Solução de Problemas por Busca

- *Problem-Solving Agents*
  - Um tipo de *goal based agent* que decide o que fazer procurando sequências de ações que os levem a um objetivo (estado desejado) para um dado problema

- Como formular o problema?
- Como definir o objetivo?
- Como achar a solução?

Solução de Problemas por Busca

- O objetivo vai ser representado por um conjunto de estados
- Ações fazem o agente mudar de um estado para outro
- O agente deve encontrar então um conjunto de ações (de preferência o melhor) que o levem do estado inicial ao estado final
- Para isso, precisa decidir quais ações e estados precisam ser levados em consideração: **Abstração**

Solução de Problemas por Busca

- Para atingir um mesmo objetivo, um agente pode executar diferentes sequências de ações
- O processo mais simples para se encontrar uma dessas sequências é chamado de **busca**
- O agente deve executar as seguintes etapas
  - Formular o problema e o objetivo
  - Realizar uma busca para encontrar a sequência de ações até o objetivo
  - Executar a sequência de ações encontrada

Ambiente

- Em geral, o ambiente onde esses agentes trabalham é:
  - Estático
  - Observável
  - Discreto
  - Determinístico

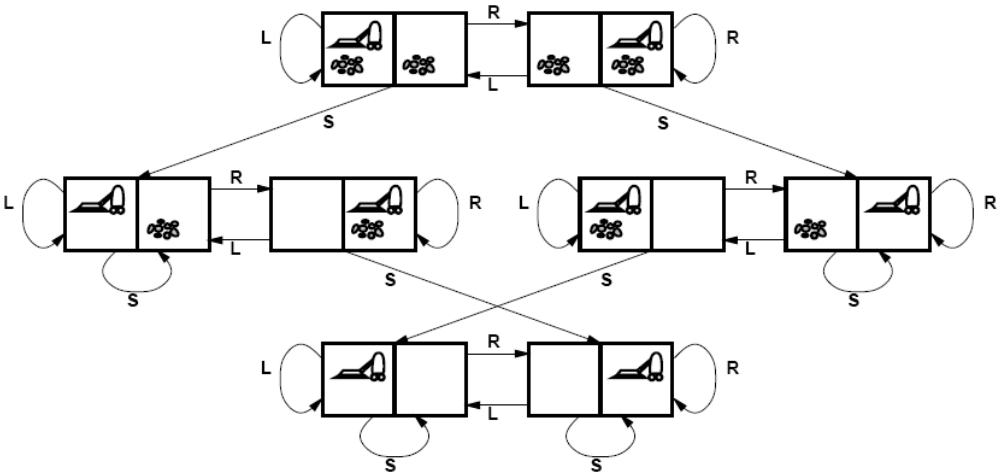
- Além disso, uma vez feito o "plano" o agente o executa sem considerar os *percepts*
  - Controle de Malha Aberta (*Open Loop*)

Formulação do Problema

- A formulação do problema tem 5 componentes
  - Estado inicial
  - Conjunto de ações aplicáveis naquele estado
  - Modelo de Transição (resultados das ações)
    - Função *sucessora* (retorna pares <ação, estado>)
    - O estado inicial juntamente com a função sucessora definem o **espaço de estados** do problema, ou seja todos os estados atingíveis a partir do estado inicial (grafo)
  - Teste do objetivo
  - Custo da solução (do caminho)
    - Normalmente cada ação tem um custo
    - Solução Ótima é aquela cujo caminho tem o menor custo

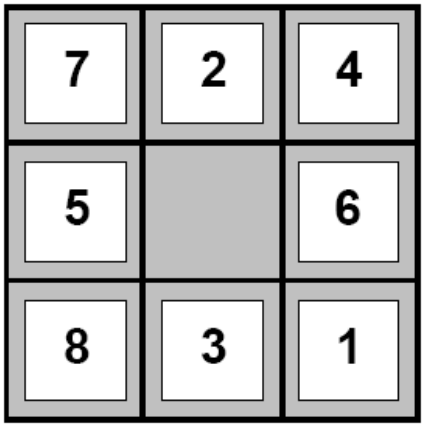
Exemplo: Aspirador de Pó

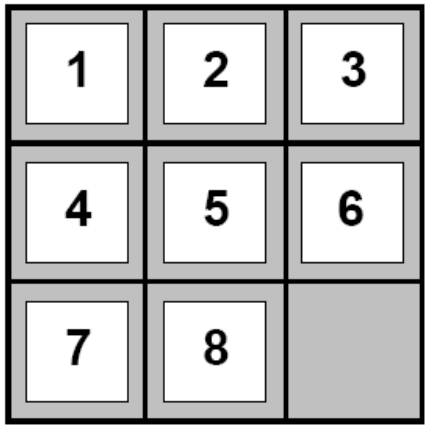
- Estados:
  - o agente pode estar em 1 de 2 locais possíveis, que podem estar sujos ou limpos (8 estados)
- Estado inicial:
  - qualquer um
- Sequência de ações:
  - *esq, dir, aspira*
- Modelo de Transição
- Teste de objetivo:
  - todos os locais limpos
- Custo:
  - cada ação tem custo 1



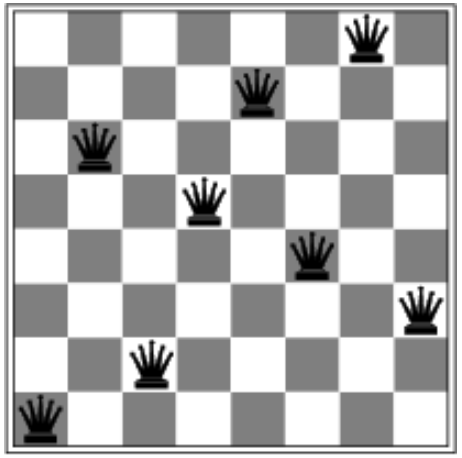
Exemplo: 8-Puzzle

- Estados
  - Posição de cada uma das peças e do espaço no tabuleiro
- Estado Inicial
  - Qualquer um
- Ações / Modelo de Transição
  - Cima, Baixo, Dir, Esq alteram o estado "movendo" o quadrado vazio
- Teste do objetivo
  - Checar se a configuração foi atingida
- Custo
  - Cada ação tem custo 1





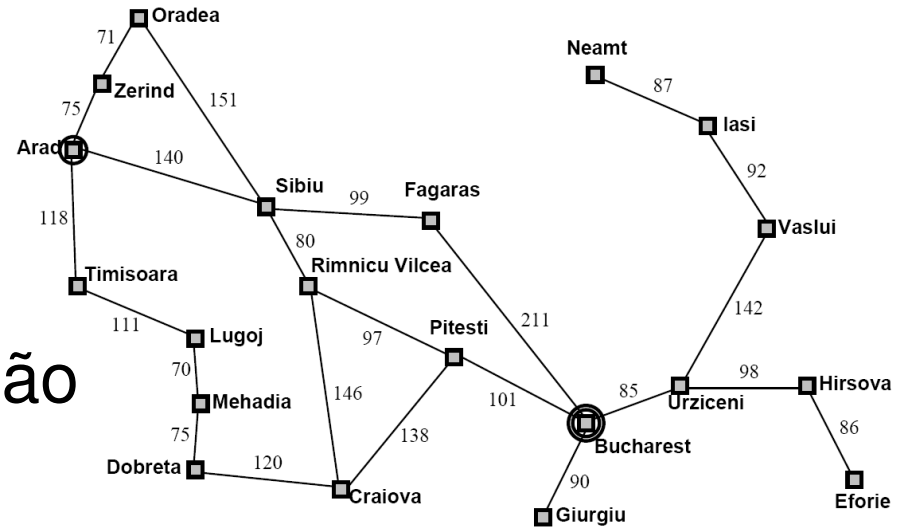
Exemplo: 8-Queens



- Estados:
  - Qualquer combinação de 8 rainhas no tabuleiro
- Estado Inicial
  - Tabuleiro Vazio
- Ação / Modelo de Transição
  - Colocar uma rainha em uma posição vazia
- Teste objetivo
  - 8 Rainhas não se atacando
- Custo do caminho não importa nesse caso

Exemplo: Viagem na Romênia

- Estados
  - Conjunto de cidades
- Estado Inicial
  - Arad
- Ações / Modelo de Transição
  - Dirigir de uma cidade para outra
- Teste do Objetivo
  - Estar em Bucareste
- Custo
  - Distância entre as cidades



Outros exemplos reais

- TSP
- VLSI Design
- Robot Navigation
- Assembly sequencing
- Internet Search
- ...

Busca no Espaço de Estados

- Árvore de busca
  - Construída com a aplicação sucessiva da função sucessora a partir do estado inicial
  - Pode ser muito maior que o espaço de estados
- Nó ≠ Estado
- Um nó da árvore guarda várias informações
  - Estado, Pai, Ação, Custo, Profundidade

Busca no Espaço de Estados

- **Expansão** de um nó
  - Aplicar a função sucessora naquele nó
  - A decisão de qual nó expandir define a estratégia de busca

- Borda *(fringe)* ou Fronteira ou *Open list*
  - Conjunto de nós que foram gerados mas ainda não expandidos

Estados Repetidos

- A presença de estados repetidos pode tornar o espaço de estados infinito
- São necessárias técnicas para evitar a expansão desses estados
- Testar se o estado já foi expandido
  - Só funciona se eles são mantidos na memória
    - Graph Search: listas *Open* e *Closed (ou explored)*

"Algorithms that forget their history are doomed to repeat it"

Tree x Graph Search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Estratégias de Busca sem Informação

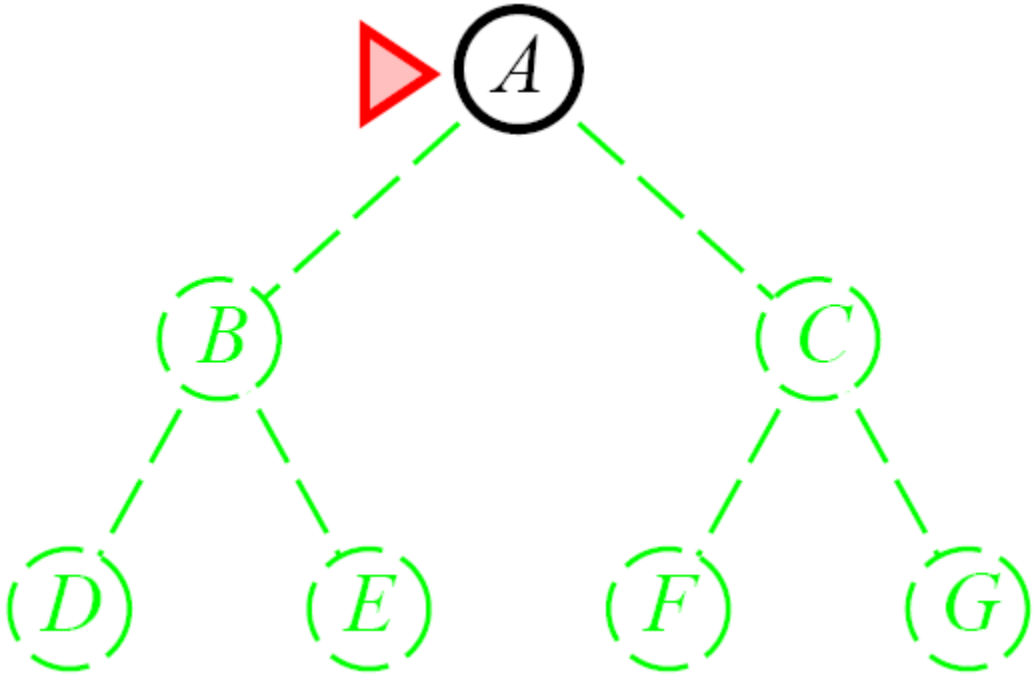
- *Blind Search or Uninformed Search*
  - Usa somente a informação sobre os estados fornecida na formulação do problema
- Algoritmos:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

Como avaliar os algoritmos

- Algoritmo Completo
  - Acha a solução se ela existir
- Algoritmo Ótimo
  - A solução encontrada é a de menor custo
- Complexidades de Espaço e Tempo
  - *Branching Factor: b*
  - Profundidade da solução mais rasa: *d*
  - Tamanho máximo de um caminho: *m*
- Custo da Busca x Custo Total
  - Custo total = custo da busca + custo da solução

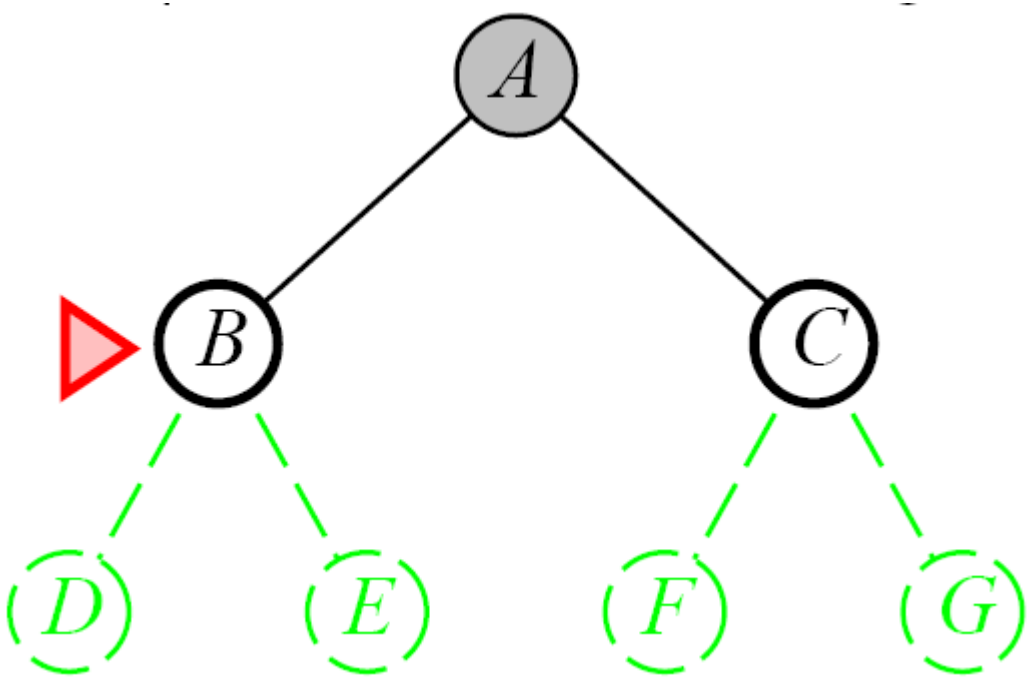
Busca em Largura (BFS)

- Expande o nó mais raso ainda não expandido
  - **Expande a raiz**, depois os sucessores da raiz depois os sucessores dos sucessores...
  - FIFO



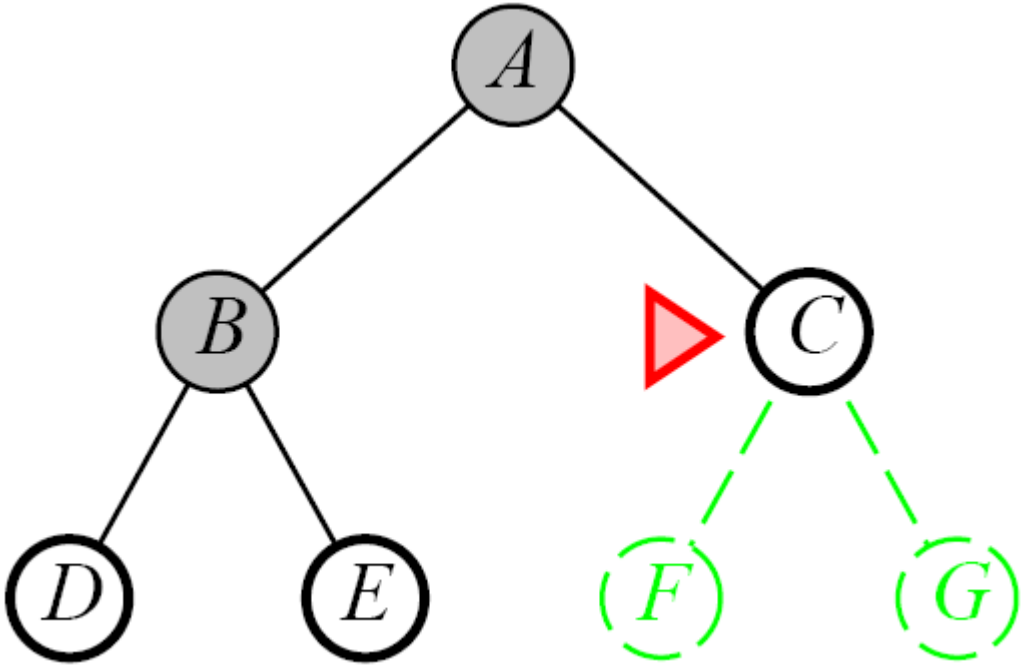
Busca em Largura (BFS)

- Expande o nó mais raso ainda não expandido
  - Expande a raiz, **depois os sucessores da raiz** depois os sucessores dos sucessores...
  - FIFO



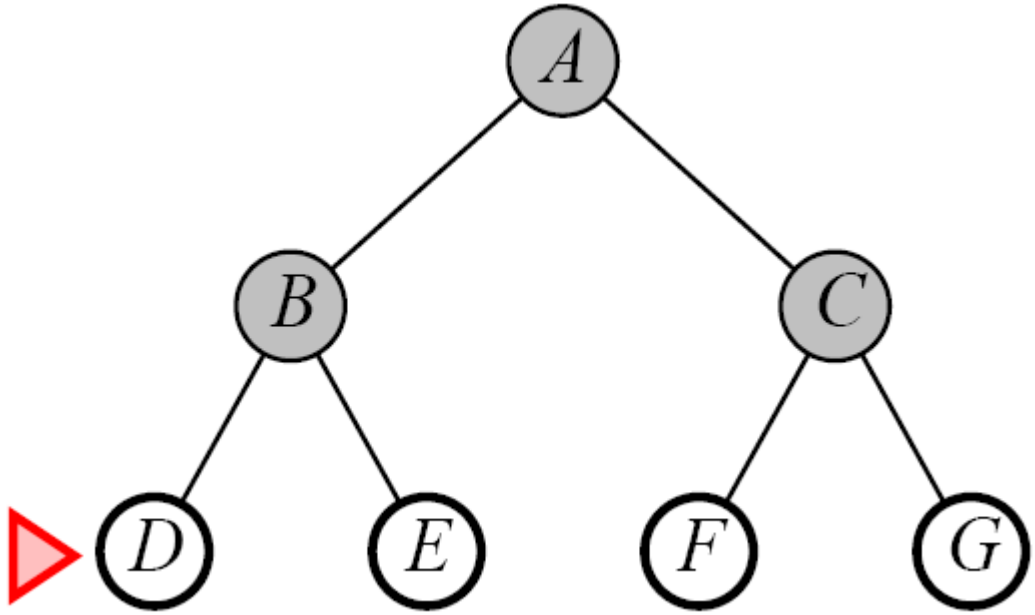
Busca em Largura (BFS)

- Expande o nó mais raso ainda não expandido
  - Expande a raiz, **depois os sucessores da raiz** depois os sucessores dos sucessores...
  - FIFO



Busca em Largura (BFS)

- Expande o nó mais raso ainda não expandido
  - Expande a raiz, depois os sucessores da raiz **depois os sucessores dos sucessores**...
  - FIFO



BFS usando Graph Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Busca em Largura (BFS)

- Completo: **sim**
- Ótimo: **sim, desde que...**
  - **...** o custo seja uma função não decrescente da profundidade do nodo (ex. custo 1 para cada passo)
- Complexidade
  - Considerando *branch factor b,* solução no nível *d*
  - Tempo: $b+b^2+b^3+\ldots + b^d = O(b^d)$
  - Espaço: igual (na verdade, +1 para a raiz), pois todos os nós são mantidos na memória
    - Espaço é mais problemático que o tempo

Busca de Custo Uniforme

- Semelhante ao BFS, mas expande o nodo de menor custo até o momento
- Uso de uma fila de prioridades (Heap)
- Completo (se cada passo tem um custo ≥ ε)
- Ótimo (segue o menor custo)
- Complexidade
  - Considerando C^* como o custo da solução ótima e que cada passo tem um custo de pelo menos ε, no pior caso:
  - $O(b^{1+C^*/\varepsilon})$

Busca de Custo Uniforme

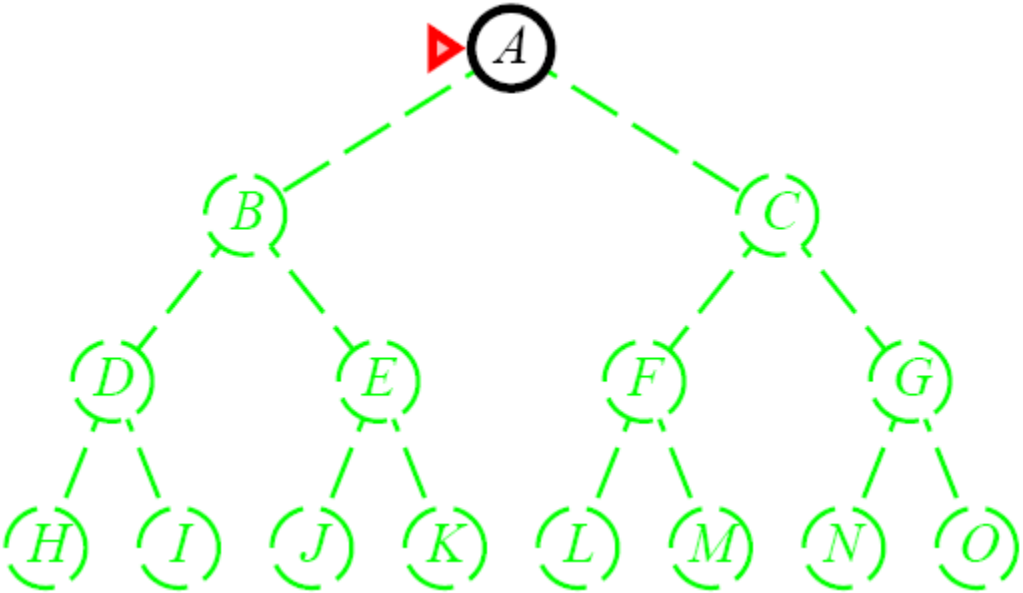
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure ??, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

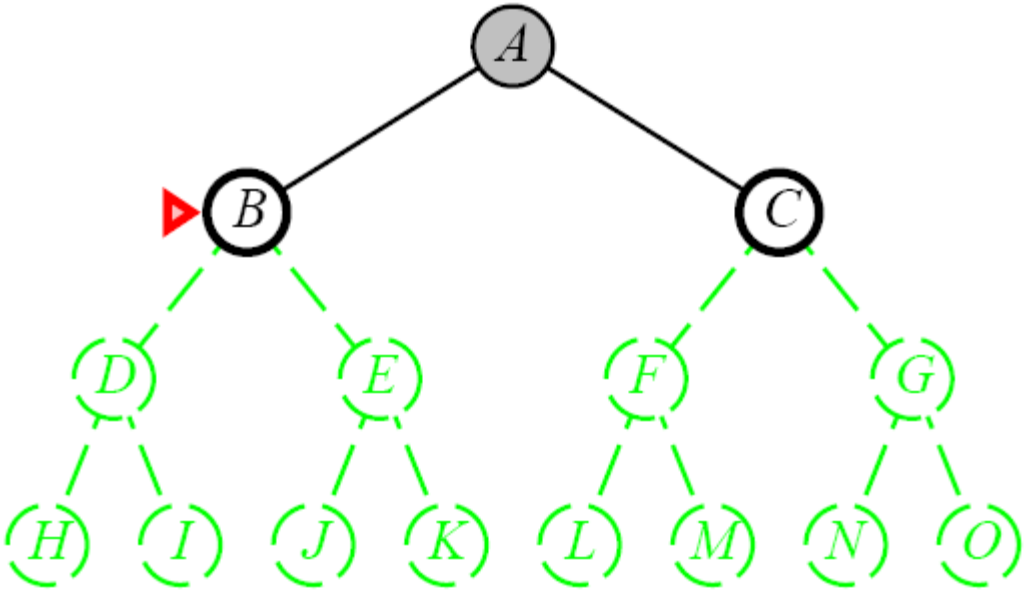
Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



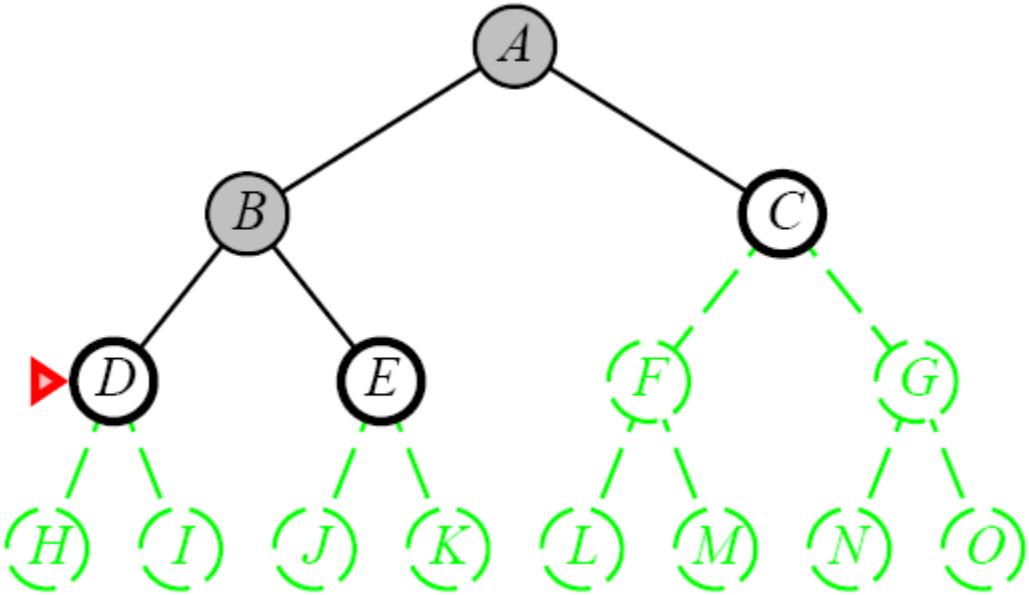
Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



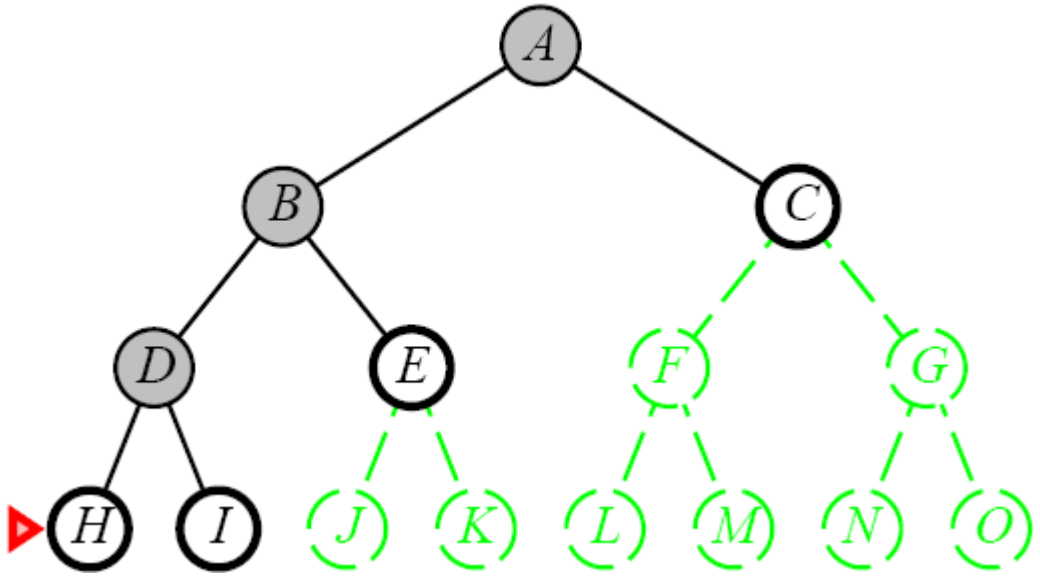
Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



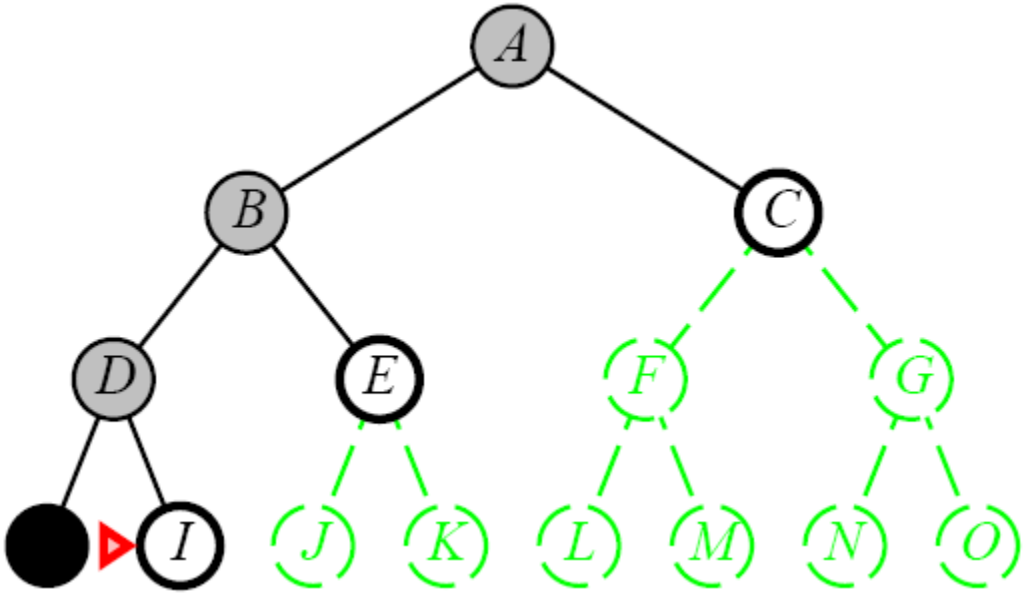
Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



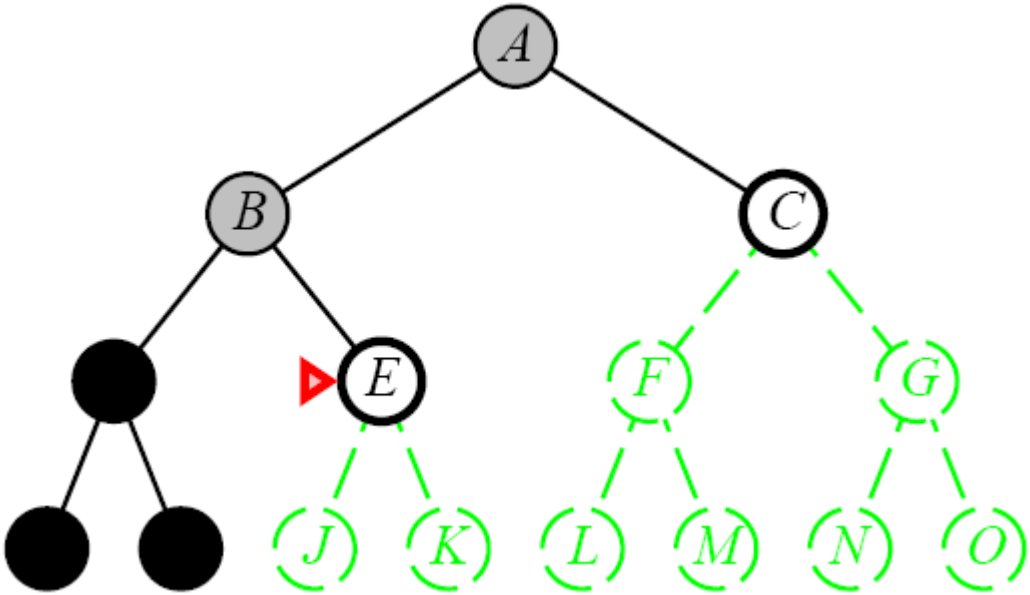
Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



Busca em Profundidade (DFS)

- Expande o nó mais profundo
- LIFO



Busca em Profundidade (DFS)

- Completo: **não**
  - Falha com caminhos infinitos ou *loops*
- Ótimo: **não**
- Complexidade
  - Considerando *branch factor b* e *m* como a profundidade máxima
  - Espaço: O(bm): **linear!**
  - Tempo: $O(b^m)$, no pior caso (pode ser terrível se m >> d)
- Variação: *Backtracking search*

Busca em Profundidade Limitada

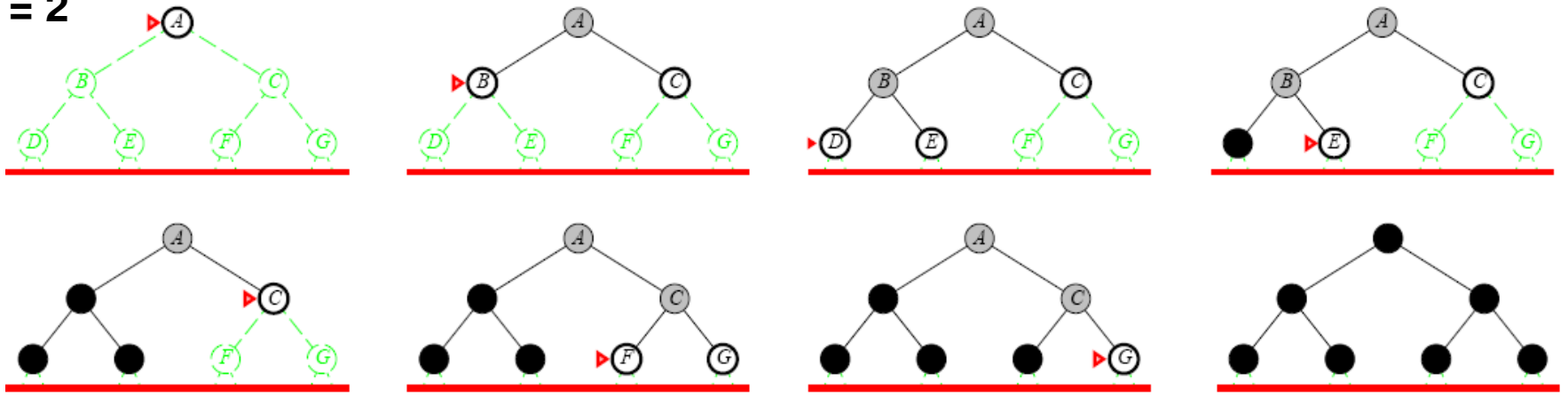
- Determinar uma profundidade máxima *L*, no qual a DFS deve parar.
- Interessante quando há um conhecimento maior sobre o espaço de solução
- Completa: **sim**, se *L* >= *d*
- Ótima: **não**
- Complexidade
  - Espaço: O(bL)
  - Tempo: $O(b^L)$,

Aprofundamento Iterativo (IDS)

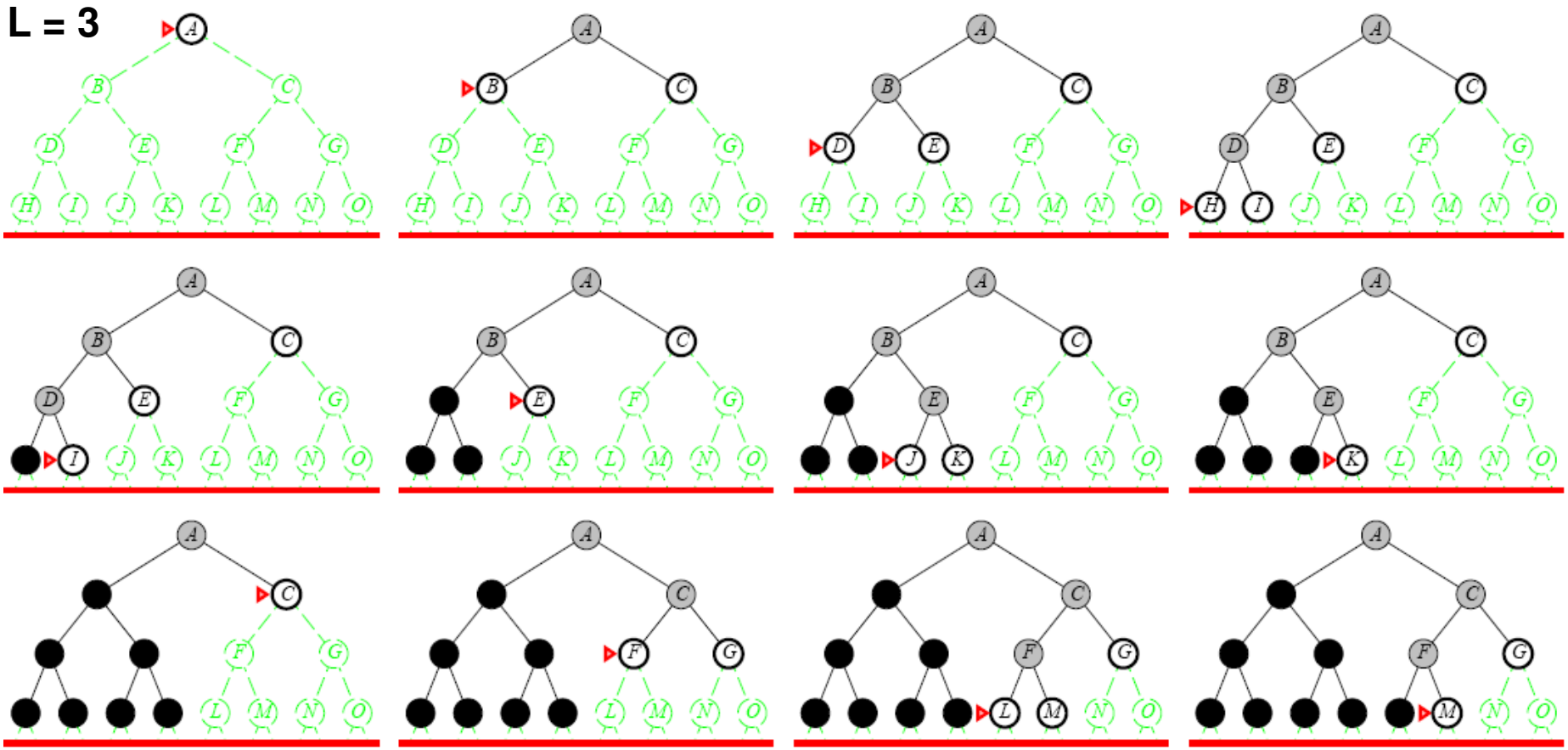
- Faz repetidas buscas em profundidade, aumentando o limite a cada iteração

L = 1

L = 2



Aprofundamento Iterativo (IDS)

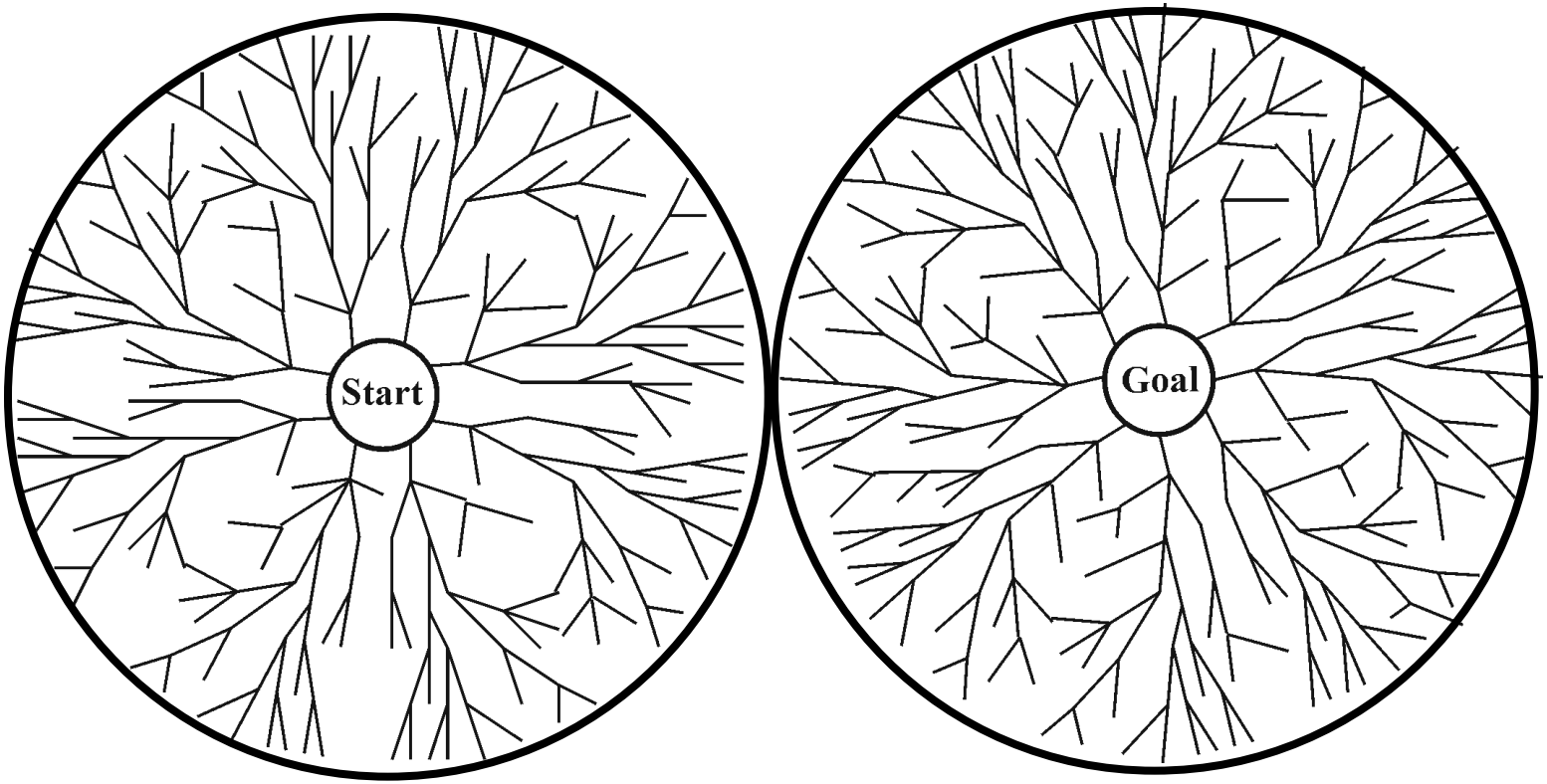


Aprofundamento Iterativo (IDS)

- Combina os benefícios da BFS e DFS
  - Completo e Ótimo (considerando custo crescente)
  - Não gasta muita memória
- Repetição da expansão de estados...
  - Não é tão ruim, pois a maior parte dos estados está nos níveis mais baixos
  - IDS: $(d)b+(d-1)b^2+(d-2)b^3+\ldots +(1)b^d$
  - Ex b=10, d=5:
    - N(IDS) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450
    - N(BFS) = 10 + 100 + 1.000 + 10.000 + 100.000 = 111.110

Busca Bidrecional

- Duas buscas em paralelo, uma começando na raiz e outra no objetivo



Busca Bidirecional

- Complexidade de Espaço e Tempo
  - $O(b^{d/2})$
- Ótimo e Completo dependendo do algoritmo usado (por exemplo BFS)

- Problema comum: Como “andar para trás”?
  - Gol único: simples
  - Múltiplos gols: cria-se um estado predecessor “dummy”
  - Gols “implícitos”: complexo. Ex. Checkmate

Comparação

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes