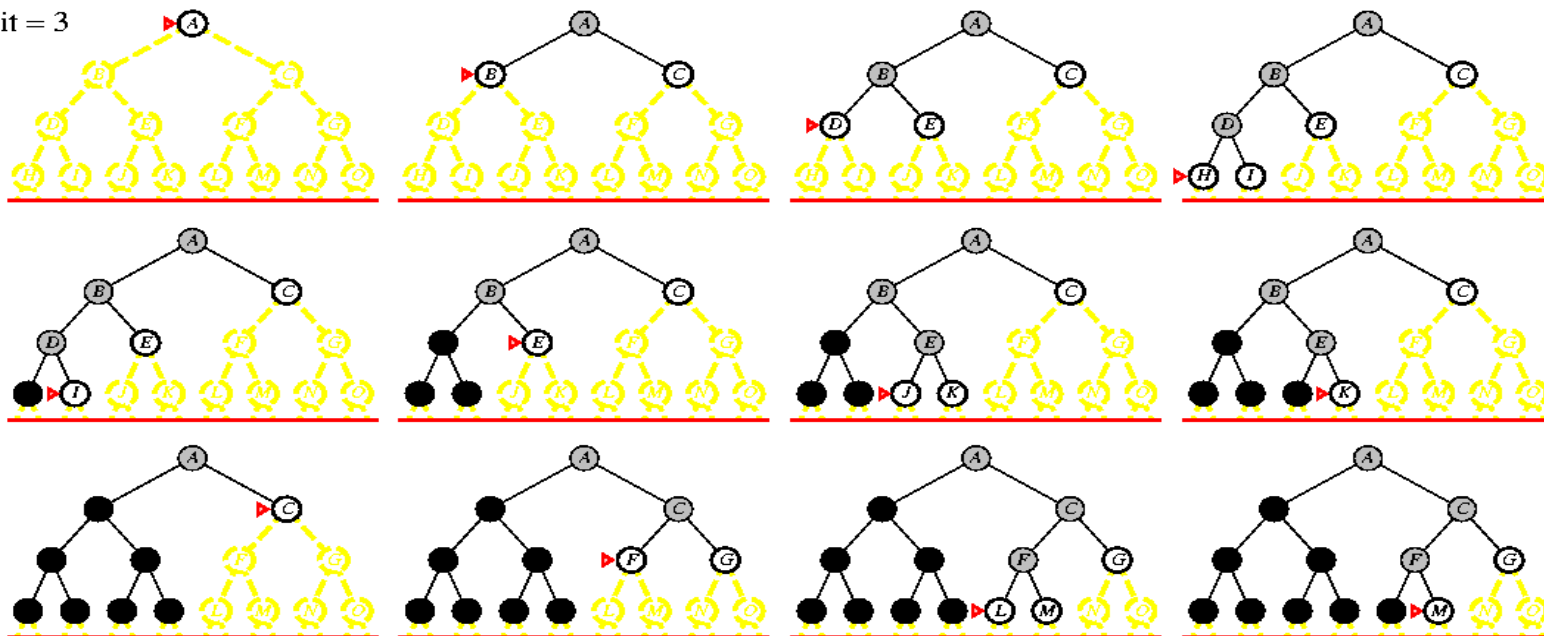
Limit = 1



Limit = 2



Limit = 3



# Comparando Estratégias de Busca Exaustiva

<b>Critério</b>	Largura	Custo Uniforme	Profundidade	Aprofundamento Iterativo
Tempo	$b^d$	$b^d$	$b^m$	$b^d$
Espaço	$b^d$	$b^d$	$bm$	$bd$
Otima?	Sim	Sim*	Não	Sim
Completa?	Sim	Sim	Não	Sim

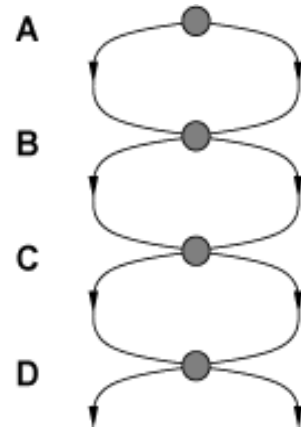
# Como evitar estados repetidos?

- Um processo de busca pode perder tempo expandindo nós já explorados antes;
- Estados repetidos podem levar a laços infinitos;
- É inevitável quando existe operadores reversíveis
  - ex. encontrar rotas, canibais e missionários, 8-números, etc.
  - a árvore de busca é potencialmente infinita

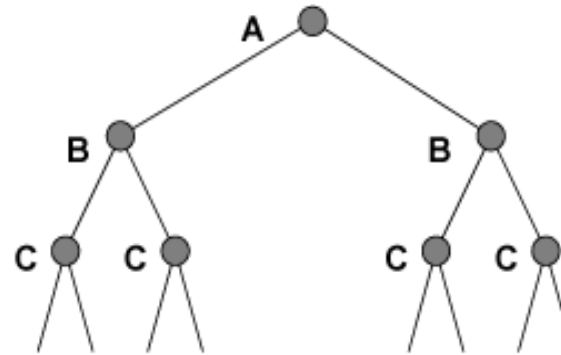
# Evitar Geração de Estados Repetidos

- Exemplo:
  - $(m + 1)$  estados no espaço  $\Rightarrow 2^m$  caminhos na árvore

*Espaço de estados*



*Árvore de busca*



- Questões
  - Como evitar expandir estados presentes em caminhos já explorados?
  - Em ordem crescente de eficácia e custo computacional?



# Evitando operadores reversíveis

- se os operadores são reversíveis:
  - conjunto de predecessores do nó = conjunto de sucessores do nó
  - porém, esses operadores podem gerar árvores *infinitas*!

# Como Evitar Estados Repetidos ?

## Algumas Dicas

1. Não retornar ao estado “pai”
  - função que rejeita geração de sucessor igual ao pai
2. Não criar caminhos com ciclos
  - não gerar sucessores para qualquer estado que já apareceu no caminho sendo expandido
3. Não gerar qualquer estado que já tenha sido criado antes (em qualquer ramo)
  - requer que todos os estados gerados permaneçam na memória
  - custo de memória:  $O(b^d)$
  - pode ser implementado mais eficientemente com *hash tables*

# A seguir...

- Busca heurística