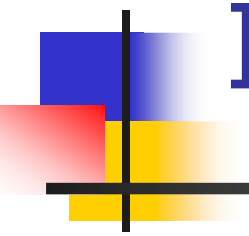
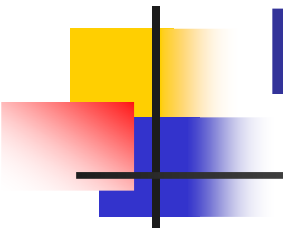


Inteligência Artificial

Prof. Tiago A. E. Ferreira

Aula 6 – Busca Cega



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
- Direção da expansão:
  1. Do estado inicial para um estado final
  2. De um estado final para o estado inicial
  3. Busca bi-direcional

Critérios de 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/Otimização (*optimality*):
  - a estratégia encontra **a melhor solução** quando existem soluções diferentes?
    - menor custo de caminho

Busca em Largura

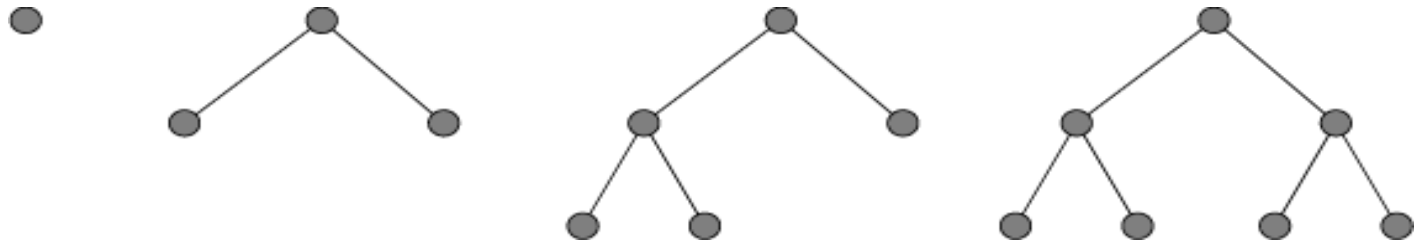
- Ordem de expansão dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc…

- Algoritmo:

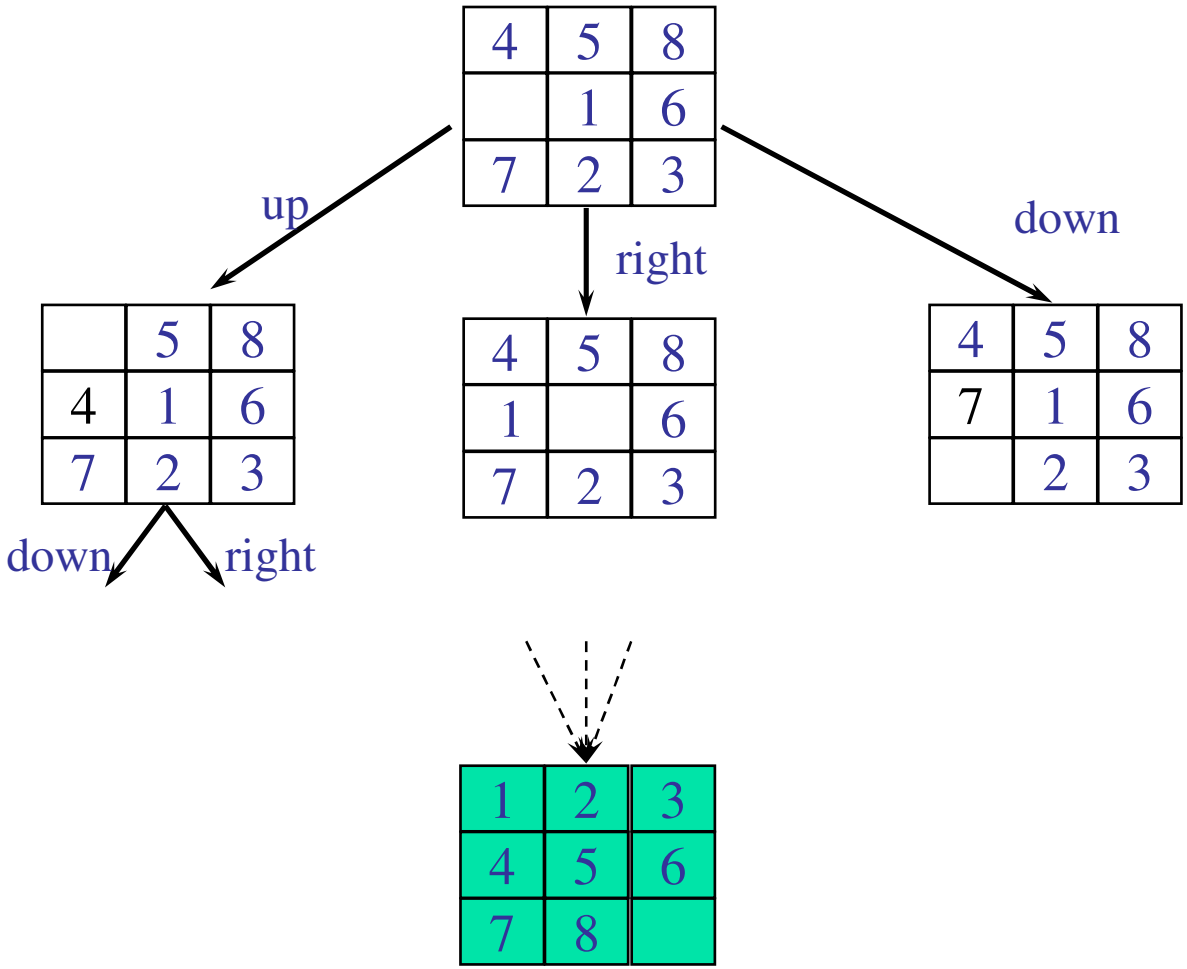
  função **Busca-em-Largura** (***problema***)

  retorna **uma solução ou falha**

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



Exemplo: Jogo dos 8 números



Busca em Largura

- 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' profundidade(n') $\geq$ profundidade(n) $\Rightarrow$

    custo de caminho(n') $\geq$ custo de caminho (n).
    - A função **custo de caminho** é não-decrescente com a profundidade do nó.
    - Essa função 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

- 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 = $1 + b + b^2 + b^3 + \ldots + b^d$
  - **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

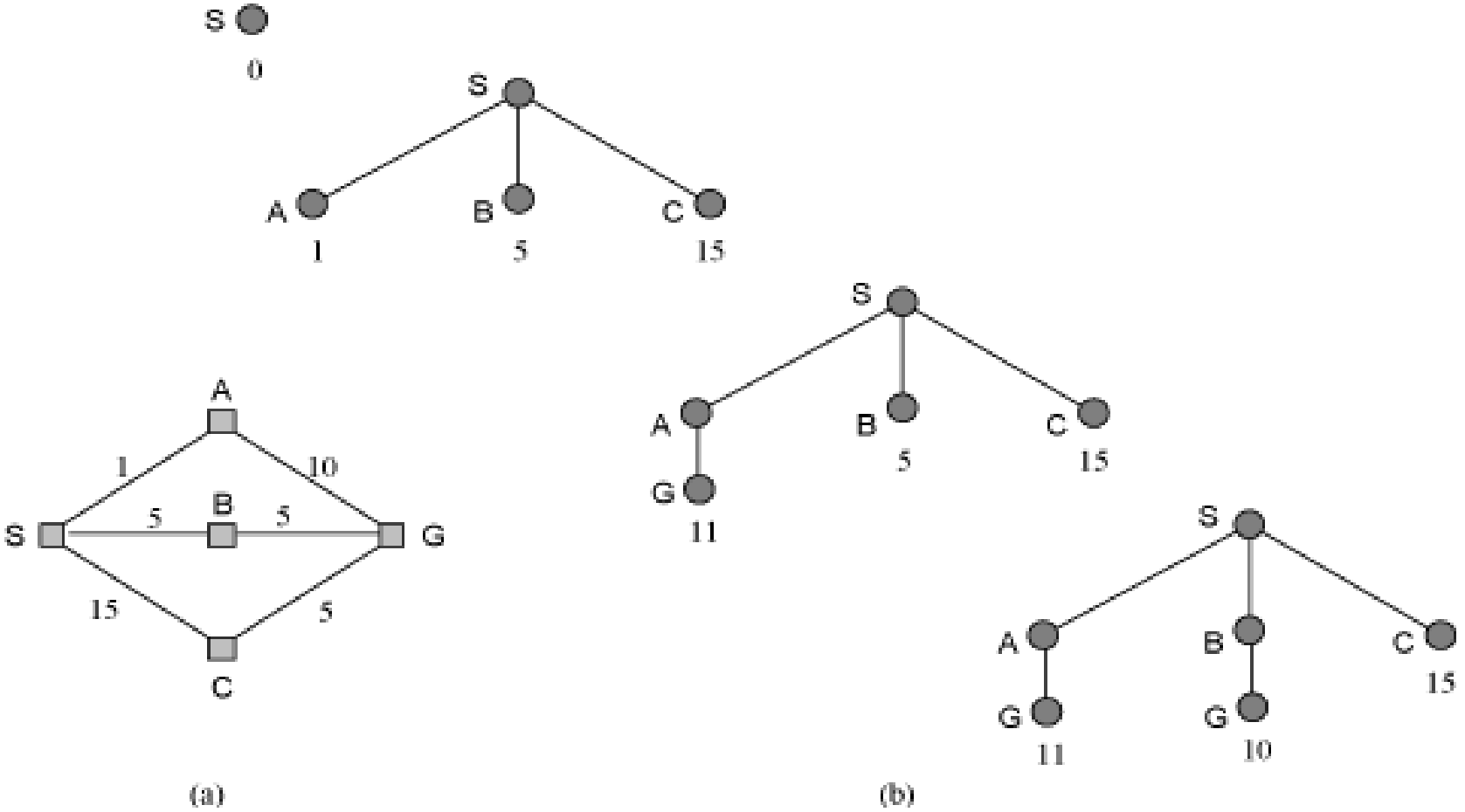
- Modifica a busca em largura:
  - expande o nó da fronteira com menor custo de caminho na fronteira do espaço de estados
  - cada operador pode ter um custo associado diferente, medido pela função *g(n)*, para o nó *n*.
    - onde g(n) dá o custo do caminho da origem ao nó n
- Na busca em largura: ***g(n)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

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

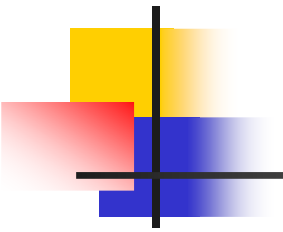
  retorna **uma solução ou falha**

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

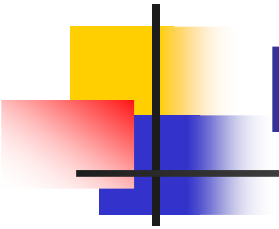
Busca de Custo Uniforme



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 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}
  - testa B, expande-o e guarda seu filho GB ordenadamente
- F= {GB, GA, C}
  - testa GB e para!



Busca de Custo Uniforme

- Esta estratégia é *completa*
- É *ótima* se
  - g (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

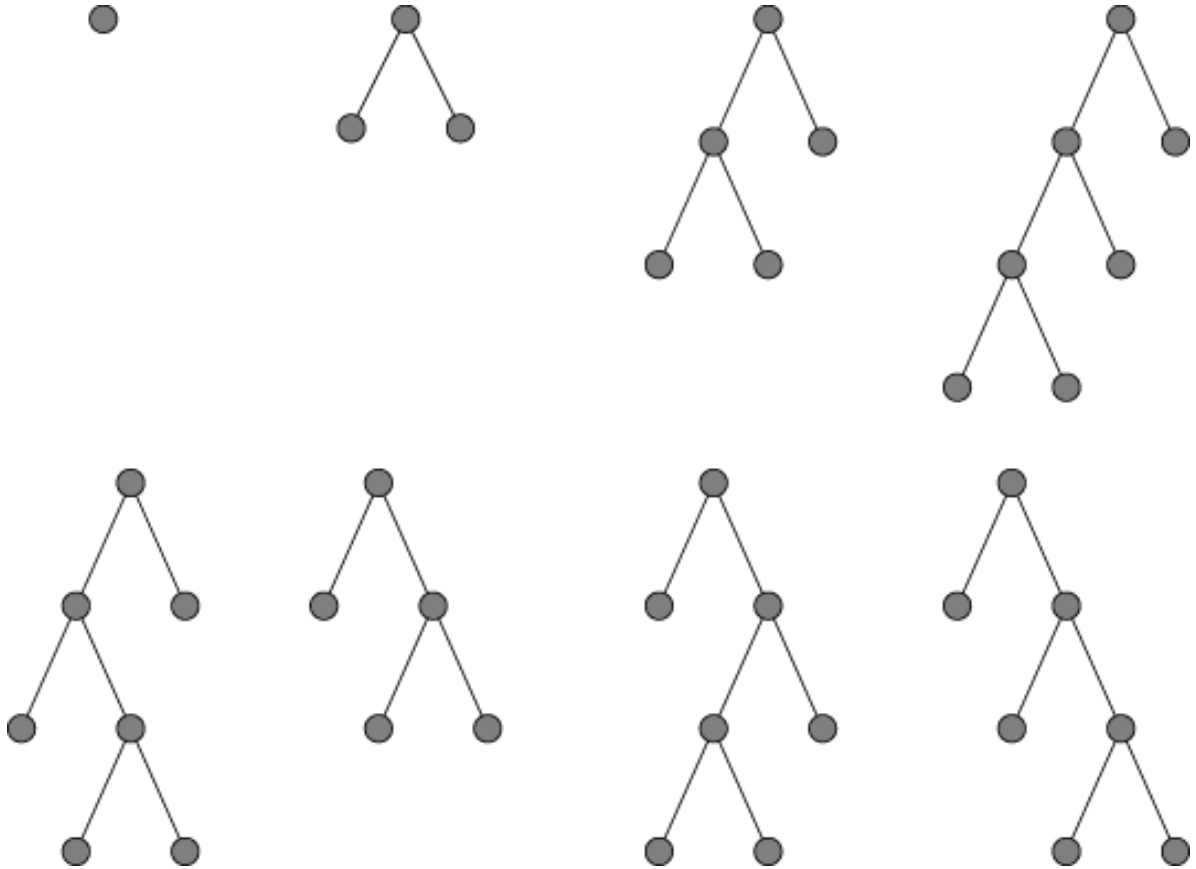
- Ordem de expansão dos nós:
  - sempre expande o nó no *nível mais profundo* da árvore:

    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)

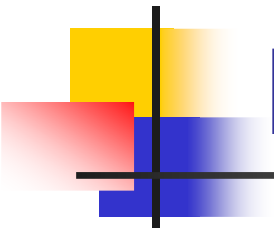
Busca em Profundidade



Busca em Profundidade

- Esta estratégia não é *completa* nem é *ótima*.
- Custo de memória:
  - mantém na memória o caminho que está sendo expandido no momento, e os nós irmãos dos nós no caminho (para possibilitar o *backtracking*)

    $\Rightarrow$ 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.



Busca em Profundidade

- **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*.**

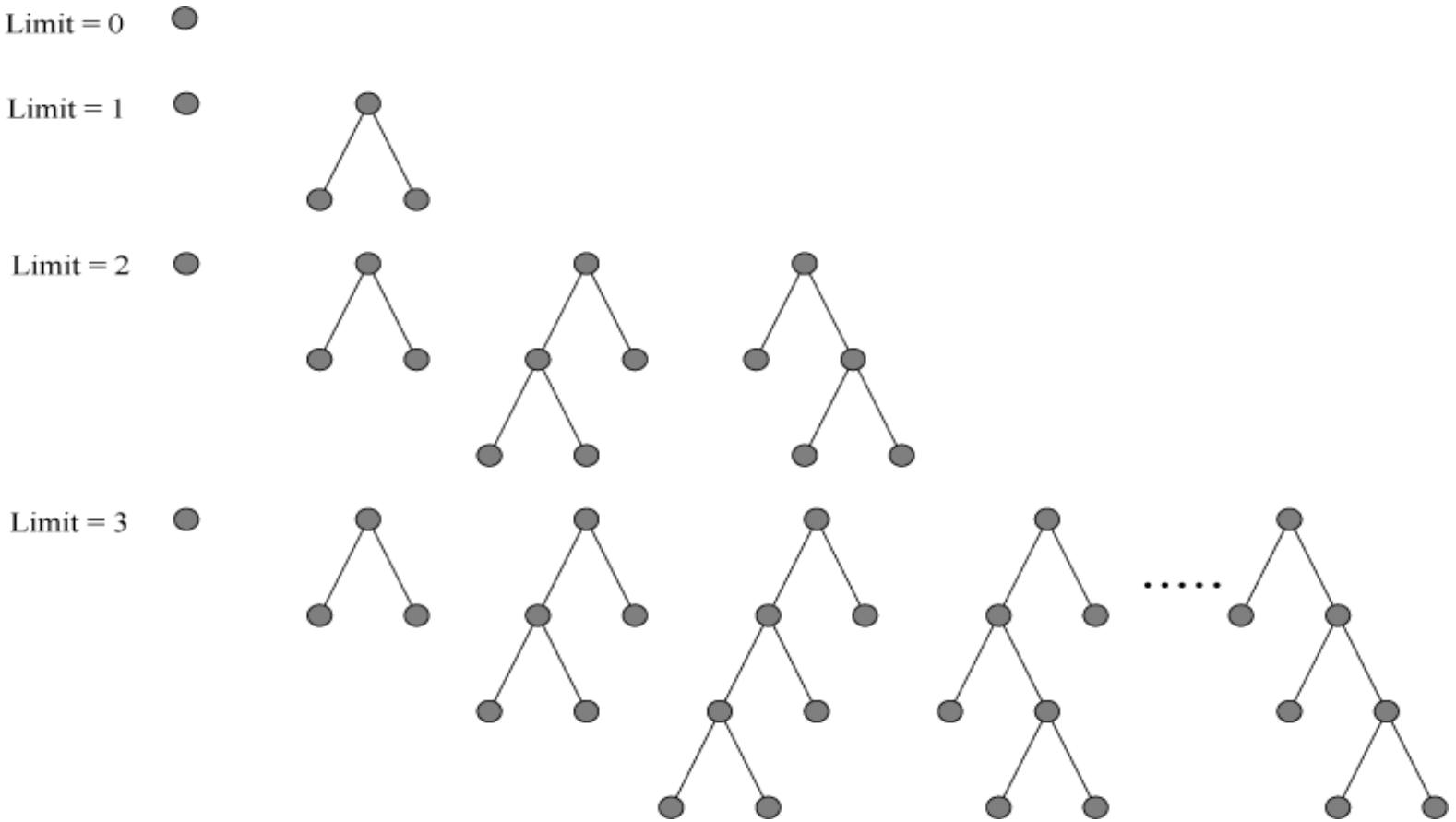
Busca com Aprofundamento Iterativo

- **Evita o problema de caminhos muito longos ou infinitos impondo um limite máximo *(l)* de profundidade para os caminhos gerados.**
  - $l \geq d$, onde l é o limite de profundidade e d é a profundidade da primeira solução do problema
- **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!
- **Igual à Busca em Largura para *i=1* e *n=1***

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.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*.

Busca com Aprofundamento Iterativo

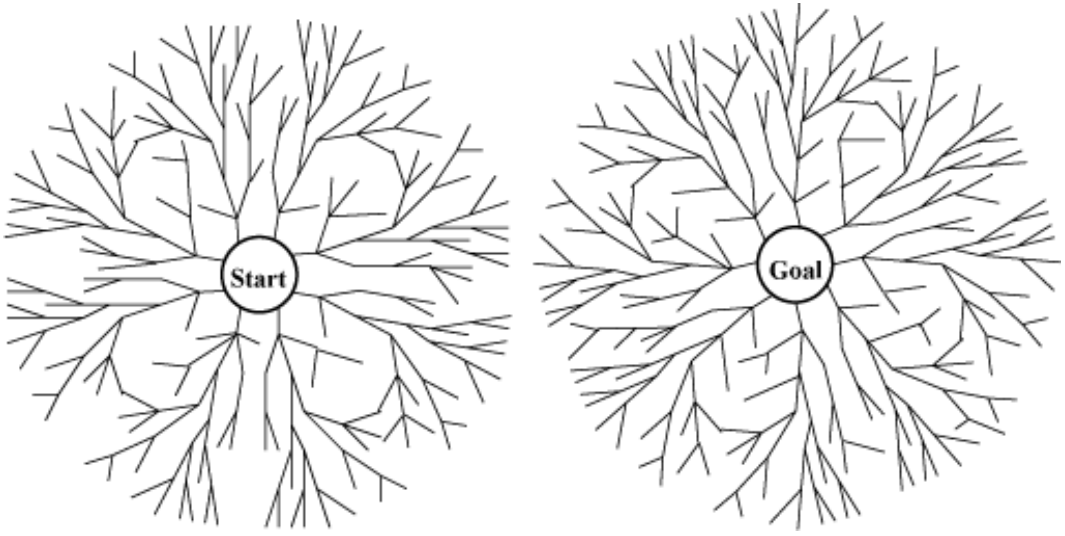


Comparando Estratégias de Busca Exaustiva

Critério	Largura	Custo Uniforme	Profun-didade	Aprofun-damento 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

Busca Bidirecional

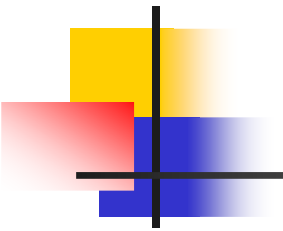
- Busca em duas direções:
  - para frente, a partir do nó inicial, e
  - para trás, a partir do nó final (objetivo)
- A busca pára quando os dois processos geram um mesmo estado intermediário.
- É possível utilizar *estratégias* diferentes em cada direção da busca.



Busca Bidirecional

- Custo de tempo:
  - Se fator de expansão *b* nas duas direções, e a profundidade do último nó gerado é *d*: $O(2b^{d/2}) = O(b^{d/2})$
- Custo de memória: $\boldsymbol{O(b^{d/2})}$
- Busca para trás gera *predecessores* do nó final
  - se os operadores são **reversíveis**:
    - conjunto de predecessores do nó = conjunto de sucessores do nó
    - porém, esses operadores podem gerar árvores *infinitas*!
  - **caso contrário**, a geração de predecessores fica muito difícil
    - descrição desse conjunto é uma propriedade abstrata
    - e.g., como determinar exatamente todos os estados que precedem um estado de xeque-mate?
  - problemas também quando existem muitos estados finais (objetivos) no problema.

Evitar Geração de Estados Repetidos

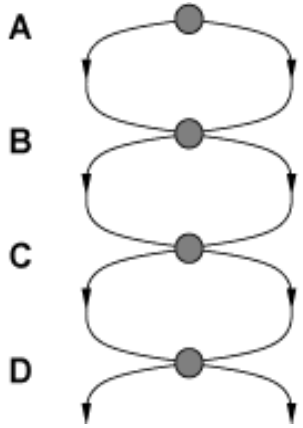


- Problema geral em Busca
  - expandir estados presentes em caminhos já explorados
- É inevitável quando existe operadores reversíveis
  - ex. encontrar rotas, canibais e missionários, 8-números, etc.
  - a árvore de busca é potencialmente infinita
- Idéia
  - **podar** (prune) estados repetidos, para gerar apenas a parte da árvore que corresponde ao grafo do espaço de estados (que é finito!)
  - mesmo quando esta árvore é finita...evitar estados repetidos pode reduzir exponencialmente o custo da busca

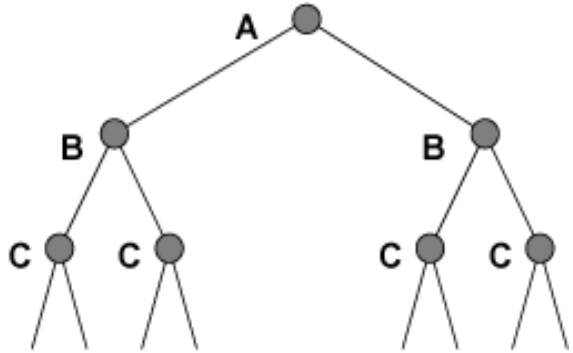
Evitar Geração de Estados Repetidos

- Exemplo:
  - (m + 1) estados no espaço => 2^m caminhos na árvore

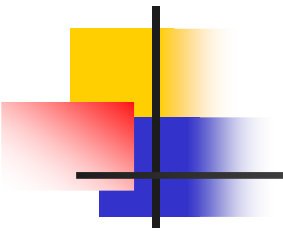
Espaço de estados



Árvore de busca

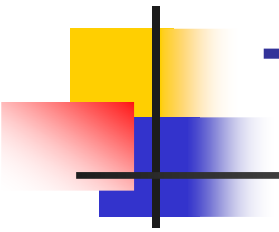


- Questões
  - Como evitar expandir estados presentes em caminhos já explorados?
  - Em ordem crescente de eficácia e custo computacional?



Evitar Estados Repetidos: soluções

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*



Tensão (trade-off) básica

- Problema:
  - Custo de armazenamento e verificação
- Solução
  - depende do problema
  - quanto mais “loops”, mais vantagem em evitá-los!