

# *Estratégias de Busca*

March 9, 2014

# Introdução

- estratégia de busca
- árvore de busca
- nó de busca (estado inicial: raiz da árvore)
- expansão de nós da árvore
- estruturas de dados para as árvores de busca
- cada nó é uma estrutura com pelo menos 5 componentes/campos:
  - ▶ estado
  - ▶ nó pai
  - ▶ jogada/regra aplicada para gerar o nó
  - ▶ número de nós no caminho para este nó (profundidade do nó na árvore)
  - ▶ custo do caminho desde o nó raiz

# Estruturas

- datatype node components: STATE, PARENT\_NODE, OPERATOR, DEPTH, PATH\_COST
- Necessária representação para lista de nós que ainda não foram expandidos
- Escolha: **fila** de nós com as seguintes operações:
  - ▶ MAKE\_QUEUE(Elementos)
  - ▶ EMPTY?(Queue)
  - ▶ REMOVE\_FRONT(Queue) (função)
  - ▶ QUEUEING\_FN(Elementos, Queue)

## Algoritmo Geral

```

function GENERAL_SEARCH(problem, QUEUEING_FN) retorna
    solucao ou falha
nodes := MAKE_QUEUE(MAKE_NODE(INITIAL_STATE(problem)))
loop
    if EMPTY?(nodes) then return falha
    node := REMOVE_FRONT(nodes)
    if STATE(node) = GOAL_TEST[problem] then return
        node
    new_nodes := EXPAND(node, OPERATORS(problem))
    nodes := QUEUEING_FN(nodes, new_nodes)
end

```

Obs: QUEUEING\_FN é uma função variável!

# Busca de Soluções: Estratégias

- Avaliação:
  - ▶ completude
  - ▶ complexidade temporal
  - ▶ complexidade espacial
  - ▶ otimalidade

# Métodos não informados de busca (busca cega)

- largura (BL)
- custo uniforme (BCU)
- profundidade (BP)
- limitada em profundidade (BLP)
- profundidade iterativa (BPI)
- bidirecional (BB)

## Busca em Largura (BL)

- todos os nós e nível  $d$  na árvore de busca são expandidos **antes** dos nós de nível  $d+1$
- função QUEUEING\_FN: fifo

```
function BREADTH_FIRST_SEARCH(problem) retorna  
    solucao ou falha  
return GENERAL_SEARCH(problem, ENQUEUE_AT_END)
```

# Busca em Largura (BL)





## Busca em Largura: análise

- completude: se houver solução, BL garante que vai ser encontrada → completa
- se houver + de 1 solução garante que encontra a mais rasa primeiro → é ótima se o custo do caminho for uma função não decrescente da profundidade do nó
- espaço/memória = número máximo de nós para computar 1 solução de profundidade  $d$  (fator de ramificação  $b$ )

$$1 + b + b \times b + (b \times b) \times b + \dots + b^d$$

- complexidades de espaço e tempo iguais:  $O(b^d)$ , todas as folhas da árvore têm que ser armazenadas ao mesmo tempo

## Busca em Largura: análise

Profundidade	Nós	Tempo	Memória
0	1	1 ms	100 Bytes
2	111	1s	11 KB
4	11.111	11s	1MB
6	$10^6$	18 min	111 MB
8	$10^8$	31 h	11 GB
10	$10^{10}$	128 dias	1TB
12	$10^{12}$	35 anos	111TB
14	$10^{14}$	3500 anos	11.111TB

Obs: considere que 1000 nós podem ser testados e expandidos por segundo, 1 nó precisa de 100 Bytes,  $b = 10$

## Busca em Largura: análise

- qtde de memória: maior problema de BL
- se o problema tiver uma solução em nível 12, então levaria 35 anos para computar a resposta!!!
- em geral, problemas de busca com complexidade de tempo exponencial não podem ser resolvidos para instâncias muito grandes de entrada
- problema de todos os métodos de busca não informados

# Busca de Custo Uniforme (BCU)

- modificação da estratégia de BL para expandir apenas nós de baixo custo (utiliza função de custo  $g(n)$ )
- BL é de custo uniforme com  $g(n) = \text{DEPTH}(n)$
- função `QUEUEING_FN`: ordem crescente de custos

# Busca de Custo Uniforme: Análise

- BCU encontra a solução de menor custo se o custo do caminho nunca decrescer para os descendentes:

$$g(SUCCESSOR(n)) \geq g(n)$$

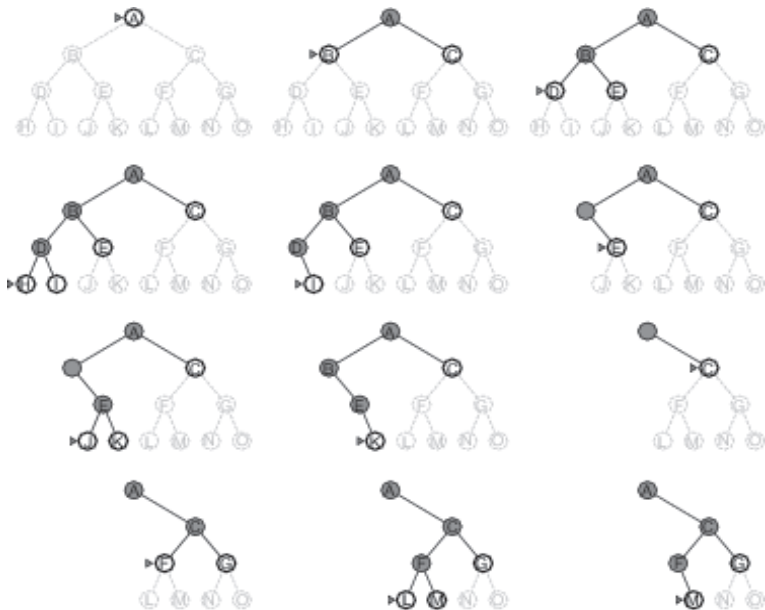
- Complexidades temporal e espacial iguais às de BL.

# Busca em Profundidade

- expande nós mais profundos da árvore primeiro
- função `QUEUEING_FN`: lifo (pilha)

```
function DEPTH_FIRST_SEARCH(problem) retorna
    solucao ou falha
return GENERAL_SEARCH(problem, ENQUEUE_AT_BEGINNING)
```

# Busca em Profundidade



## Busca em Profundidade: Análise

- espaço: somente os nós do caminho da solução + nós ainda não expandidos
- total:  $b \times m$ , onde  $b$  = fator de ramificação e  $m$  = profundidade máxima
- BP precisaria de apenas 12KB invés de 111TB para encontrar a solução em nível 12.
- tempo:  $O(b^m)$
- para problemas que têm muitas soluções, BP pode ser + rápida que BL por ter boa chance de encontrar uma solução depois de explorar um espaço pequeno do espaço de busca total
- estratégia não é completa nem ótima!!
- BP não deve ser usada para árvores de muita ou profundidade infinita!!



# Busca Limitada em Profundidade

- Impõe um corte na profundidade máxima de um caminho
- necessário verificar profundidade de cada nó expandido
- função **EXPAND** responsável por verificar se profundidade limite foi atingida

# Busca Limitada em Profundidade: Análise

- estratégia é completa, mas....
- ...não é ótima!
- se limite de profundidade muito pequeno, não garantimos completude
- tempo:  $O(b^l)$ , com 1, limite de prof.
- espaço:  $O(b \times l)$

# Busca em Profundidade Iterativa

- tenta todos os possíveis limites de profundidade
- combina vantagens de BL com BP

```
function ITERATIVE_DEEPENING_SEARCH(problem) retorna
    solucao ou falha
    for depth := 0 to infinito do
        if DEPTH_LIMITED_SEARCH(problem,depth) then
            return solucao
    end
    return falha
```

## Busca em Profundidade Iterativa: Análise

- é ótima e completa
- espaço: como busca em profundidade
- ordem de expansão dos nós análogo à BL, com exceção de que alguns estados são expandidos várias vezes
- tempo:

$$(d+1) \times 1 + (d) \times b + (d-1) \times b^2 + \dots + 3 \times b^{d-2} + 2 \times b^{d-1} + 1 \times b^d$$

- nós da folha da última iteração são expandidos uma única vez
- próximo nível: duas vezes etc
- raiz expandida  $d+1$  vezes
- complexidade temporal:  $O(b^d)$
- método utilizado qdo a árvore de busca é muito grande e prof da solução não conhecida

## Busca Bidirecional (BB)

- Busca a partir dos dois estados - inicial e final
- BB termina quando as duas buscas se encontram
- para problemas com fator de ramificação  $b$ , BB pode fazer gde diferença em termos de tempo de execução/espaço consumido
- $b=10, d=6$ :
  - ▶ BL: 1.111.111 nós
  - ▶ BB: 2.222 nós ( $2 \times b^{\frac{d}{2}} \rightarrow O(b^{\frac{d}{2}})$ )

## Busca Bidirecional (BB): Algoritmo

```
function BIDIRECIONAL_SEARCH(problem, QUEUEING_FN1, QUEUEING_FN2)
    retorna solucao ou falha
nodes1 := MAKE_QUEUE(MAKE_NODE(INITIAL_STATE(problem)))
nodes2 := MAKE_QUEUE(MAKE_NODE(FINAL_STATE(problem)))
loop
    if EMPTY?(nodes1) || EMPTY?(nodes2) then return falha
    node1 := REMOVE_FRONT(nodes1)
    node2 := REMOVE_FRONT(nodes2)

    if node1 in nodes2 || node1 = node2 then return solucao
    if node2 in nodes1 || node1 = node2 then return solucao

    new_nodes1 := EXPAND(node1, OPERATORS(problem))
    nodes1 := QUEUEING_FN1(nodes1, new_nodes1)
    new_nodes2 := EXPAND(node2, OPERATORS(problem))
    nodes2 := QUEUEING_FN2(nodes2, new_nodes1)
```

# Busca Bidirecional (BB): Algoritmo

## Busca Bidirecional (BB)

- Na prática:
  - ▶ precisamos de formas eficientes para descobrir se um novo nó já apareceu na árvore de busca da outra busca
  - ▶ decidir o tipo de busca a ser utilizada em cada metade do espaço de busca
  - ▶ seleção de sucessores e predecessores

## Comparação entre buscas

Critério	BL	BCU	BP	BLP	BPI	BB
tempo	$b^d$	$b^d$	$b^m$	$b^l$	$b^d$	$b^{\frac{d}{2}}$
espaço	$b^d$	$b^d$	$b \times m$	$b \times l$	$b \times d$	$b^{\frac{d}{2}}$
otimalidade	sim	sim	não	não	sim	sim
completude	sim	sim	não	sim, se $l \geq d$	sim	sim