Inteligência artificial

Resolução de problemas via busca

Adaptação dos slides de **Stuart Russel** e **Peter Norvig**, disponíveis em **aima.cs.berkeley.edu.**

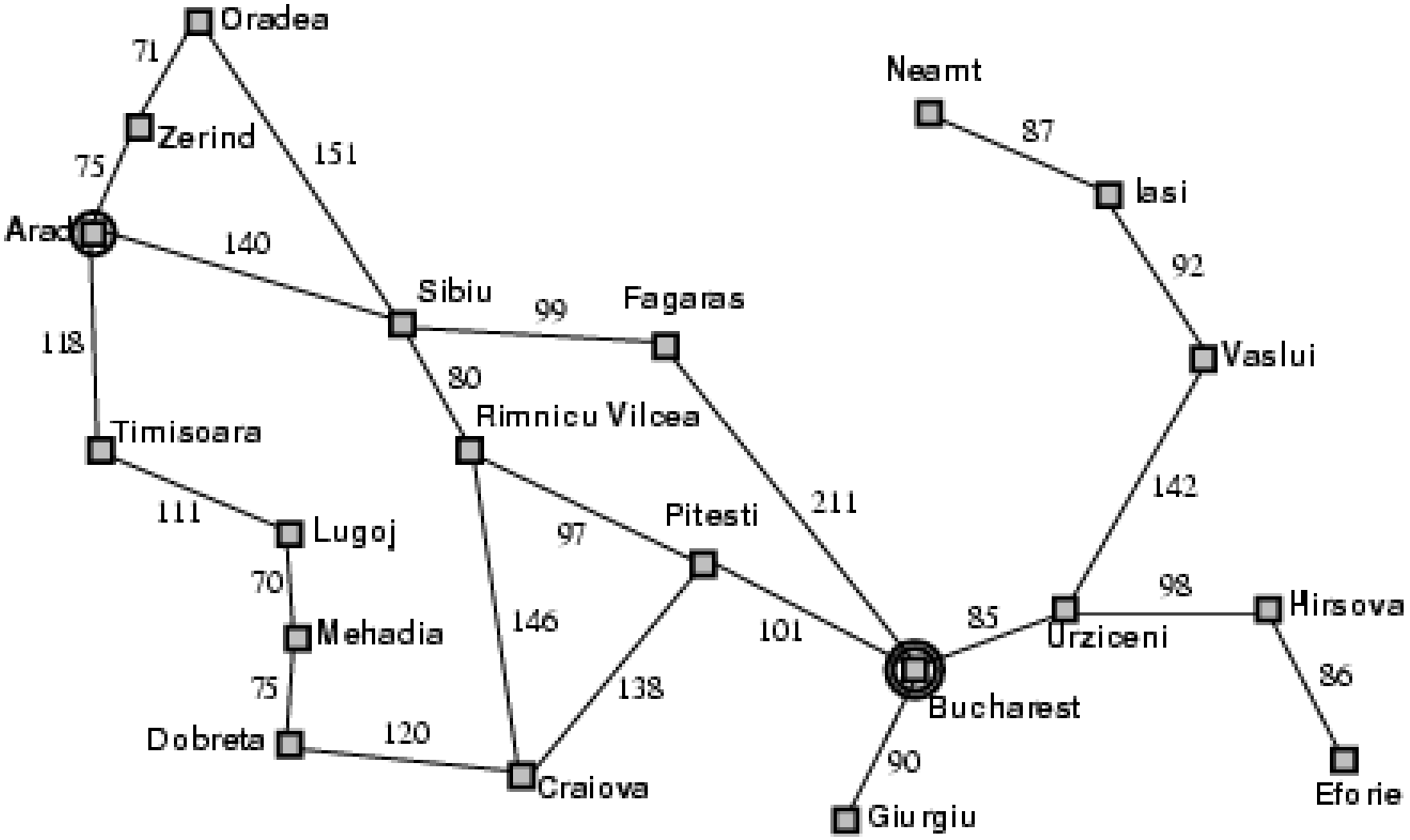
Estrutura da apresentação

- Tipos de Problemas
  - Estados únicos (totalmente **observável**)
  - Os ambientes devem ser **discretos** e **determinísticos**
- Formulação do Problema
  - Problemas exemplo
- Algoritmos de Busca Básicos
  - Não informados

Introdução

- Estudaremos um paradigma para resolução de problemas;
- Encontrar soluções por meio da geração sistemática de novos estados, os quais são testados a fim de se verificar se correspondem à solução do problema;
- Assume-se que o *raciocínio* se reduz à busca;
- Abordagem eficiente para uma série de problemas práticos (principalmente nas versões *informadas)*.

Exemplo: Mapa da Romênia



Exemplo: Mapa da Romênia

- Férias na Romênia, atualmente em Arad
  - Pegar voo que sai de Bucareste
- Formulação da Meta
  - Estar em Bucareste
- Formulação do Problema
  - Estados: (estar em) cada cidade representa um estado.
  - Ações: dirigir de uma cidade para outra.
- Solução do Problema
  - Sequência de cidades; e.g. Arad, Sibiu, Fagaras, Bucareste, …

Solução de Problemas por Busca

- Quatro passos gerais:
  - Formulação da meta
    - Qual ou quais estados correspondem à solução do problema?
  - Formulação do problema
    - Quais ações e estados considerar dada a meta?
  - Busca pela solução
    - Encontre uma sequência (ou a melhor das sequências) de ações que leve à meta.
  - Execução
    - Implemente as ações.

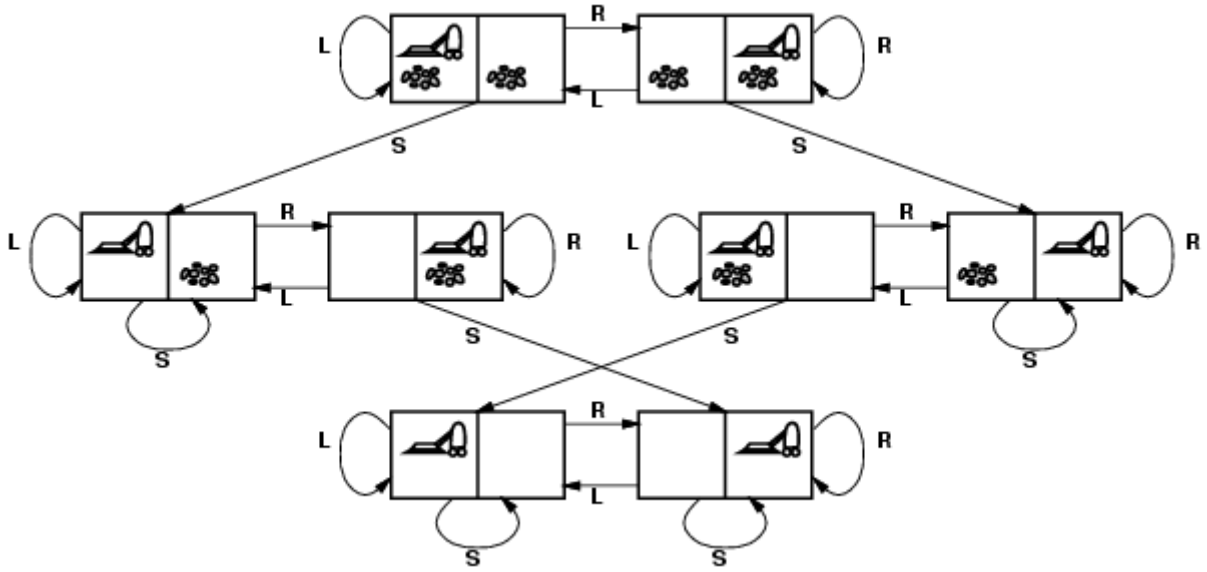
Formulação do Problema

- Um problema é definido por:
  - Um Estado Inicial, ex. *Arad*.
  - Função Sucessora *S(x)* = conjunto de pares ação-estado
    - e.g. *S(Arad)*={<*Arad* → *Zerind, Zerind*>,…}
    - Estado inicial + função sucessora = espaço de estados
  - Teste de Meta, que pode ser
    - Explícito, e.g. *x* == *'Bucareste'*
    - Implícito, e.g. *chequemate(x)*
  - Custo de Caminho (aditivo)
    - Ex. soma de distâncias, número de ações executadas, …
    - *c(x,a,y)* é o custo do passo (a partir do estado *x* para o estado *y* através da ação *a*), por premissa não negativo.
  - Uma Solução é uma sequência de ações do estado inicial para o estado meta.
  - Uma Solução Ótima é tal que possui o menor custo de caminho.

Espaço de Estados

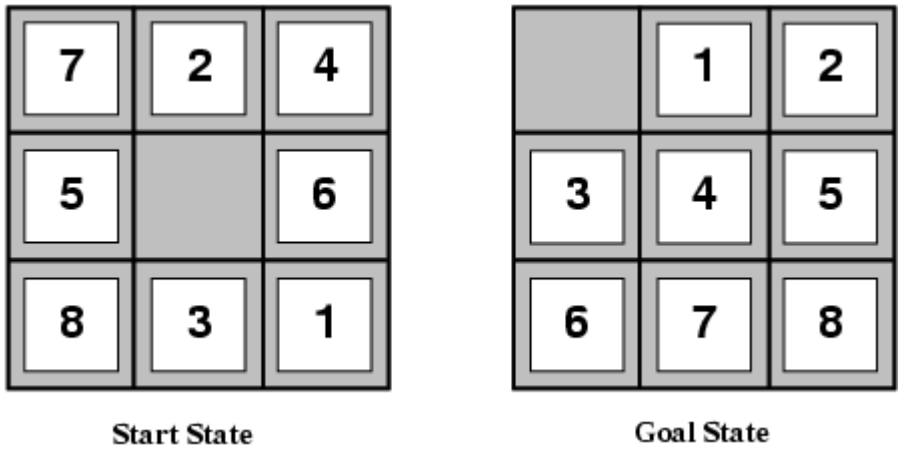
- Mundo real é muito complexo.
- Espaços de estados e de ações devem ser *abstraídos*.
  - ex. Arad → Zerind representa um conjunto complexo de possíveis rotas, desvios, paradas de abastecimento, etc.
  - A abstração é válida se o caminho entre dois estados é refletido no mundo real.
- Solução Abstrata = Conjunto de caminhos reais que são soluções no mundo real.
- Cada ação abstrata deve ser "mais fácil" que no problema real.

Gráfico do Espaço de Estados do Aspirador de Pó



- Quais os estados, ações, meta(s), custo de caminho?

Exemplo: 8-Puzzle

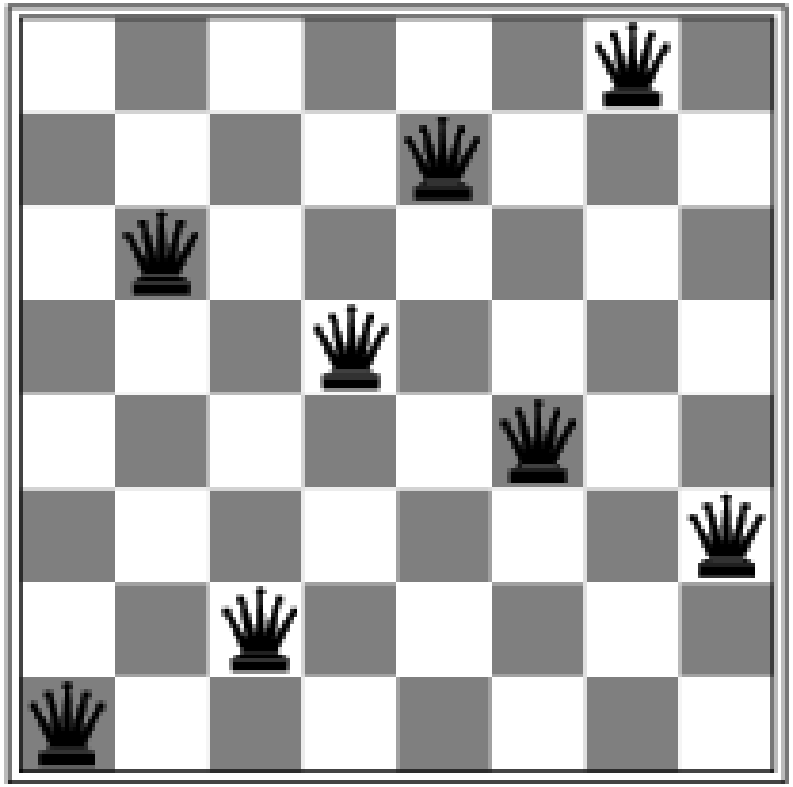


- Quais os estados, ações, meta(s), custo de caminho?

Exemplo: 8-puzzle

- Problema pertencente à classe NP-Completa
- O quebra cabeça de 8 peças tem 9!/2=181.440 estados acessíveis.
- O quebra cabeça de 15 peças (tabuleiro 4 x 4) tem aproximadamente 1,3 trilhão de estados!
- Um quebra cabeças de 24 peças tem cerca de 10^{25} estados!

Exemplo: problema das 8 rainhas

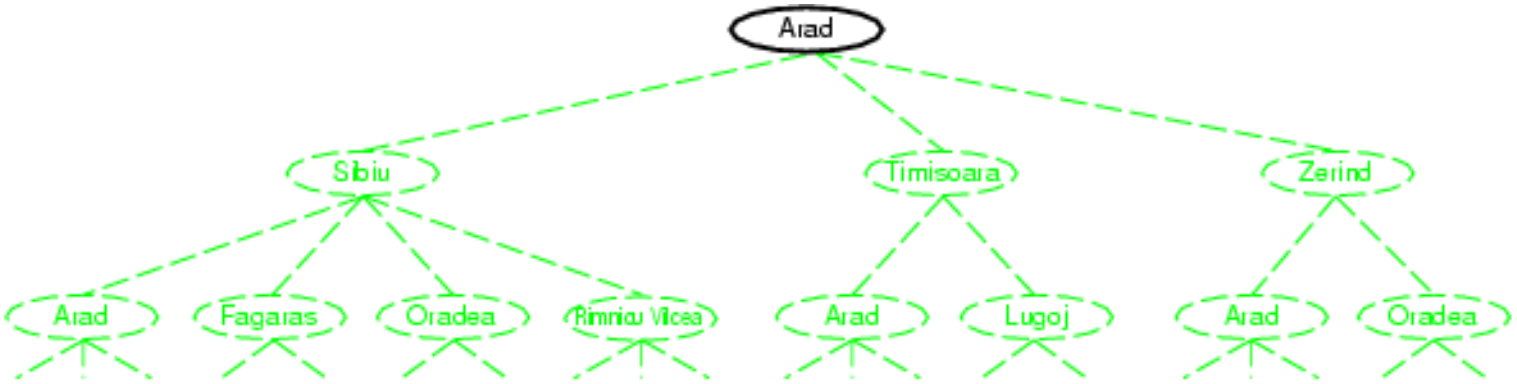


- Quais os estados, ações, meta(s), custo de caminho?

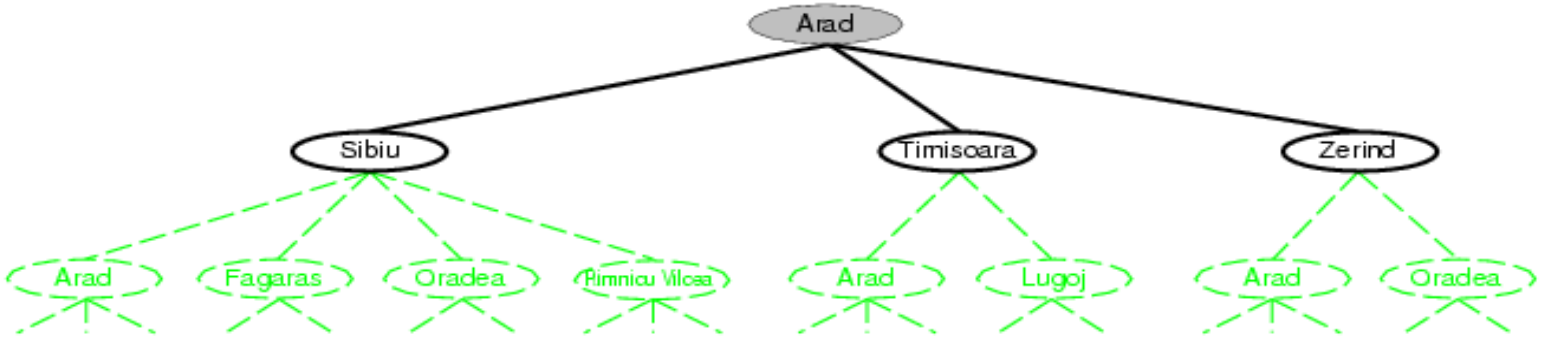
Algoritmos de Busca Básicos

- Soluções para os problemas anteriores?
  - Buscar no espaço de estados;
  - Nos concentraremos na busca através de *geração explícita de árvore*:
    - Raiz= estado inicial.
    - Demais nodos gerados através da função sucessora.
  - Em geral a busca se dá, na verdade, sobre um **grafo** (mesmo estado alcançado por múltiplos caminhos)

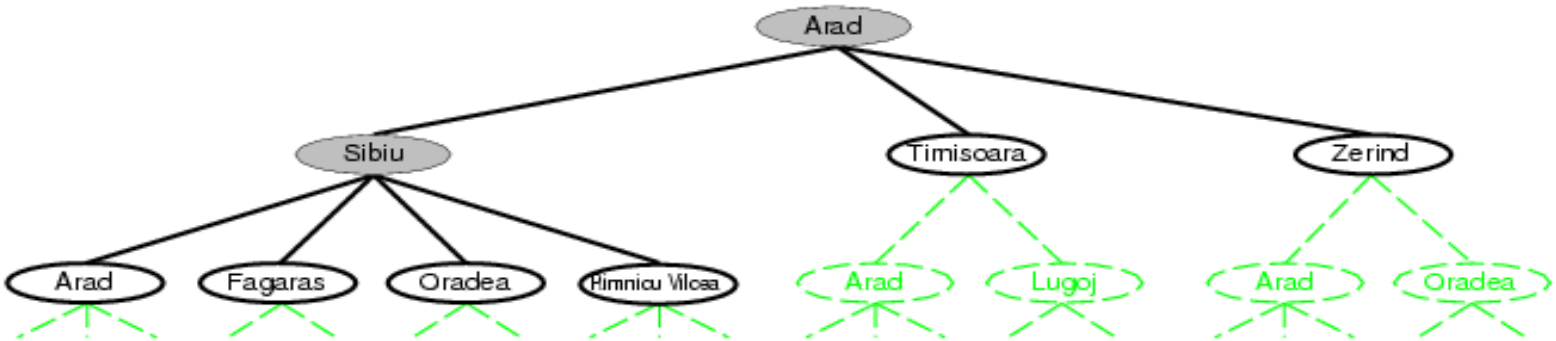
Exemplo de Busca em Árvore



Exemplo de Busca em Árvore

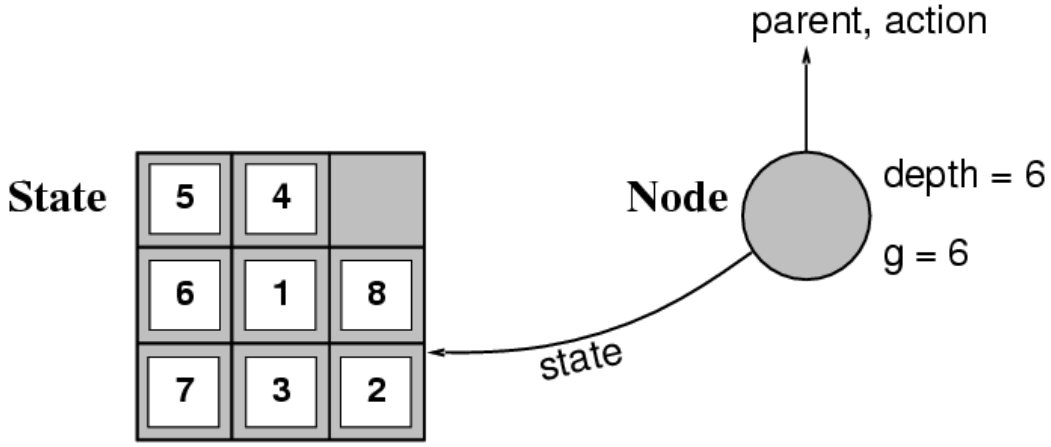


Exemplo de Busca em Árvore



Implementação: nodos vs. estados

- Um estado é uma (representação) de uma configuração física
- Um nodo é uma estrutura de dados constituindo parte da árvore de busca e que inclui estado, nodo pai, ação, custo de caminho *g(x)* e profundidade



- Vários nós diferentes podem representar o mesmo estado

Estratégias de Busca

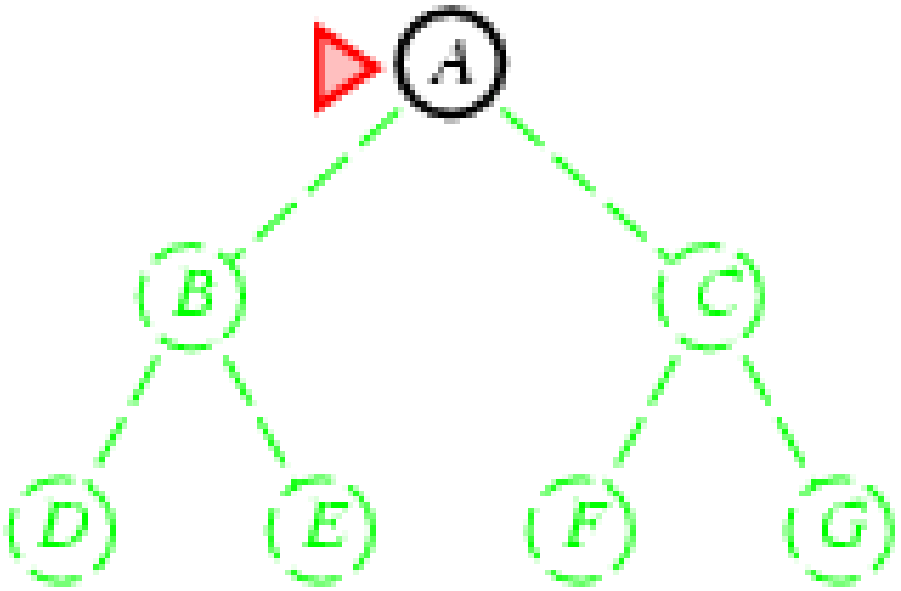
- Uma estratégia de busca é basicamente uma ordem de expansão dos nodos.
- Medidas de desempenho para diferentes estratégias:
  - Completude: Sempre encontra uma solução (se existir)?
  - Otimalidade: Sempre encontra a solução de custo mais baixo?
  - Complexidade (tempo): Número de nodos explorados?
  - Complexidade (espaço): Número de nodos armazenados?
- Complexidade usualmente medida em função da dificuldade do problema:
  - b : máximo fator de ramificação da árvore de busca.
  - d : profundidade da solução de menor custo.
  - m : máxima profundidade do espaço de estados (pode ser ∞).

Estratégias Não-Informadas

- Também denominadas de busca cega: usam estritamente a informação disponível na formulação do problema.
  - Quando é possível utilizar informação adicional para determinar se um nodo não-meta é mais promissor do que outro → busca informada.
- Diferenciam-se pela abordagem de expansão:
  - Busca em largura (*Breadth-first search*).
  - Busca uniforme (*Uniform-cost search*).
  - Busca em profundidade (*Depth-first search*)
  - Busca em profundidade limitada (*Depth-limited search*)
  - Busca em profundidade iterativa (*Iterative deepening search*).

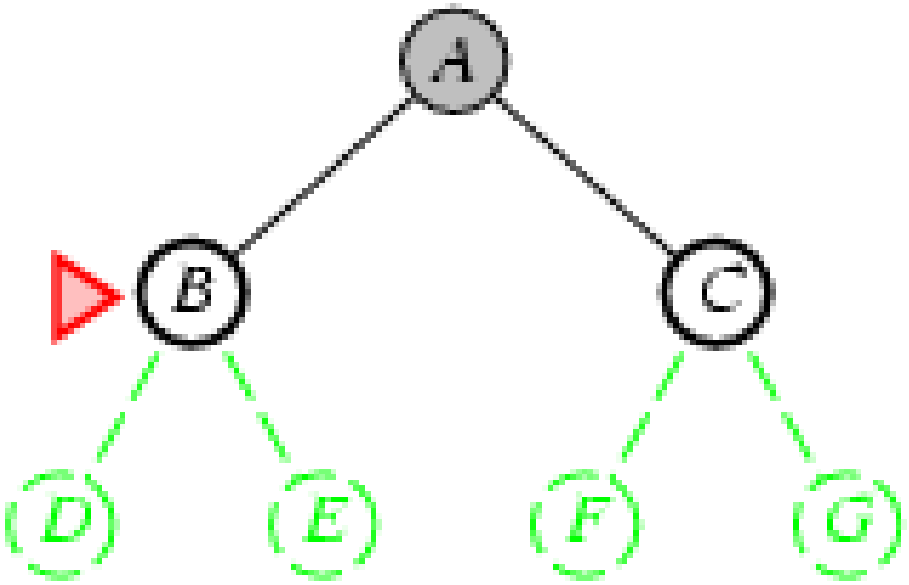
Busca em Largura

- Expande o nodo não expandido mais raso.
- Implementação: *fringe* como uma fila FIFO.
- Exemplo:



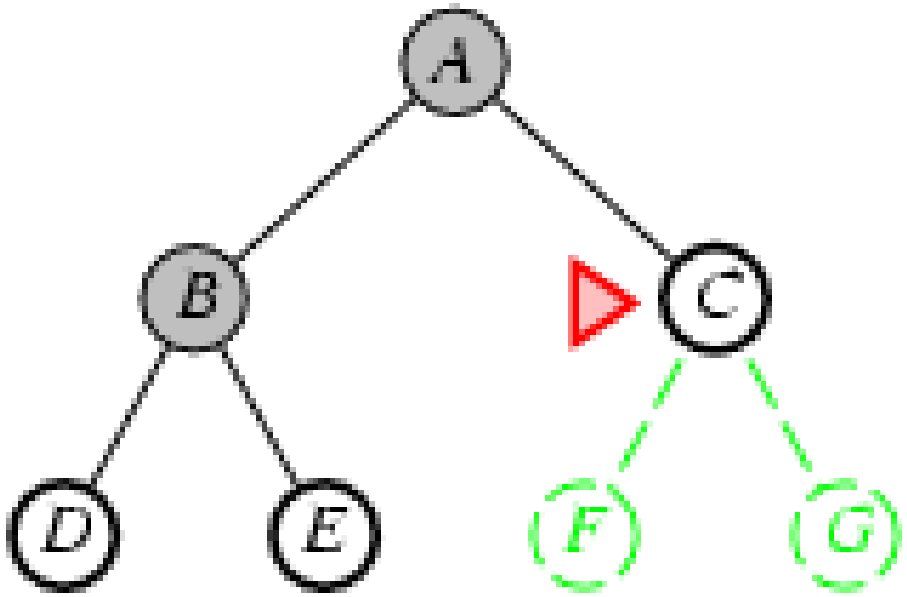
Busca em Largura

- Exemplo:



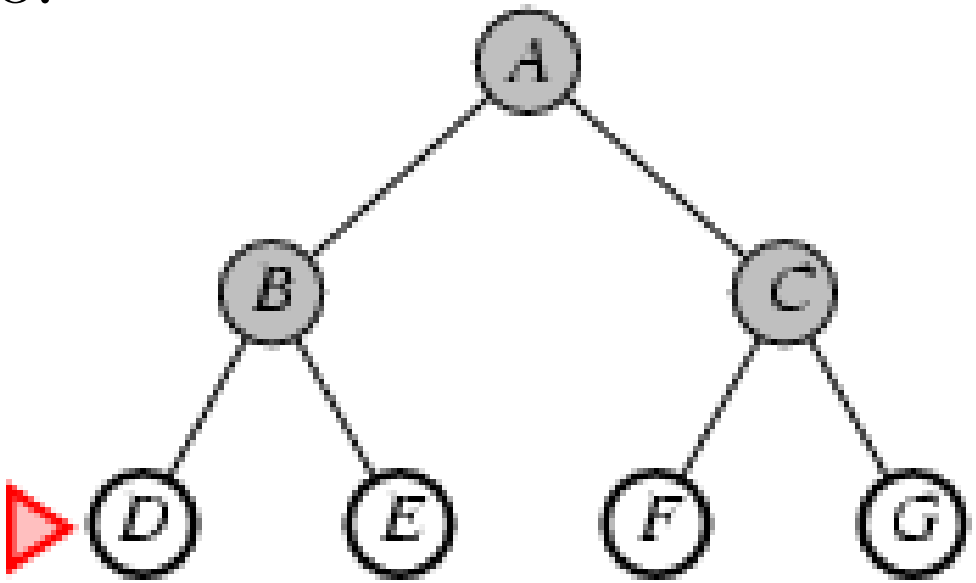
Busca em Largura

- Exemplo:



Busca em Largura

- Exemplo:



Busca em Largura

- Completude:
  - Sempre encontra uma solução?
  - SIM (se existir)
    - Se o nó meta mais raso estiver em profundidade finita *d*.
    - Condição: b finito (máx. no. nodos sucessores finito).
- Otimalidade:
  - Sempre encontra a solução de menor custo ?
    - Apenas se os custos dos caminhos até uma dada profundidade forem iguais e menores do que aqueles para profundidades maiores (ex. se todas as ações possuem o mesmo custo).

Busca em Largura

- Complexidade (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 = $b + b^2 + b^3 + \ldots + b^d + (b^{d+1}-b)$
  - **custo exponencial** = $O(b^{d+1})$.
- Complexidade(espaço)
  - **custo exponencial** = $O(b^{d+1})$.
  - 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

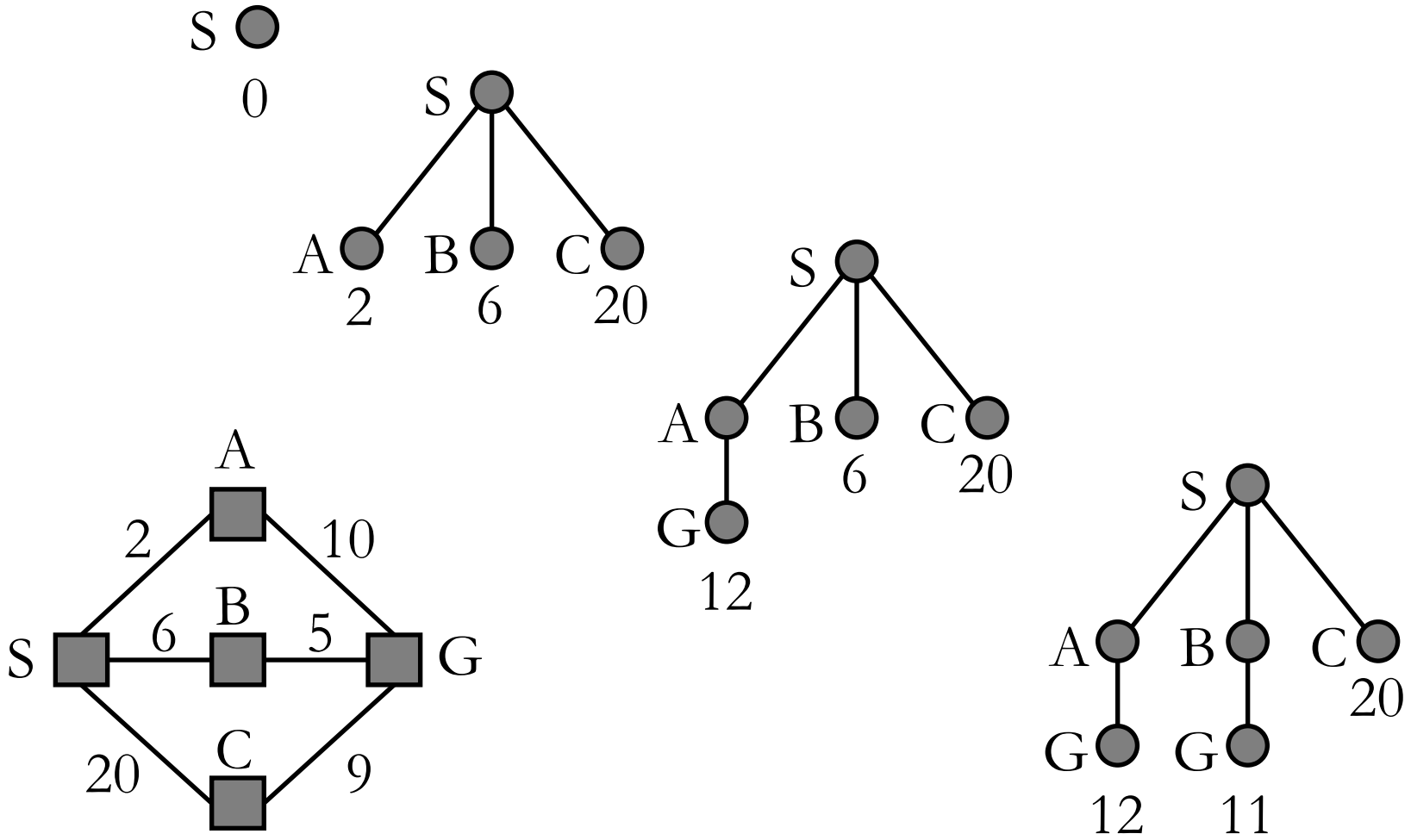
- Esta estratégia só dá bons resultados quando a *profundidade* da árvore de busca é *pequena*.
- Exemplo:
  - fator de expansão b = 10
  - 1.000 nós gerados por segundo
  - cada nó ocupa 100 bytes

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

- Extensão da Busca em Largura:
  - Expande o nodo com o *menor custo de caminho*.
- Implementação:
  - *fringe (borda)* = fila de prioridade com chave dada pelo custo.
- Busca de Custo Uniforme recai na Busca em Largura quando todos os custos de passo (custo das ações) são iguais.
- Completude e Otimalidade:
  - SIM, se o custo de passo for positivo (o que implica que nodos serão expandidos em ordem crescente de custo de caminho).
- Complexidade de tempo e espaço
  - Teoricamente igual à Busca em Largura

Busca de Custo Uniforme

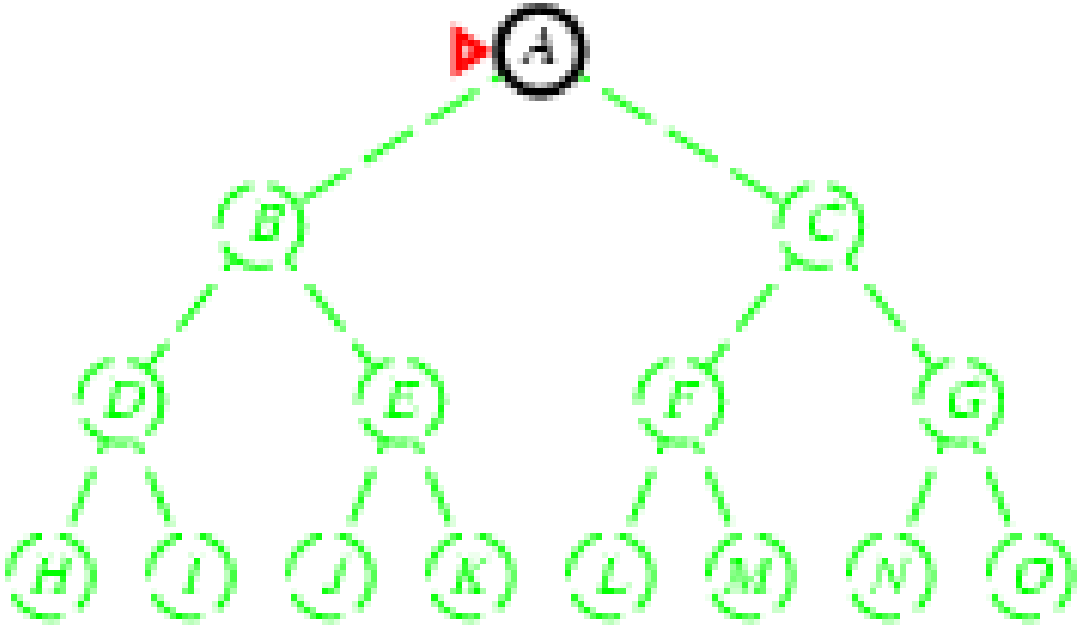


Busca de Custo Uniforme

- 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!

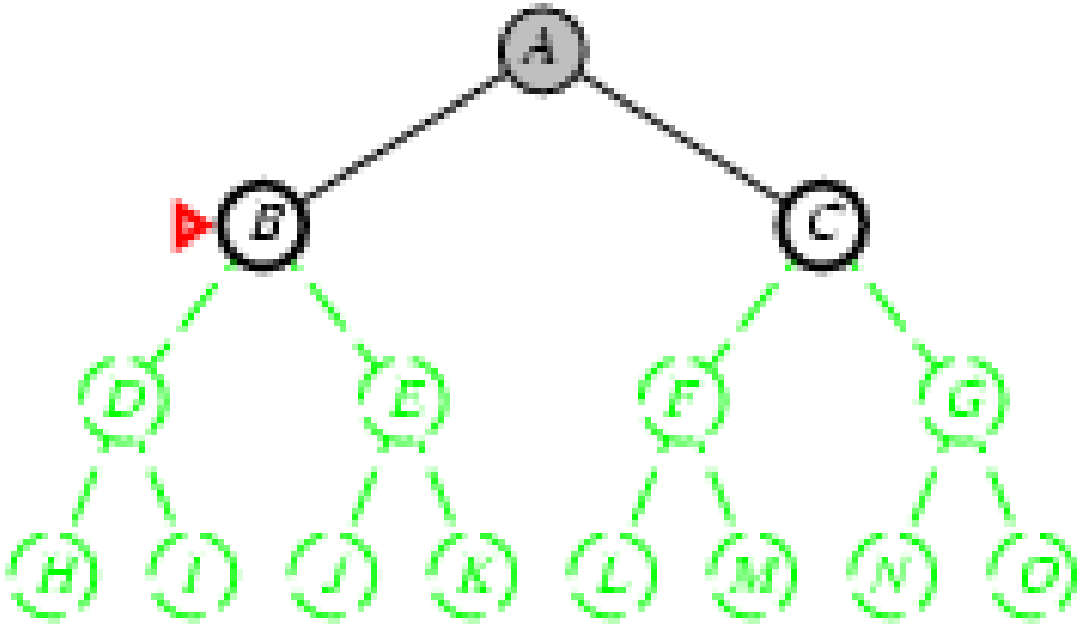
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



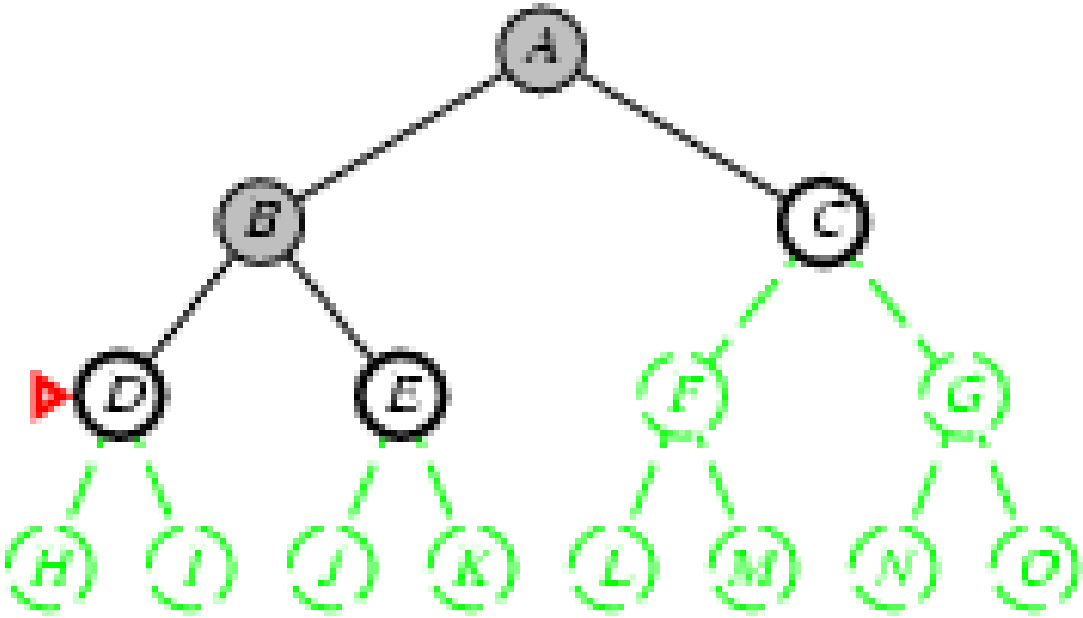
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



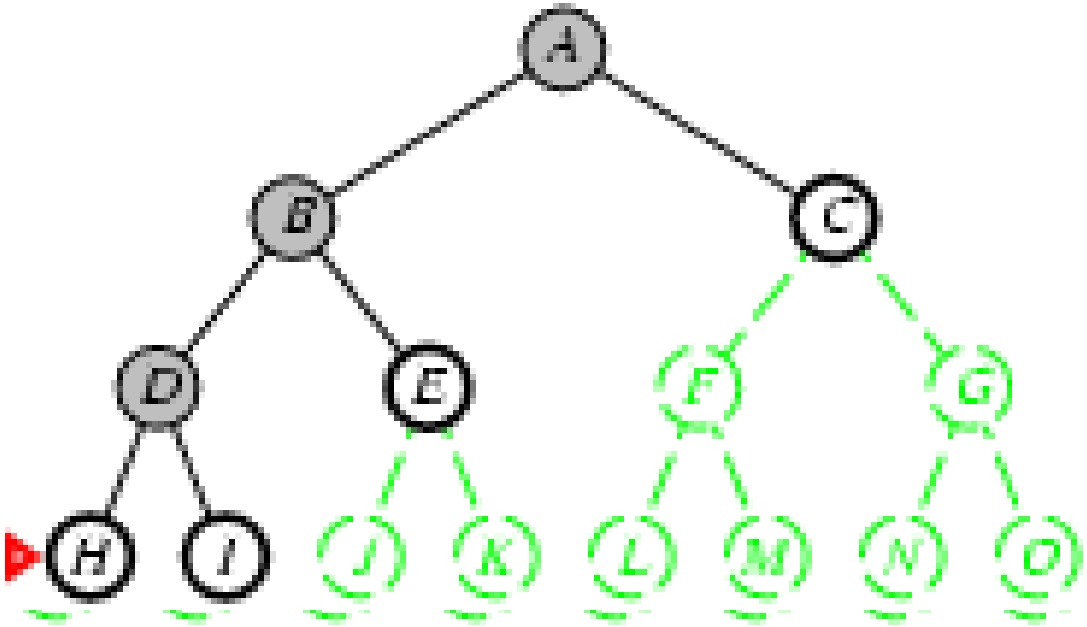
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



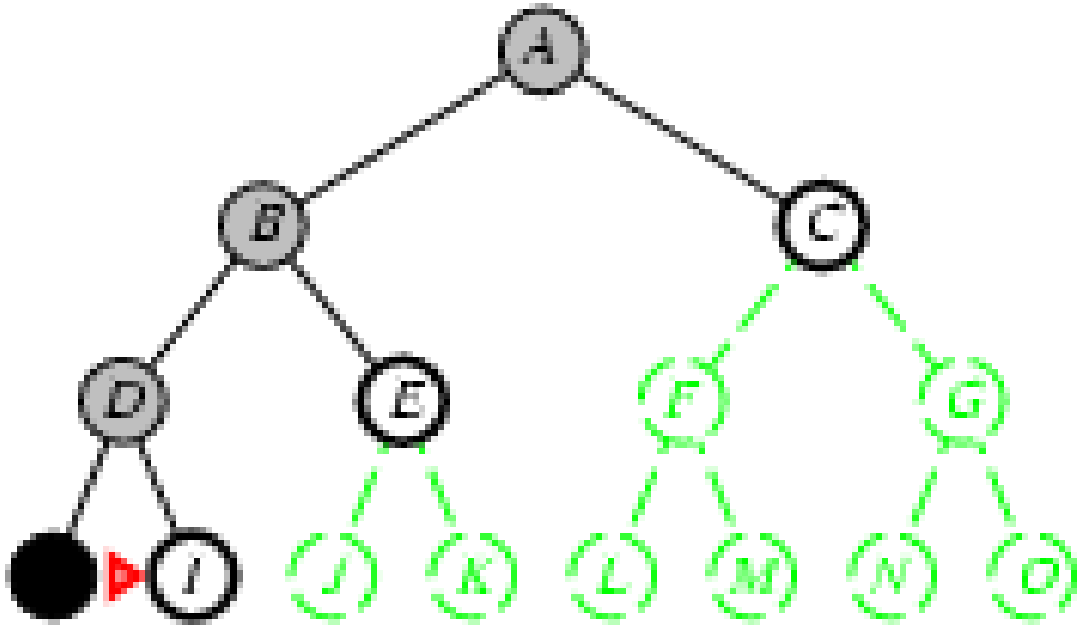
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



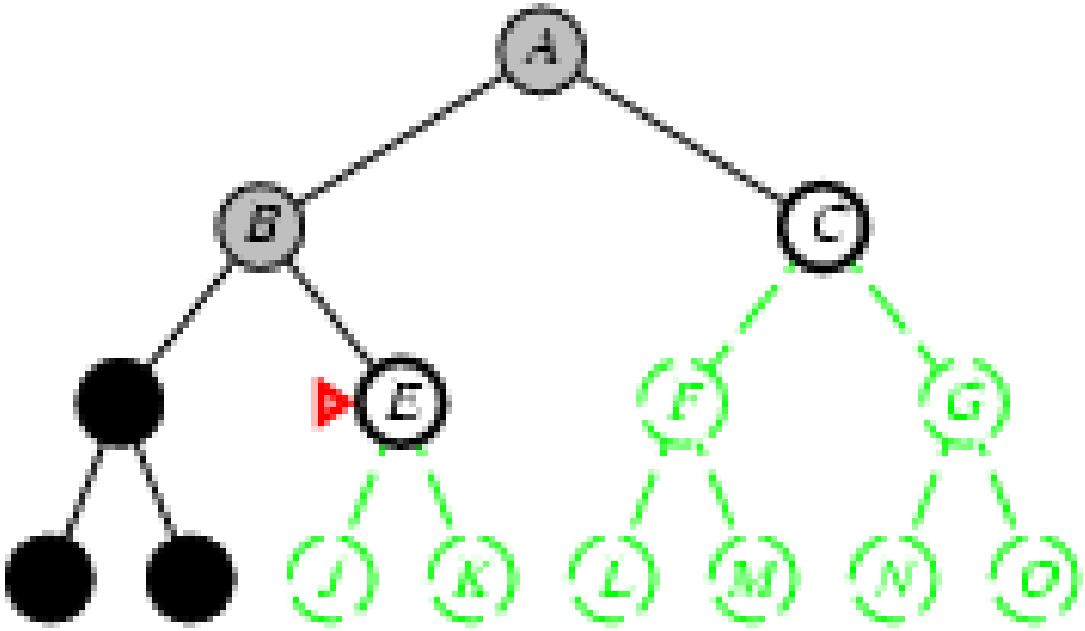
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



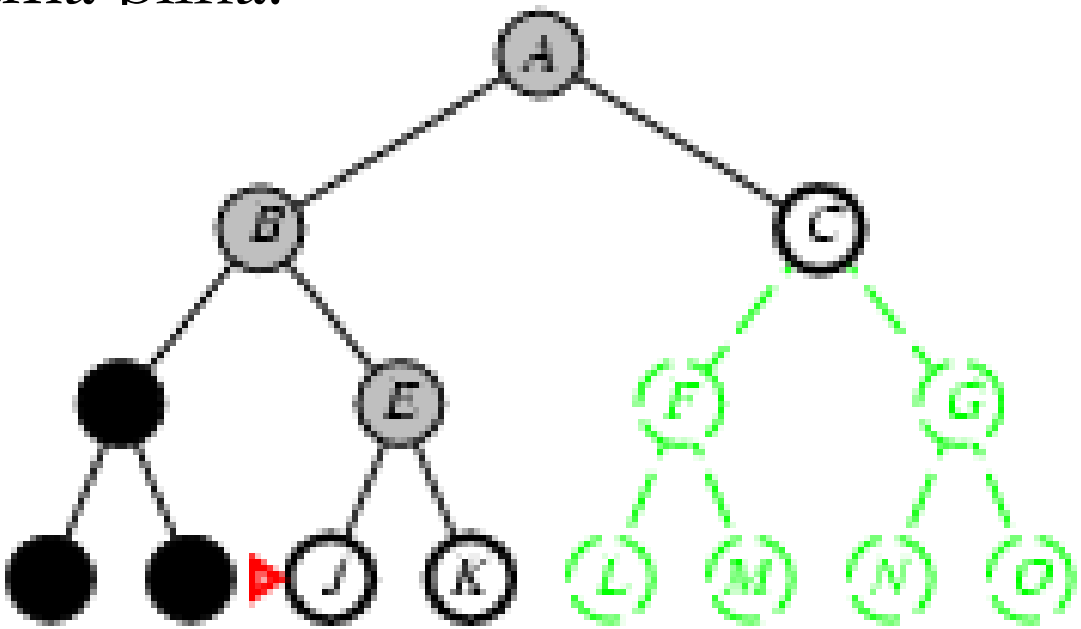
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



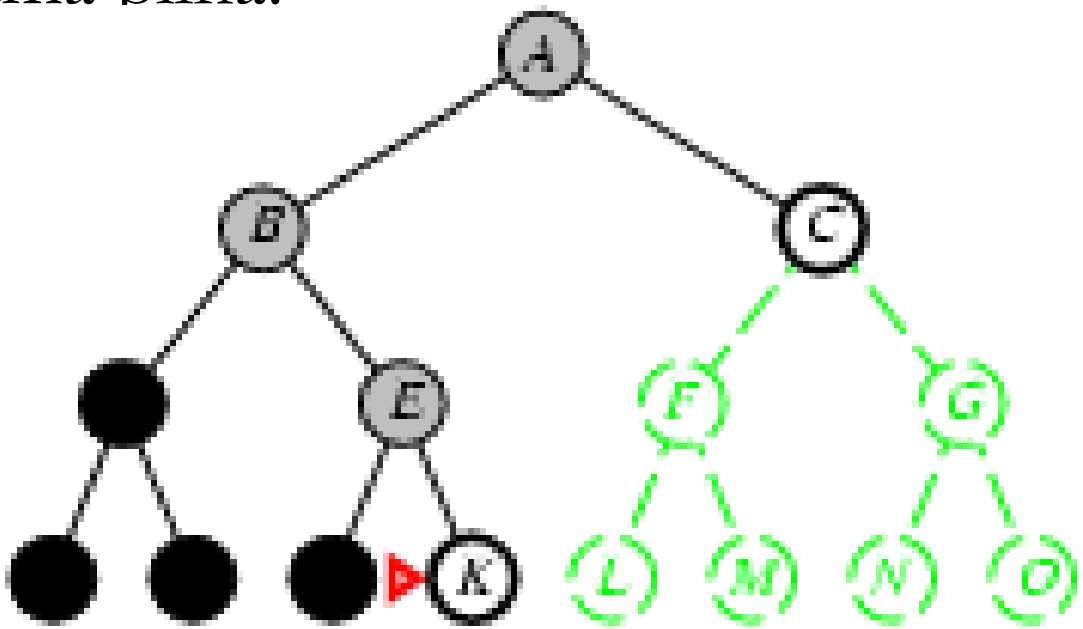
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



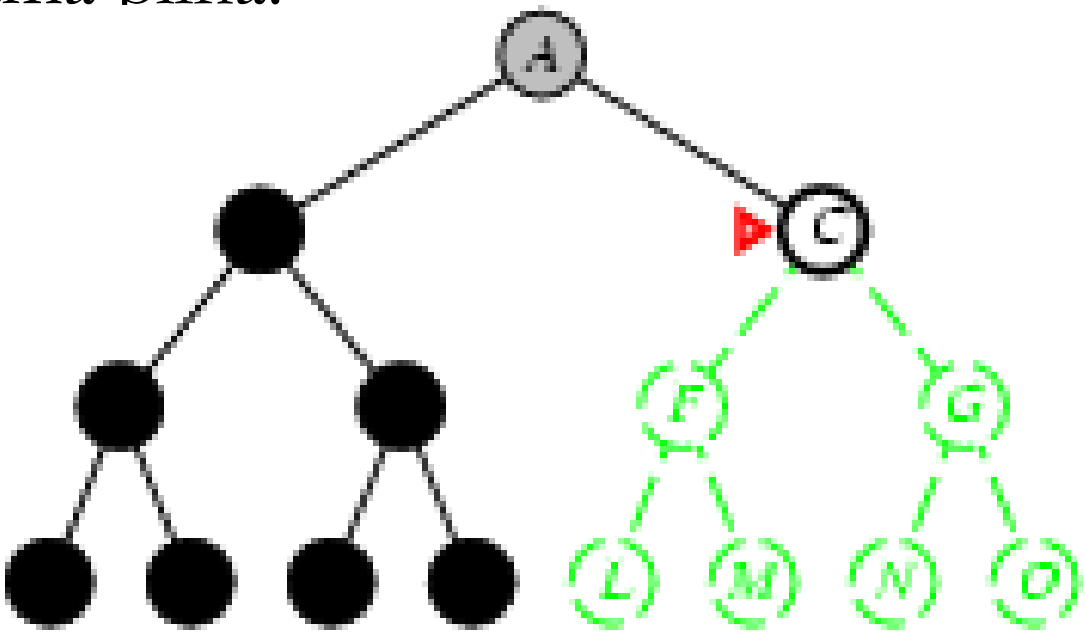
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



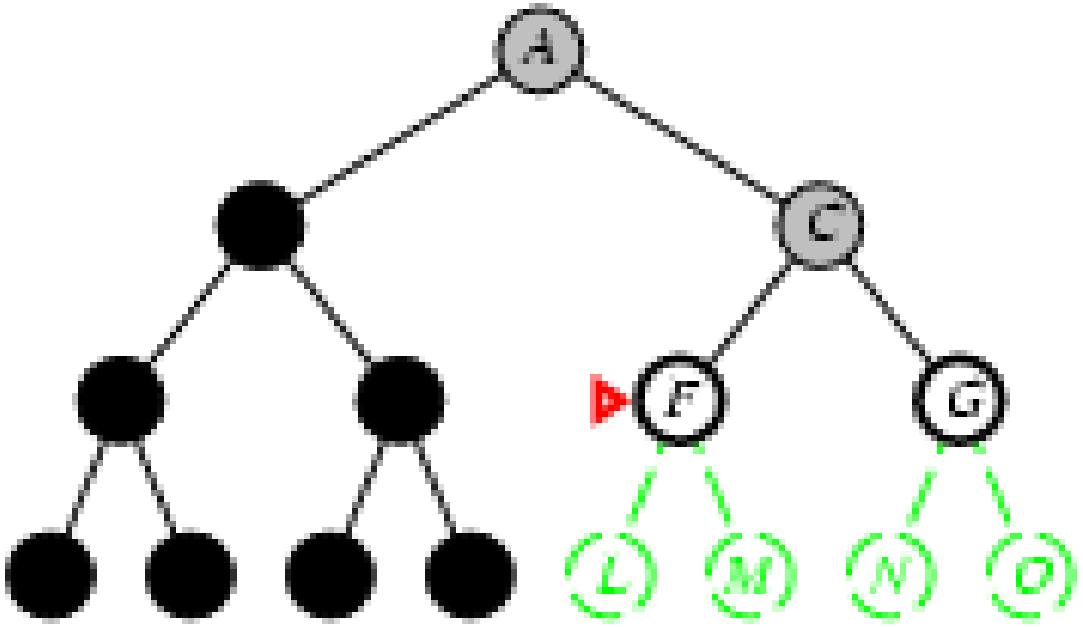
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



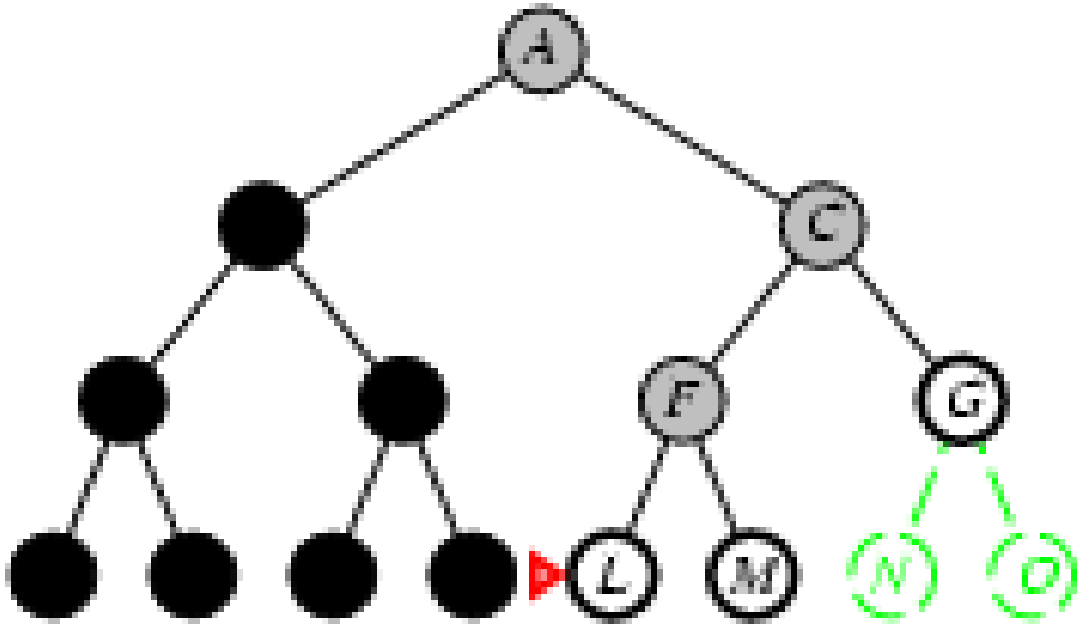
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



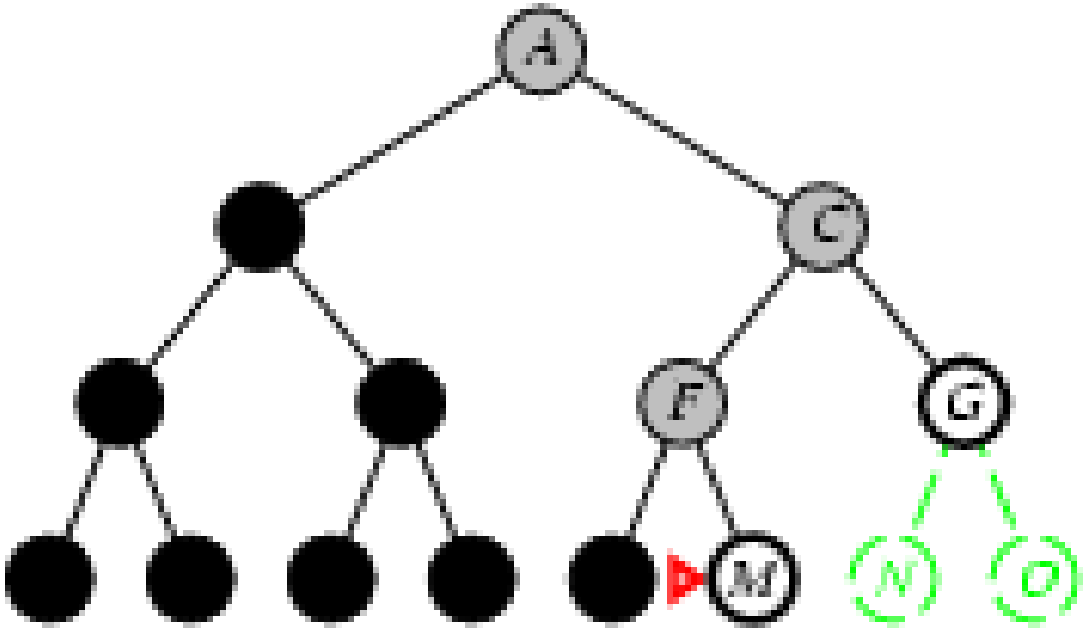
Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



Busca em Profundidade

- Expande o nodo não expandido mais profundo.
- Implementação: *fringe* (nós a serem expandidos) como uma pilha.



Busca em Profundidade

- Completude:
  - Sempre encontra uma solução se existir?
  - Não.
    - Caminho muito longo (infinito) que não contenha a meta pode impedir de que esta seja encontrada em outro caminho.
- Otimalidade:
  - Sempre encontra a solução de menor custo ?
  - Não.

Busca em Profundidade

- Complexidade (tempo)
  - $O(b^m)$: muito ruim, se *m* é muito maior que *d*.
- Complexidade (espaço)
  - O(bm): espaço linear
- Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que Busca em Largura.

Busca em Profundidade Limitada

- Trata-se da busca em profundidade com um limite de profundidade *l*.
  - Ou seja, nodos na profundidade *l* não possuem sucessores.
  - Conhecimento de domínio pode ser utilizado: No mapa da Romênia (20 cidades) qualquer solução → máximo d=19.
- Resolve o problema de árvores infinitas.
- Se $l < d$ então a estratégia não é completa.
- Se $l > d$ a estratégia não é ótima.

Busca em Profundidade Iterativa

- Trata-se de:
  - Uma estratégia geral para encontrar o melhor limite de profundidade *l*.
  - Limite é incrementado até *d* (desconhecido).
    - Meta é encontrada na profundidade *d*, a profundidade do nodo meta mais raso.
- Combina os benefícios das buscas em profundidade (espaço) e largura (possivelmente completude e otimalidade).

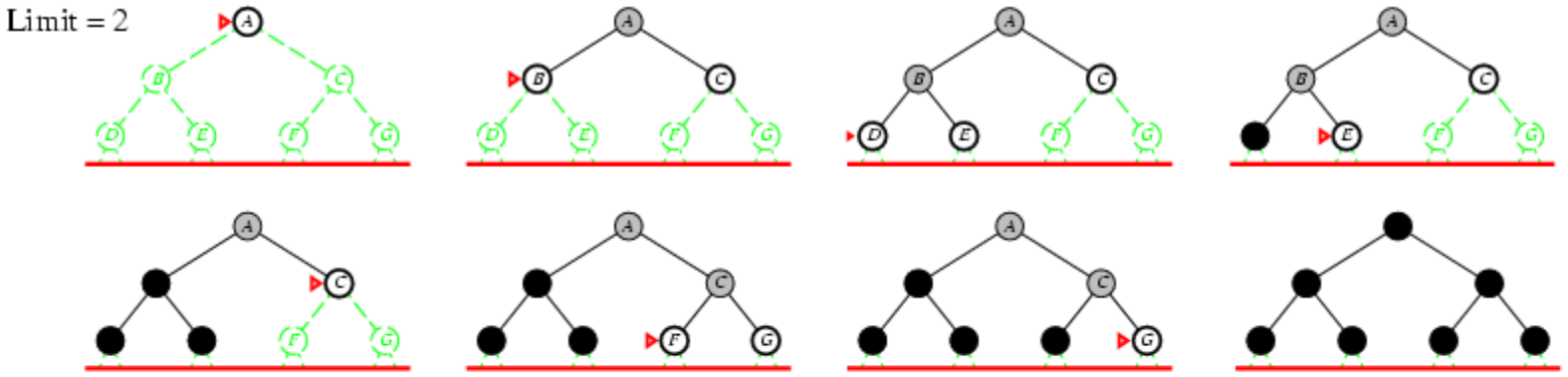
Busca em Profundidade Iterativa

Limit = 0

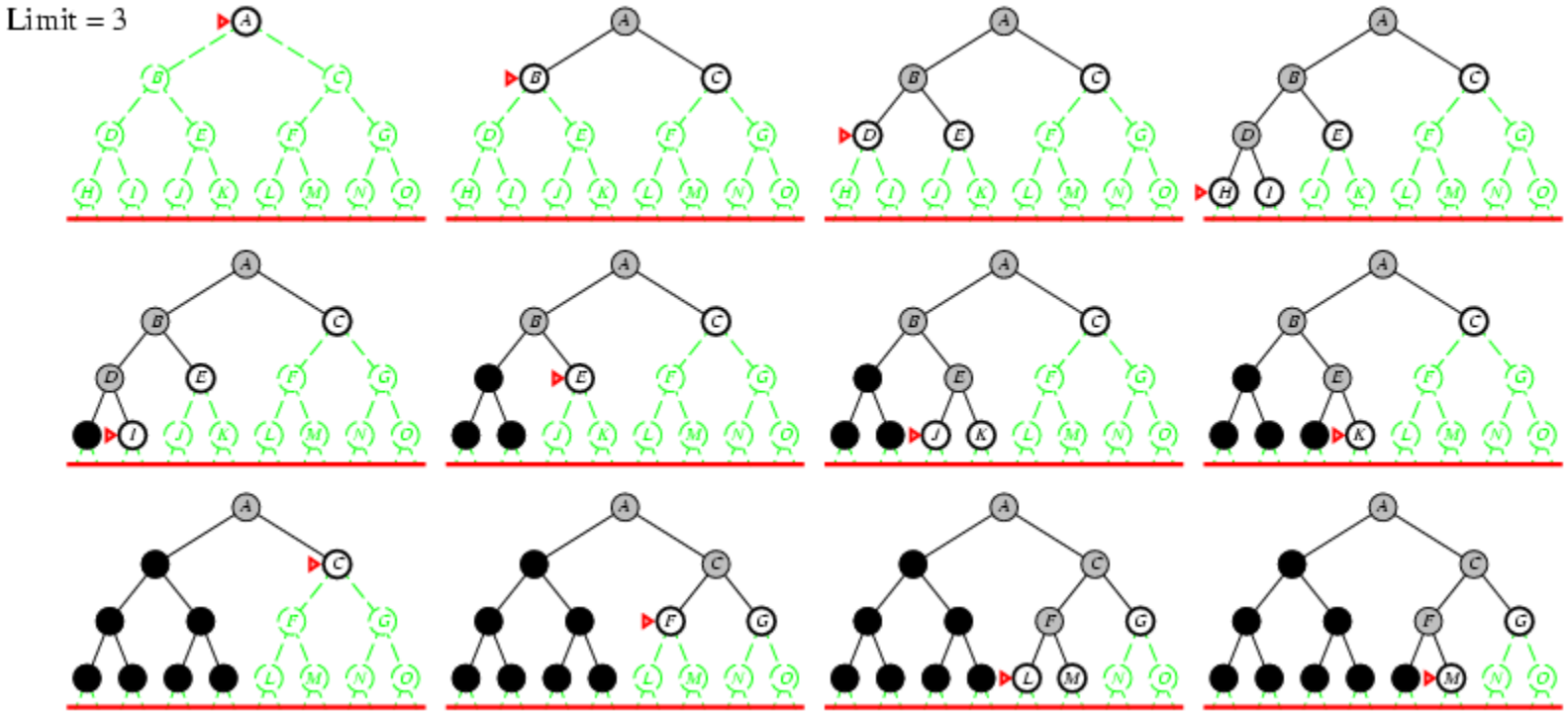
Busca em Profundidade Iterativa

Limit = 1

Busca em Profundidade Iterativa



Busca em Profundidade Iterativa

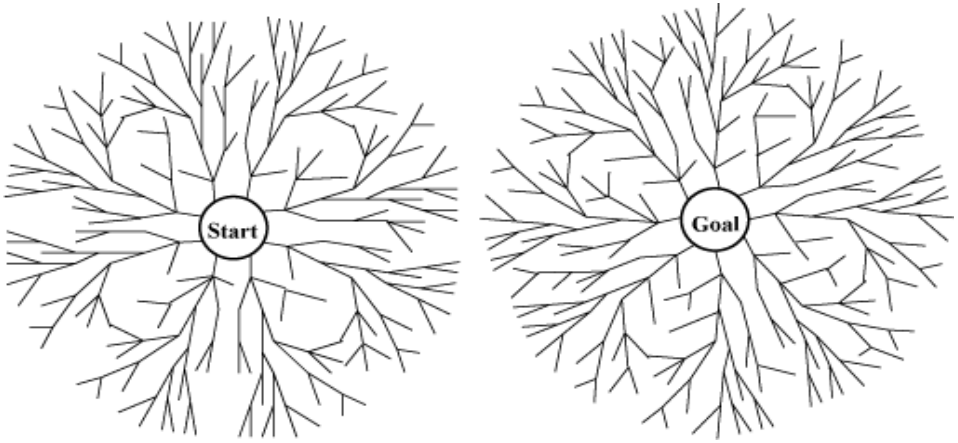


Busca em Profundidade Iterativa

- Busca de Aprofundamento Iterativo
- Semelhantemente à *busca em profundidade*, possui custo de memória reduzido;
- Tal como a *busca em largura:*
  - É completa quando o fator de ramificação é finito;
  - É ótima quando o custo do caminho é uma função não decrescente da profundidade do nó.

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: $O(b^{d/2})$
- 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.

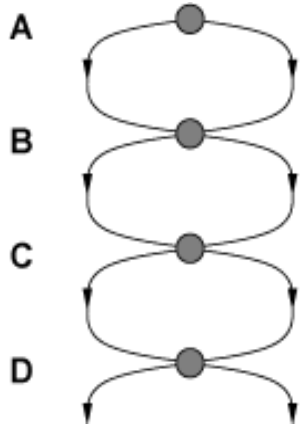
Evitar Geração de Estados Repetidos

- Problema geral em Busca
  - expandir estados presentes em caminhos já explorados
- É inevitável quando existem operadores reversíveis
  - ex. encontrar rotas, canibais e missionários, 8-números, etc.
  - a árvore de busca é potencialmente infinita
- Ideia
  - **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

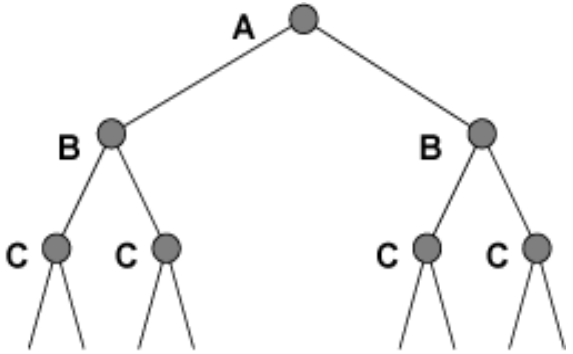
Evitar Geração de Estados Repetidos

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

Espaço de estados



Árvore de busca



- Questão
  - Como evitar expandir estados presentes em caminhos já explorados?

Evitar Geração de Estados Repetidos

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: $O(b^d)$

Tensão (tradeoff) básica

- Problema:
  - Custo de armazenamento e verificação ***X*** Custo extra de busca
- Solução
  - depende do problema
  - quanto mais "loops", mais vantagem em evitá-los!

- *"Algorithms that forget their history are doomed to repeat it."*

Referências

- Stuart Russel e Peter Norvig, **Inteligência Artificial**, 2ª edição, Editora Campus, 2004.