Resolução de Problemas e Pesquisa

Capítulo 3

Sumário

- ♢ Agentes que resolvem problemas (Problem-solving agents)
- ♢ Tipos de problemas
- ♢ Formulação de problemas
- ♢ Exemplos de problemas
- ♢ Algoritmos básico de pesquisa

Agentes que resolvem problemas

Agente geral:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

Nota: Isto é resolução offline de problemas; execução de "olhos fechados"
resolução Online de problemas obriga a agir sem conhecimento completo.

Exemplo: Roménia

Férias na Roménia; actualmente em Arad.
Voo sai de Bucareste amanhã

Formular objectivo:

estar em Bucareste

Formular o problema:

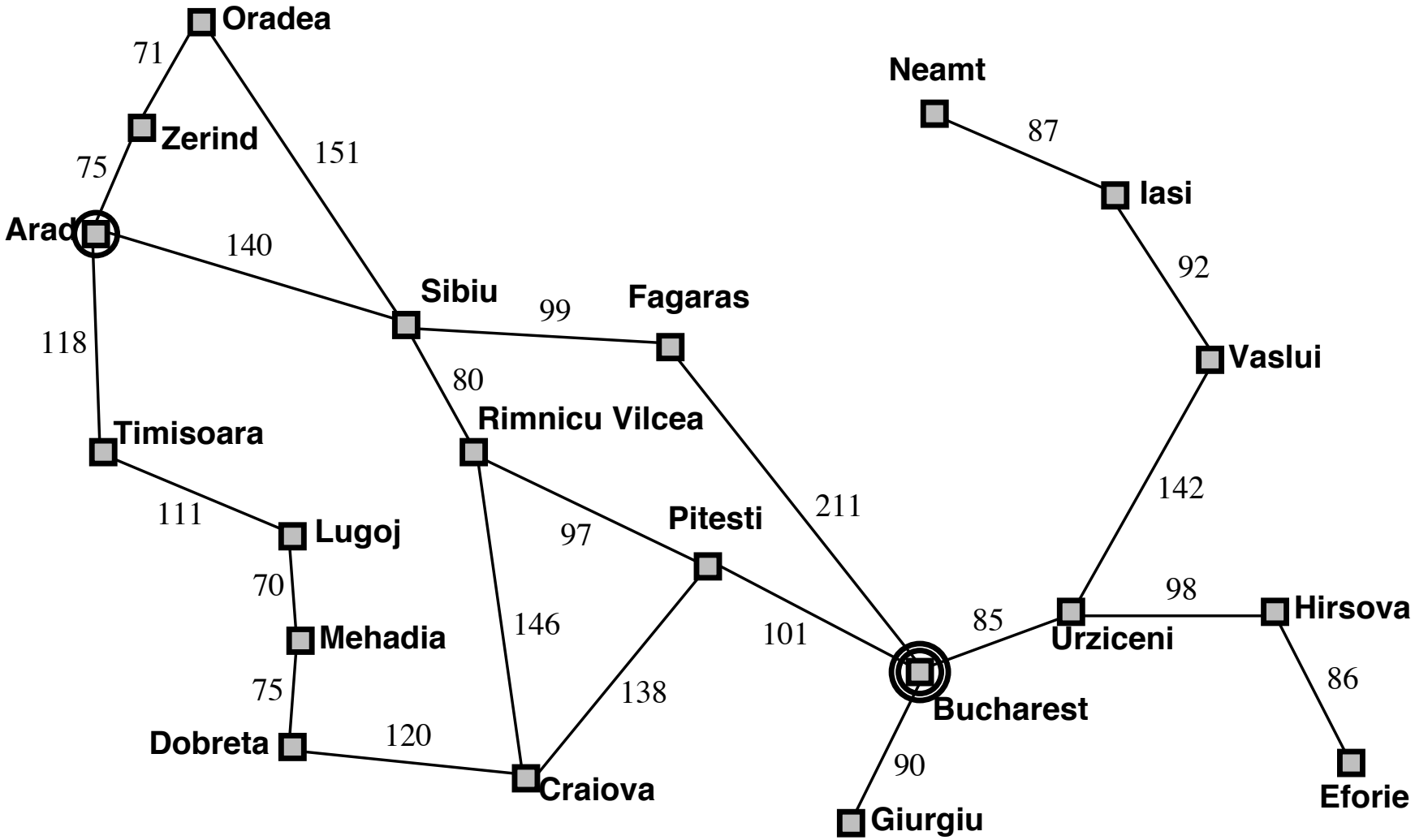
estados: as várias cidades

acções: andar entre as cidades

Encontrar solução:

sequência de cidades e.g., Arad, Sibiu, Fagaras, Bucharest

Exemplo: Roménia



Tipos de problemas

Deterministicos, observáveis $\Longrightarrow$ single-state problem

O agente sabe exactamente em que cidade está, a solução é uma sequência

Não-observável $\Longrightarrow$ conformant problem

O agente pode não ter ideia de onde está; a solução, se existir, é uma sequência

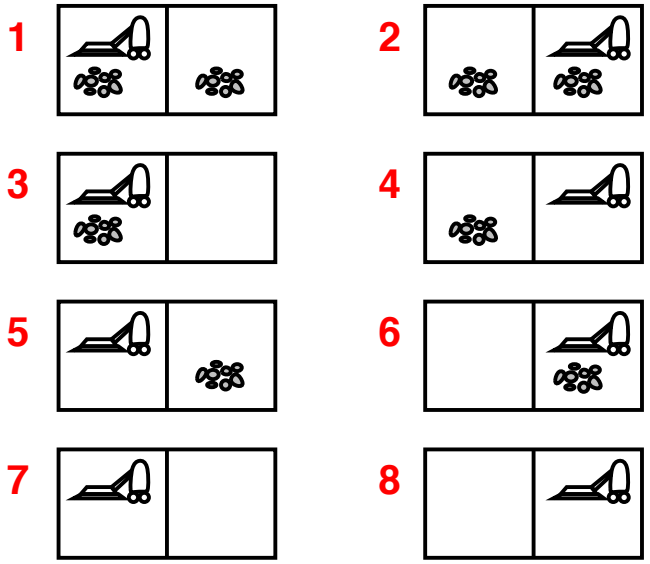
Não deterministico e/ou parcialmente observável $\Longrightarrow$ problema de contingência

as percepções dão **nova** informação sobre o estado corrente
a solução é um plano de contingência ou uma política
muitas vezes **intercala** procura, execução

Espaço de estado desconhecido $\Longrightarrow$ problema de exploração ("online")

Exemplo: mundo do aspirador (vacuum world)

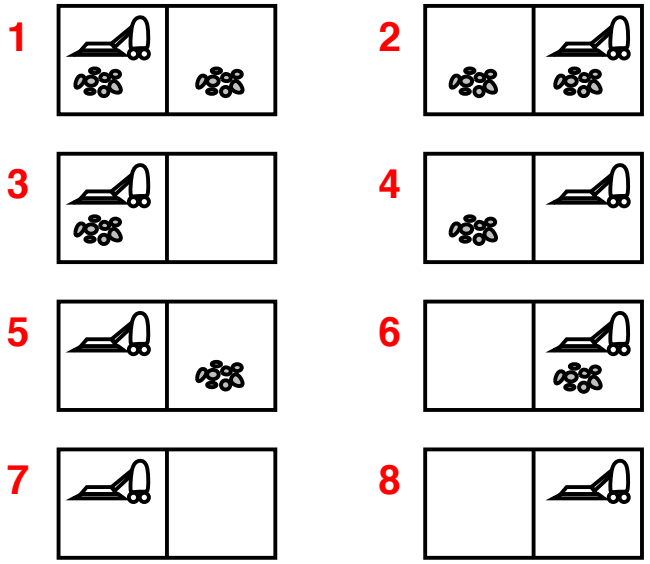
Estado único (Single-state), inicio em #5.
Solução??



Exemplo: mundo do aspirador (vacuum world)

Estado único, inicio em #5. Solução??
$[Direita, Aspira]$

Conformant, inicio em $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Direita$ vai para $\{2, 4, 6, 8\}$. Solução??

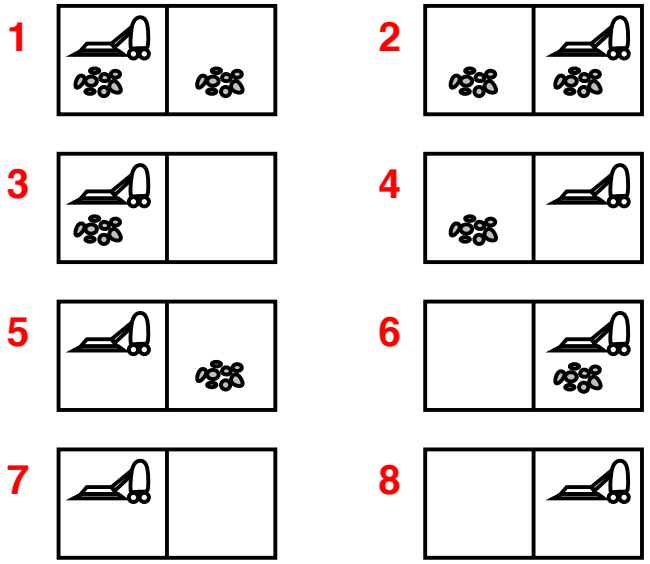


Exemplo: mundo do aspirador (vacuum world)

Estado único, inicio em #5. Solução??
$[Direita, Aspira]$

Conformant, inicio em $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ vai para $\{2, 4, 6, 8\}$. Solução??
$[Right, Suck, Left, Suck]$

Contingência, inicio em #5
Lei de Murphy: $Aspira$ pode sujar uma carpete limpa



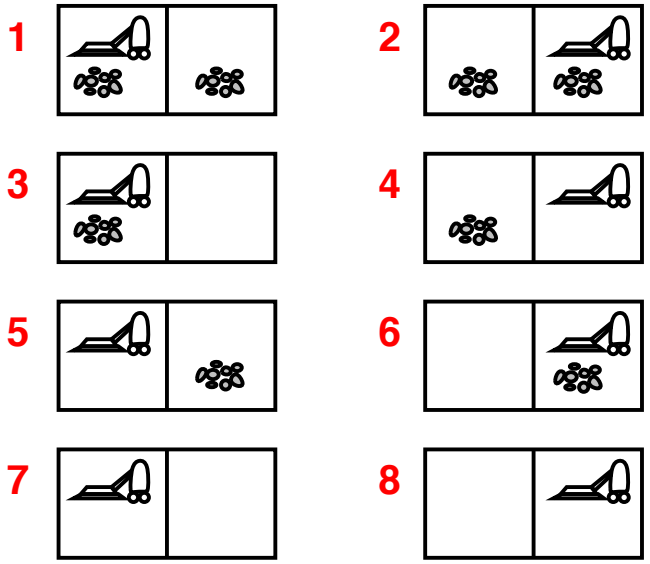
Solução??

Exemplo: mundo do aspirador (vacuum world)

Estado único, inicio em #5. Solução??
$[Direita, Aspira]$

Conformant, inicio em $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ vai para $\{2, 4, 6, 8\}$. Solução??
$[Direita, Aspira, Esquerda, Aspira]$

Contingência, inicio em #5
Lei de Murphy: $Aspira$ pode sujar uma carpete limpa
Solução??
$[Direita, \mathbf{se}\ suja\ \mathbf{entao}\ Aspira]$



Formulação de problemas de estado único

Um problema é definido com:

Estado inicial e.g., "em Arad"

função sucessor $S(x)$ = conjunto de acções–pares de estados
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

estado final, pode ser
explicito, e.g., $x =$ "em Bucharest"
implícito, e.g., $Limpo(x)$

custo do caminho (aditivo)
e.g., soma das distancias, numero de acções executadas, etc.
$c(x, a, y)$ é o custo do passo, assume-se que é ≥ 0

Uma solução é uma sequência de acções
que levam o agente do estado inicial ao estado final

Seleccionar o espaço de estados

O mudo real é muito complexo

$\Rightarrow$ o espaço de estados deve ser **abstraído** para resolver problemas

Estado (Abstraído) = conjunto de estados reais

Acção (Abstraída) = combinação complexa de acções reais

e.g., "Arad $\rightarrow$ Zerind" representa o conjunto complexo

de estradas possíveis, desvios atalhos, etc.

Para garantir a realizabilidade, **qualquer** estado real "em Arad"

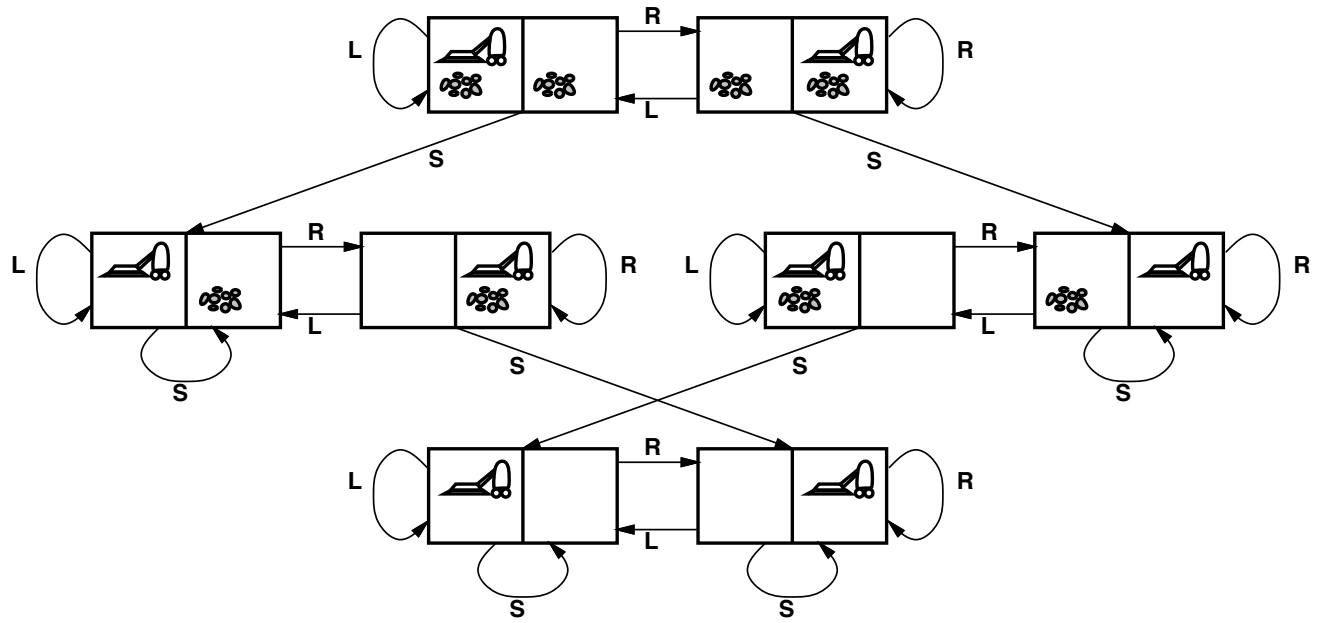
deve chegar a algum estado real "em Zerind"

Solução (Abstraída) =

conjunto de caminhos reais que são soluções no mundo real

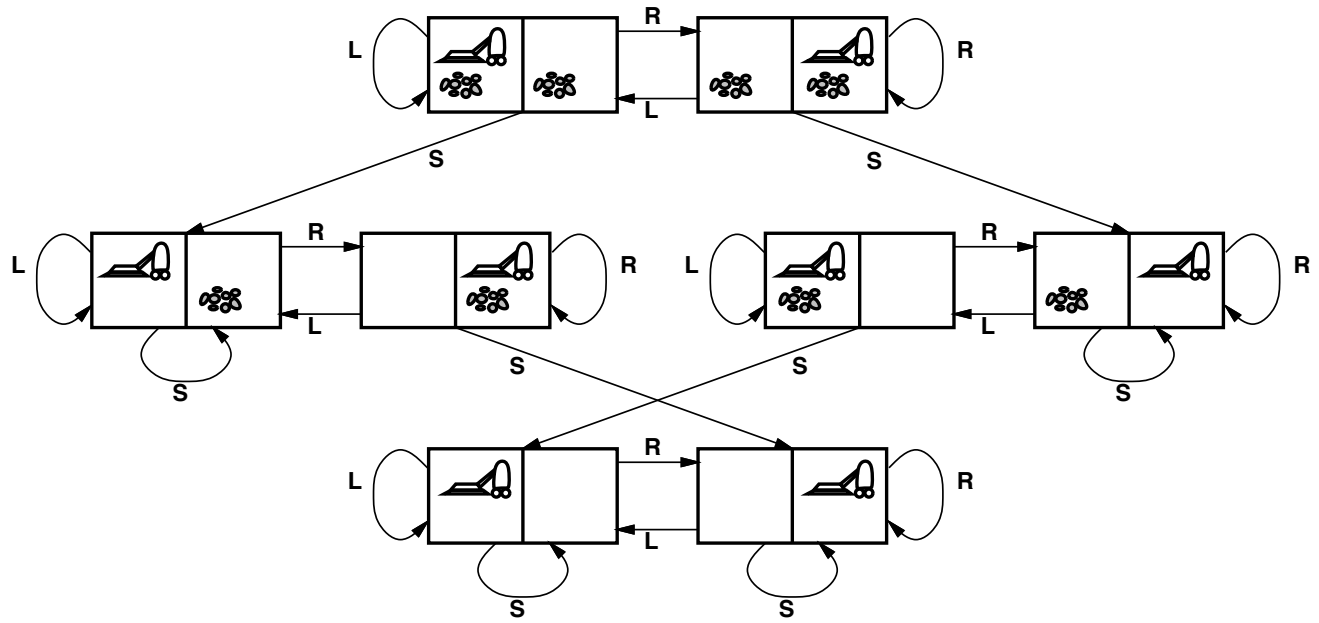
Cada acção abstraída deve ser "mais simples" que as do problema inicial!

Exemplo: grafo do espaço de estados do mundo do aspirador



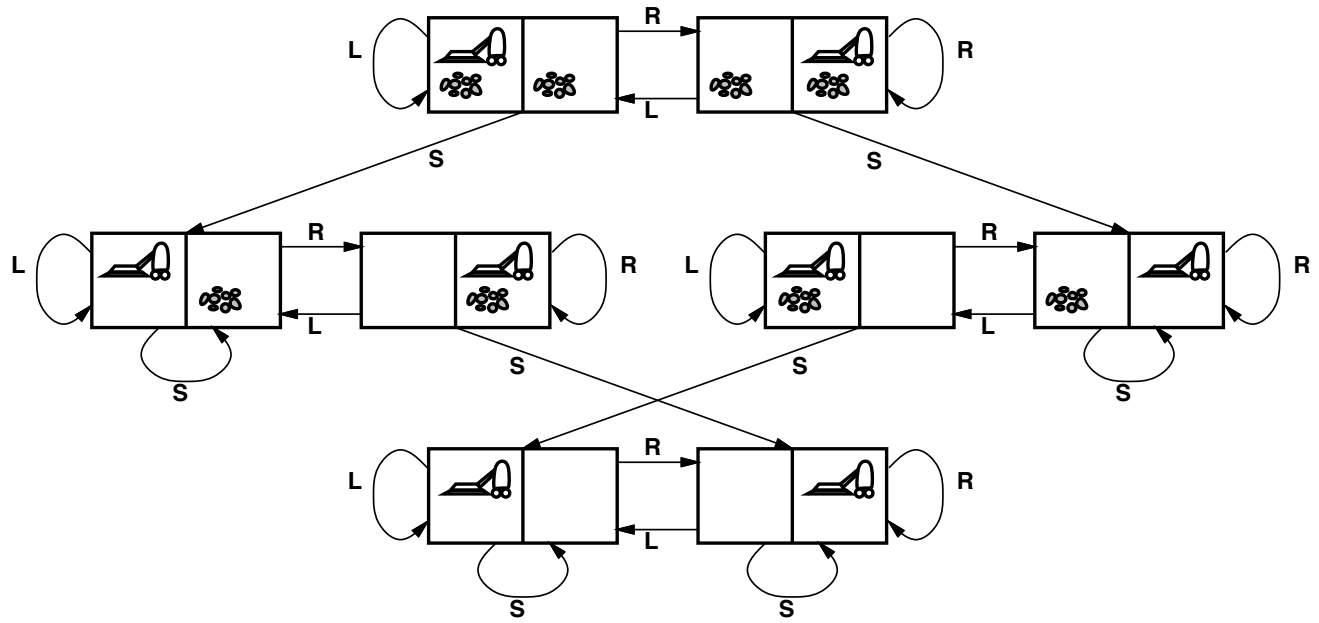
Estados??
Acções??
objectivo??
Custo??

Exemplo: grafo do espaço de estados do mundo do aspirador



Estados??: sujidade (inteiro) posição do aspirador (ignora-se a quantidade de sujidade, etc.)
Acções??
objectivo??
Custo??

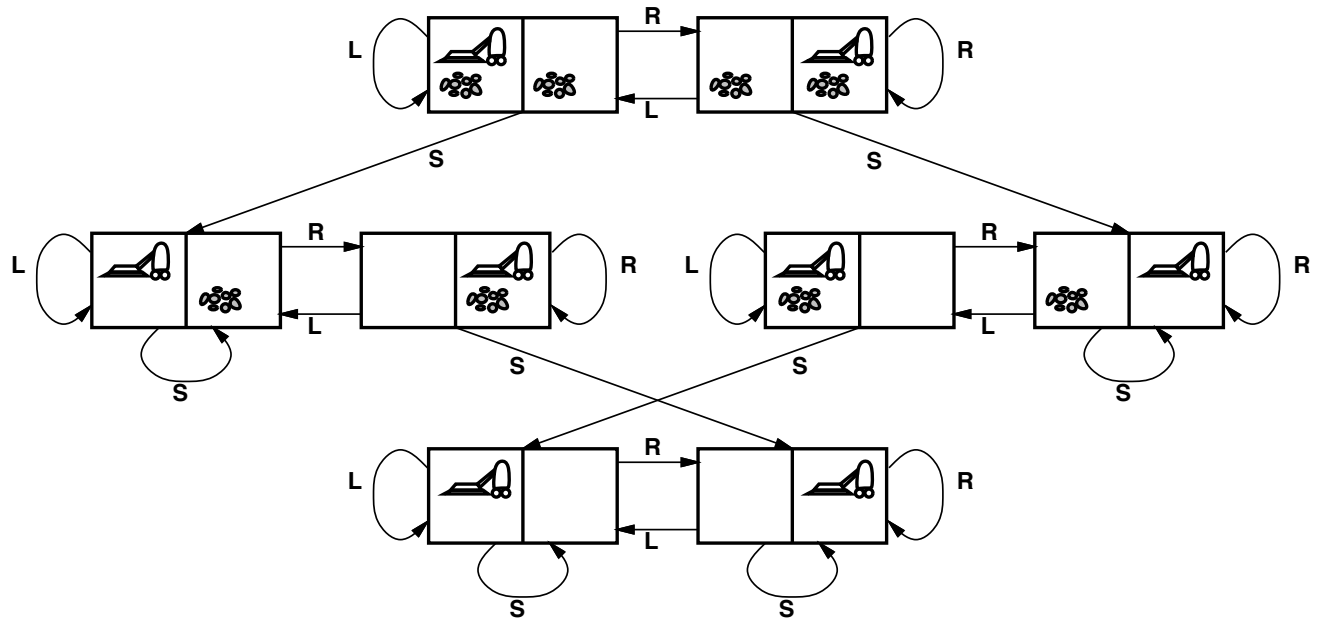
Exemplo: grafo do espaço de estados do mundo do aspirador



Estados??: sujidade (inteiro) posição do aspirador (ignora-se a quantidade de sujidade, etc.)
Acções??$Esquerda$, $Direita$, $Aspira$, $Nada$

objectivo??
Custo??

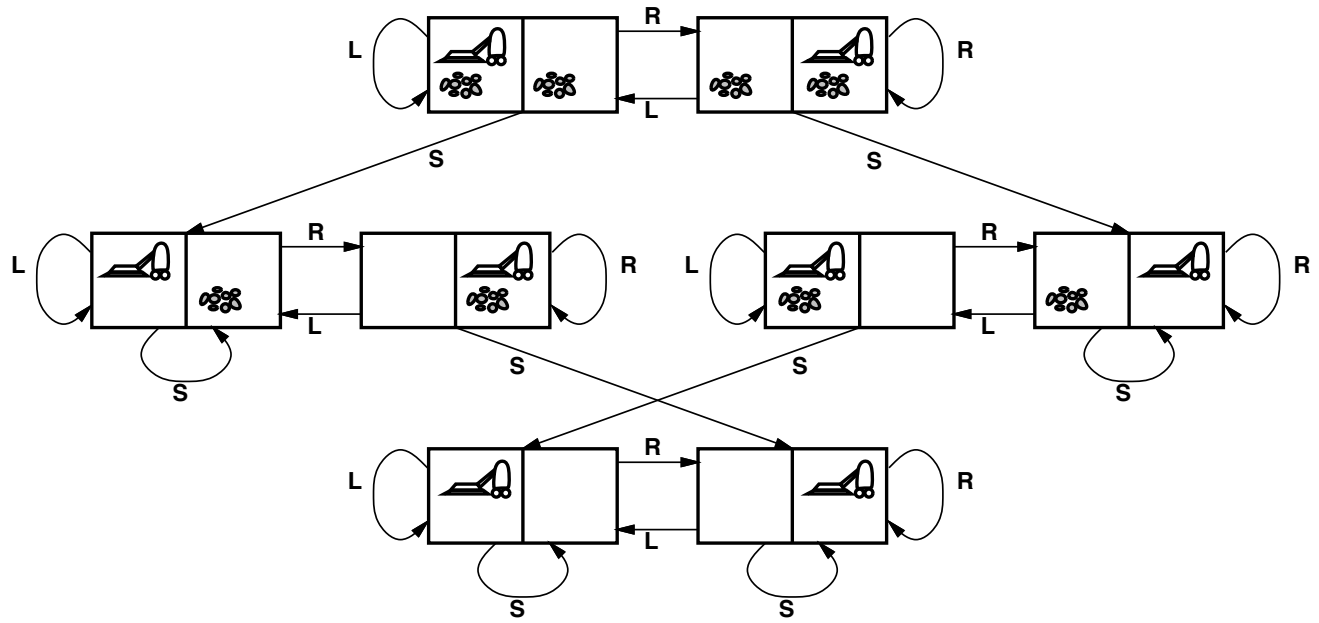
Exemplo: grafo do espaço de estados do mundo do aspirador



Estados??: sujidade (inteiro) posição do aspirador (ignora-se a quantidade de sujidade, etc.)
Acções??$Esquerda$, $Direita$, $Aspira$, $Nada$

objectivo??: limpo
Custo??

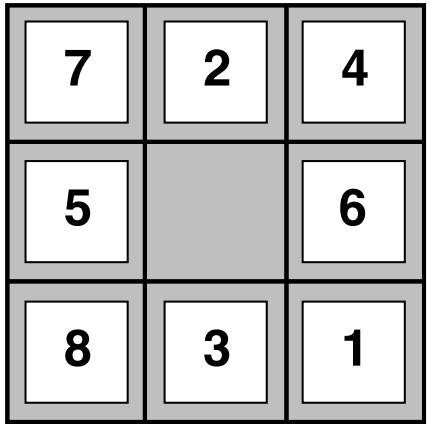
Exemplo: grafo do espaço de estados do mundo do aspirador



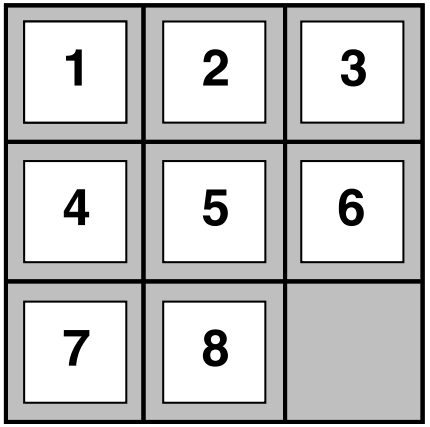
Estados??: sujidade (inteiro) posição do aspirador (ignora-se a quantidade de sujidade, etc.)
Acções??$Esquerda$, $Direita$, $Aspira$, $Nada$

objectivo??: limpo
Custo?? 1 por cada acção (0 para $Nada$)

Exemplo: O puzzle de 8 peças



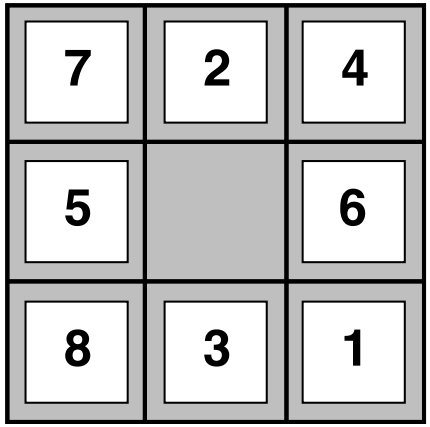
Start State



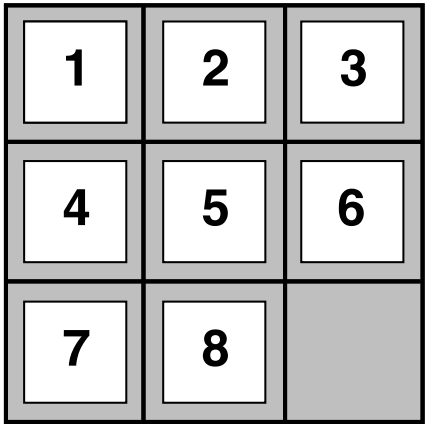
Goal State

Estados??
Acções??
objectivo??
Custo??

Exemplo: O puzzle de 8 peças



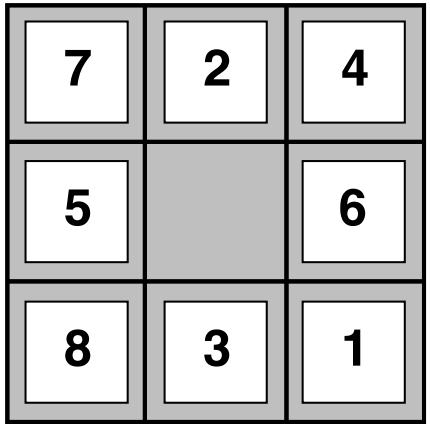
Start State



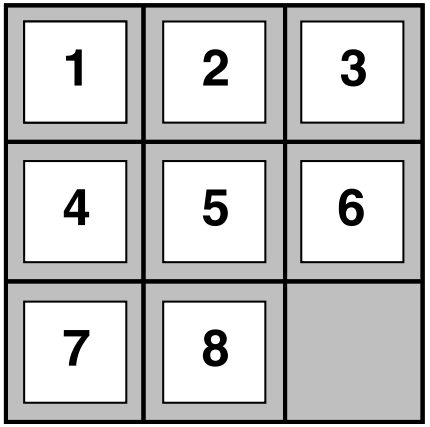
Goal State

Estados??: as posições das peças (inteiros)
Acções??
objectivo??
Custo??

Exemplo: O puzzle de 8 peças



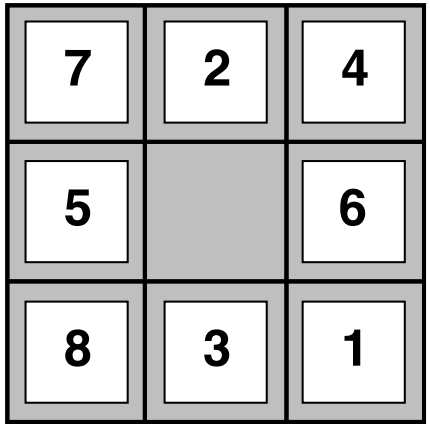
Start State



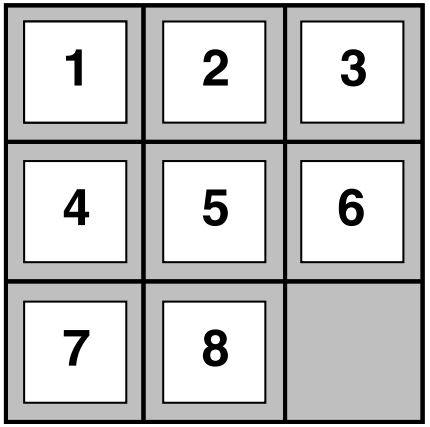
Goal State

Estados??: as posições das peças (inteiros)
Acções??: move o branco esquerda, direita, cima, baixo
objectivo??
Custo??

Exemplo: O puzzle de 8 peças



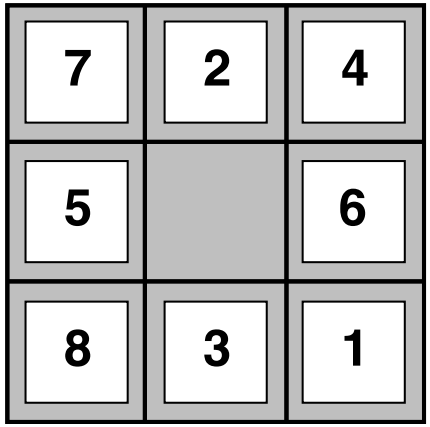
Start State



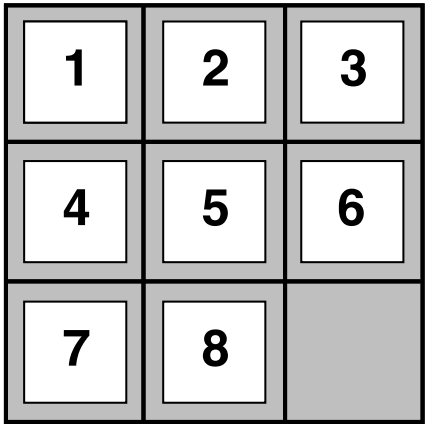
Goal State

Estados??: as posições das peças (inteiros)
Acções??: move o branco esquerda, direita, cima, baixo
objectivo??: um configuração dada
Custo??

Exemplo: O puzzle de 8 peças



Start State

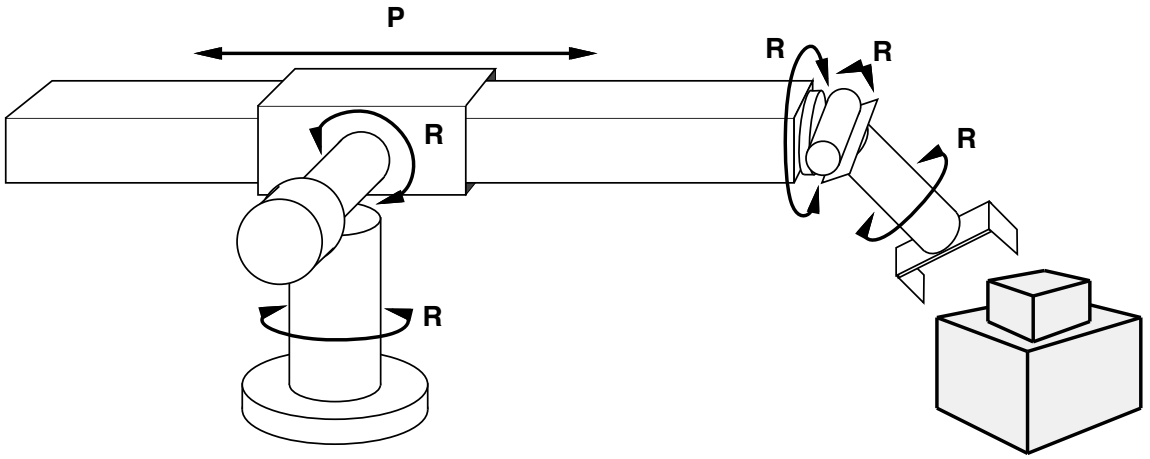


Goal State

Estados??: as posições das peças (inteiros)
Acções??: move o branco esquerda, direita, cima, baixo
objectivo??: um configuração dada
Custo??: 1 por cada movimento

[Nota: a solução óptima da família de n-Puzzle s é NP-hard]

Exemplo: montagem com robots



Estados??: valor real das coordenadas dos ângulos das junções do robot e das partes do objecto que vai ser montado

Acções??: movimento continuo das junções do robot

Objectivo??: montagem completa **sem incluir o robot**

Custo??:tempo de execução

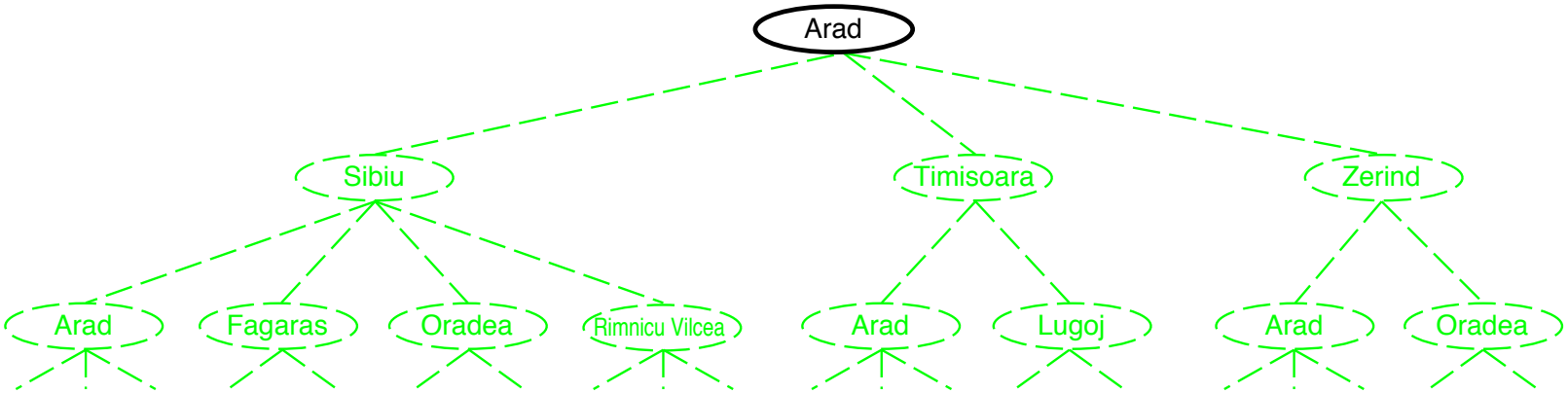
Algoritmos de pesquisa em árvores

Ideia base:

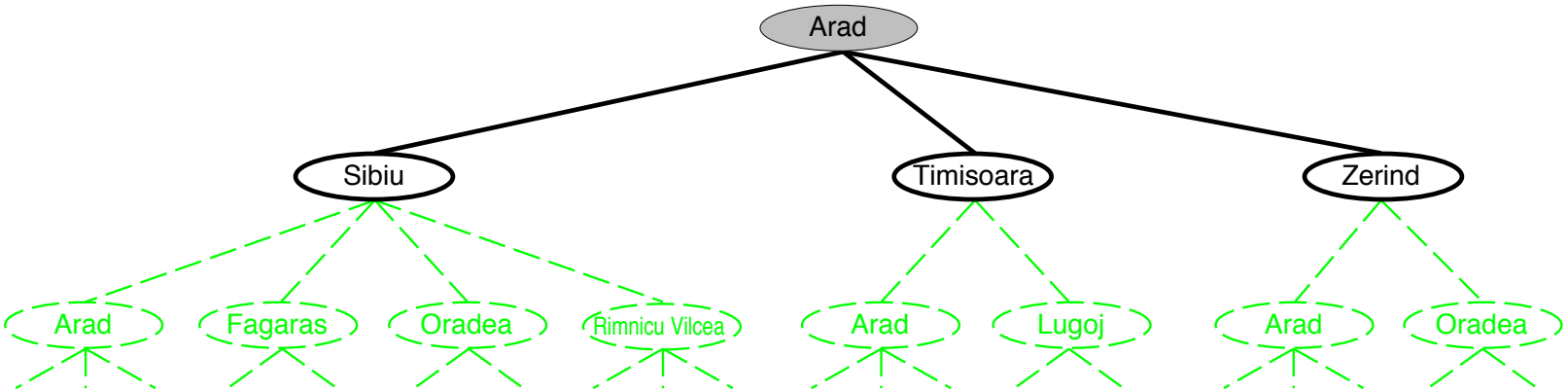
offline, simular a exploração do espaço de estado
gerando sucessores dos estados já explorados
(a.k.a. expandir os estados)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

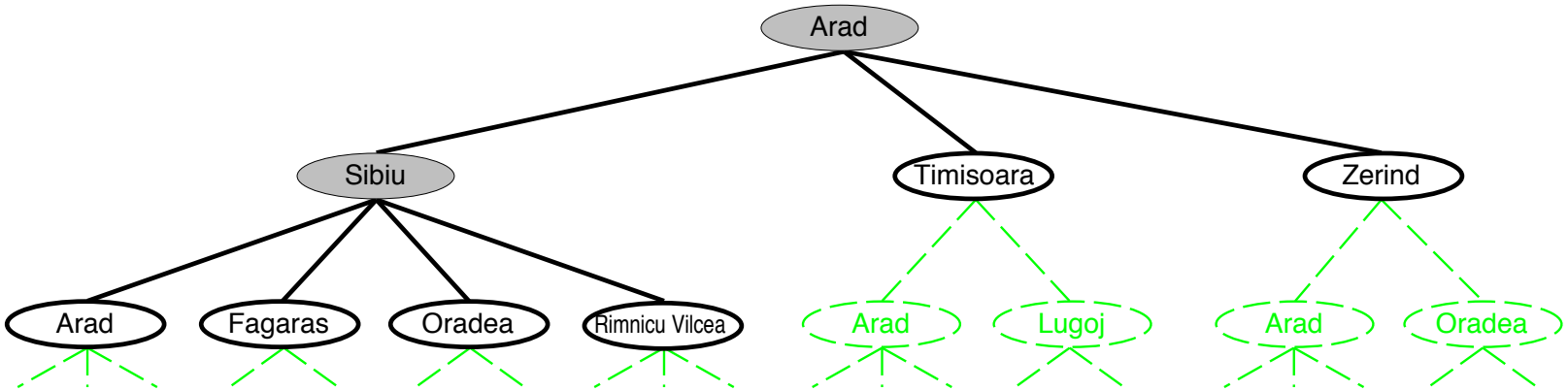
Exemplo de pesquisa numa árvore



Exemplo de pesquisa numa árvore



Exemplo de pesquisa numa árvore



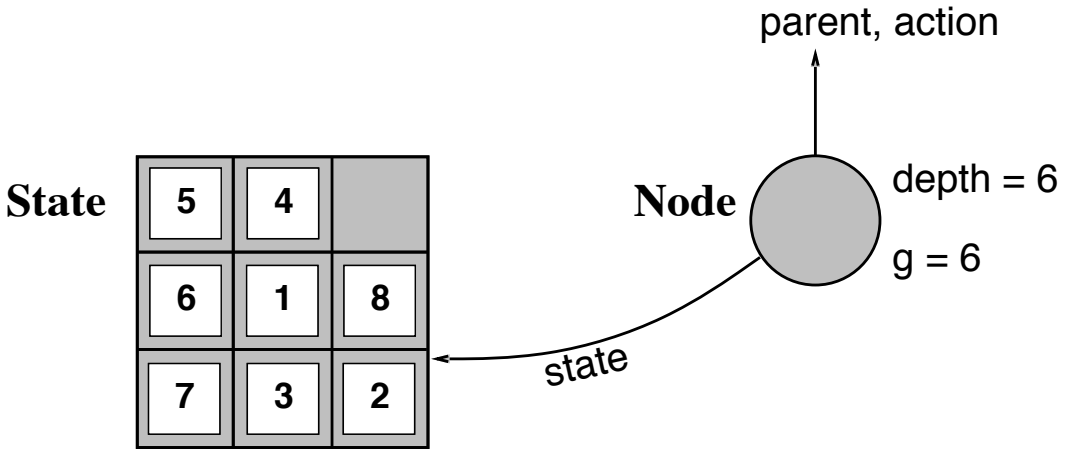
Implementação: estados vs. nós

Um estado é a representação de um configuração física

Um nó é uma estrutura de dados que faz parte da árvore de pesquisa

incluí pai, filhos, profundidade, custo $g(x)$

Os estados não têm pais, filhos, profundidade, ou custo!



A função EXPANDE cria novos nós, preenchendo os vários campos e usa SUCESSORFN do problema para criar os novos estados.

Implementação: pesquisa em árvore geral

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action,
result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Estratégias de pesquisa

Define-se um estratégia escolhendo a **ordem de expansão dos nós**

As estratégia são avaliadas nas seguintes dimensões.

completude—encontra sempre uma solução se existir?
complexidade temporal—numero de nós gerados/expandidos
complexidade espacial—numero máximo de nós em memória
optimalidade—encontra sempre a solução de menor custo?

A complexidade temporal e espacial são medidas em termos de

b— máximo factor de ramificação (branching factor) da árvore de pesquisa

d—profundidade da solução de menor custo
m—profundidade máxima do espaço de estados (pode ser ∞)

Estratégias de pesquisa não informada

Estratégias não informadas só usam a informação disponível na definição do problema

Pesquisa em largura (Breadth-first search)

Pesquisa de custo uniforme (Uniform-cost search)

Pesquisa em Profundidade (Depth-first search)

Pesquisa em profundidade limitada (Depth-limited search)

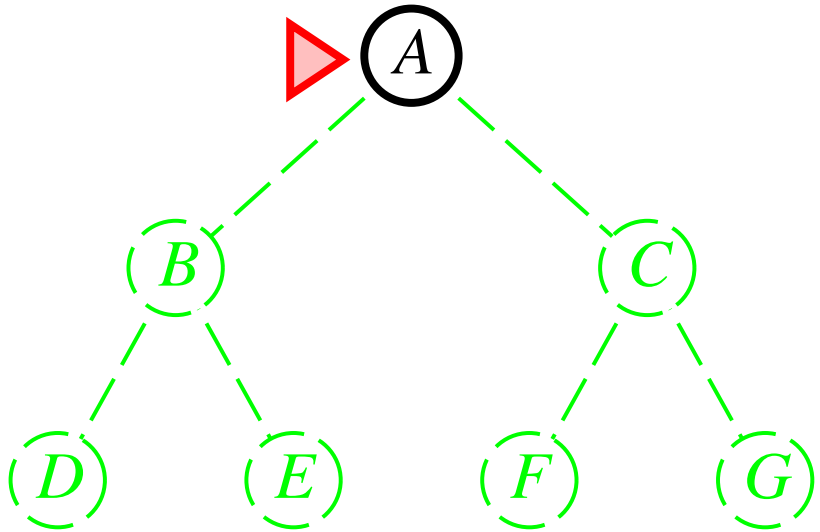
Pesquisa em Profundidade iterativa (Iterative deepening search)

Pesquisa em Largura (Breadth-first search)

Expandir o nó menos profundo que ainda não foi expandido

Implementação:

fringe é uma fila FIFO, i.e., os novos sucessores vão para o fim

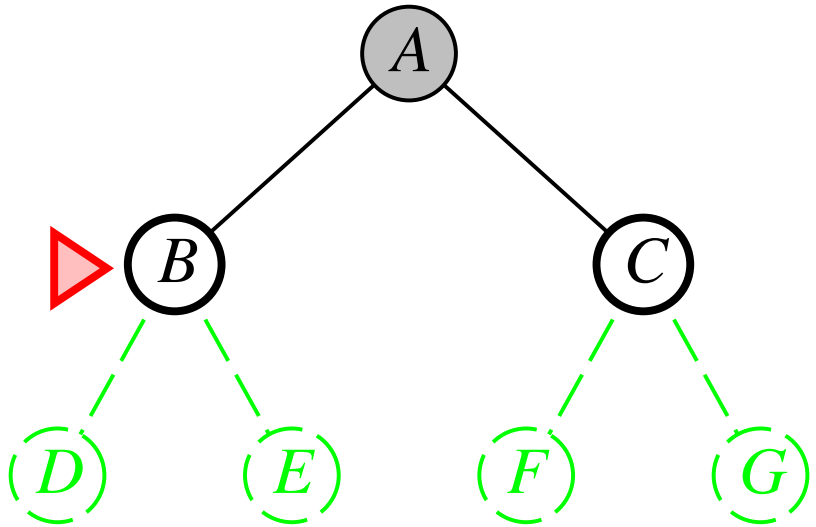


Pesquisa em Largura (Breadth-first search)

Expandir o nó menos profundo que ainda não foi expandido

Implementação:

fringe é uma fila FIFO, i.e., os novos sucessores vão para o fim

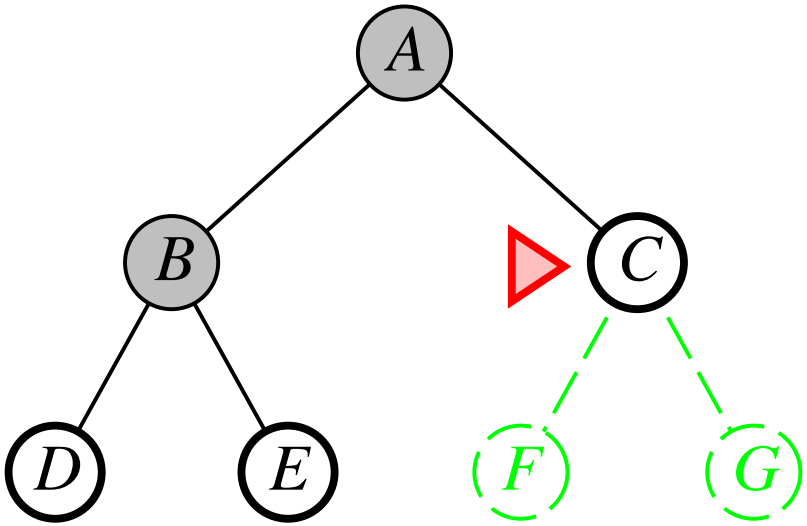


Pesquisa em Largura (Breadth-first search)

Expandir o nó menos profundo que ainda não foi expandido

Implementação:

fringe é uma fila FIFO, i.e., os novos sucessores vão para o fim

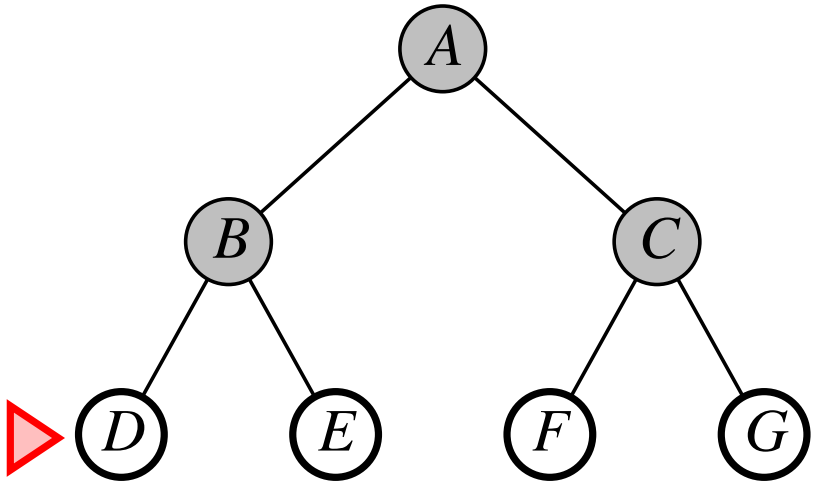


Pesquisa em Largura (Breadth-first search)

Expandir o nó menos profundo que ainda não foi expandido

Implementação:

fringe é uma fila FIFO, i.e., os novos sucessores vão para o fim



Propriedades da pesquisa em largura (breadth-first search)

Completa??

Propriedades da pesquisa em largura (breadth-first search)

Completa?? Sim, se e só se b é finito

Tempo??

Propriedades da pesquisa em largura (breadth-first search)

Completa?? Sim, se e só se b é finito

Tempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. em d

Espaço??

Propriedades da pesquisa em largura (breadth-first search)

Completa?? Sim, se e só se b é finito

Tempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. em d

Espaço?? $O(b^{d+1})$ (Guarda todos os nós em memória)

Óptima??

Propriedades da pesquisa em largura (breadth-first search)

Completa?? Sim, se e só se b é finito

Tempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. em d

Espaço?? $O(b^{d+1})$ (Guarda todos os nós em memória)

Óptima?? Sim, se o custo $= 1$ para cada passo, não é óptima em geral.

Espaço é o grande problema; pode gerar nós a 100MB/sec
em 24hrs = 8640GB.

Pesquisa de custo uniforme (Uniform-cost search)

Expande o nó de menor custo que ainda não foi expandido

Implementação:

$fringe$ = fila ordenada pelo custo, o menor primeiro

Equivalente à pesquisa em largura se os custos dos passos forem todos iguais.

Completa?? Sim, se o custo do passo $\geq \epsilon$

Tempo?? # de nós com$g \leq$ custo da solução óptima, $O(b^{\lceil C^*/\epsilon \rceil})$
onde C^* é o custo da solução óptima

Espaço?? # de nós com $g \leq$ custo da solução óptima, $O(b^{\lceil C^*/\epsilon \rceil})$

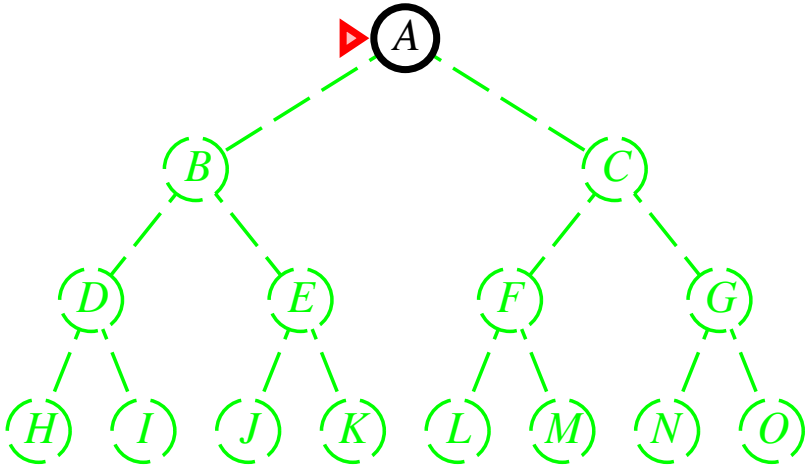
Óptima?? Sim—os nós são expandidos por ordem crescente de $g(n)$ (custo)

Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

$fringe$ = fila LIFO, i.e., colocar o sucessor na frente da fila



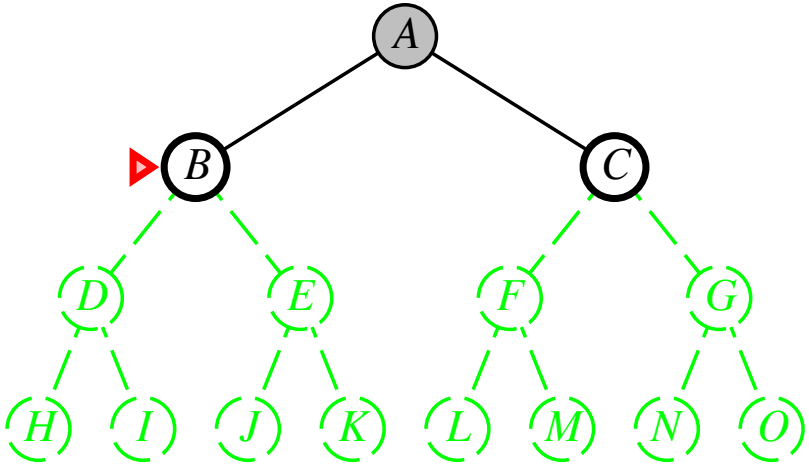
Depth-first search

Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

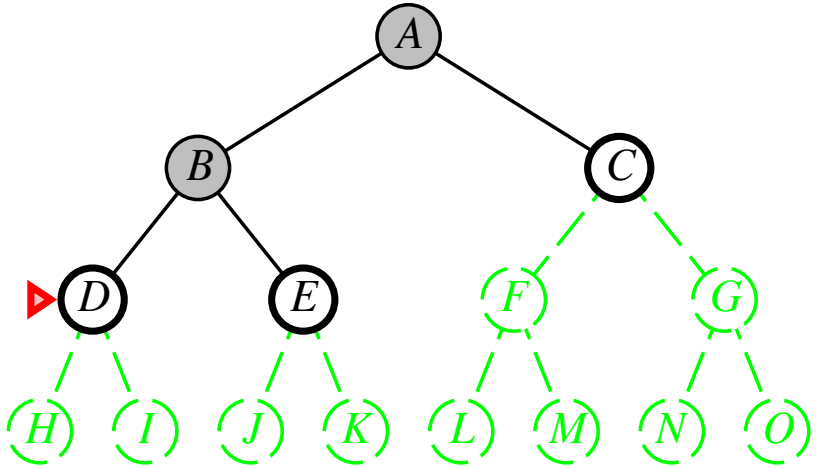


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

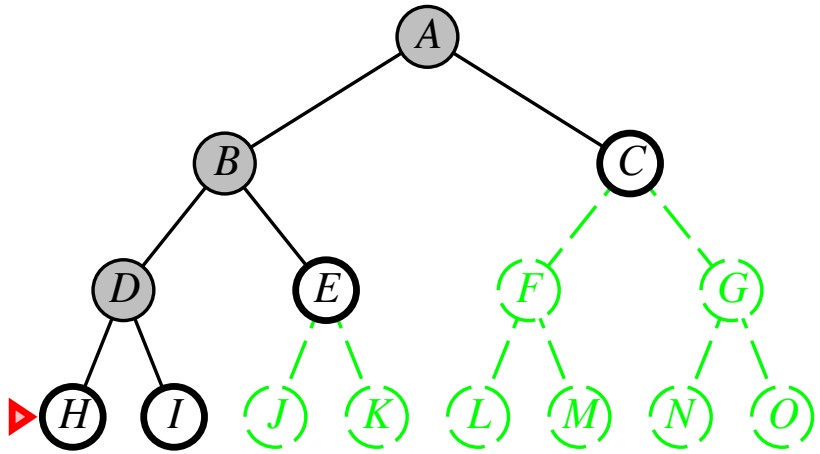


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

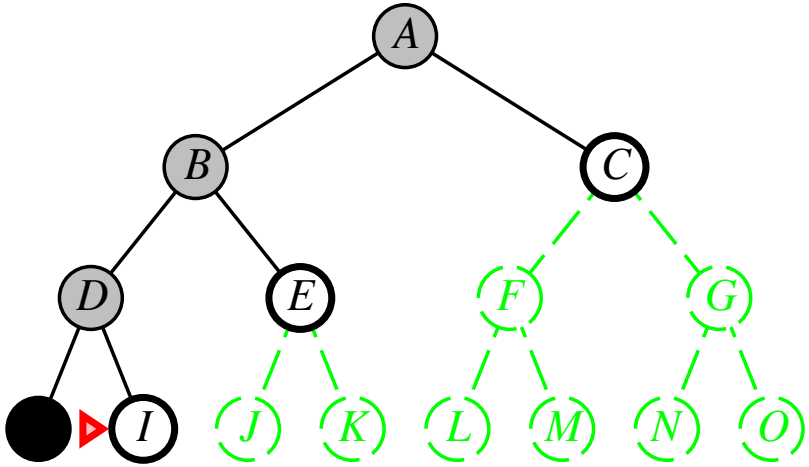


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

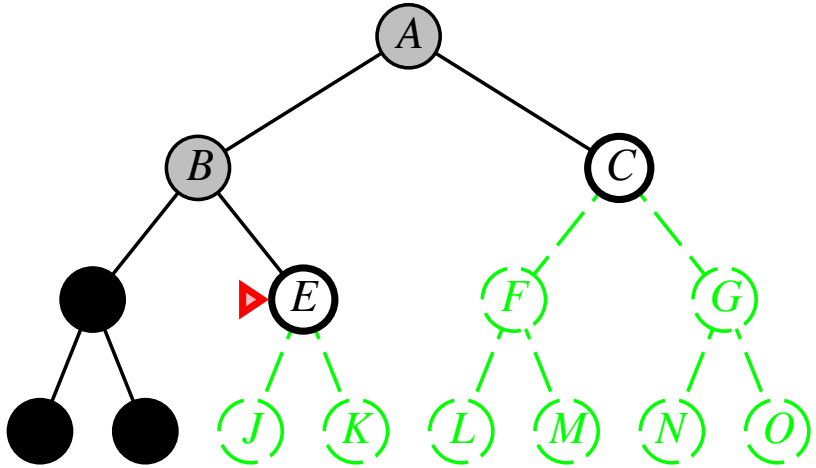


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

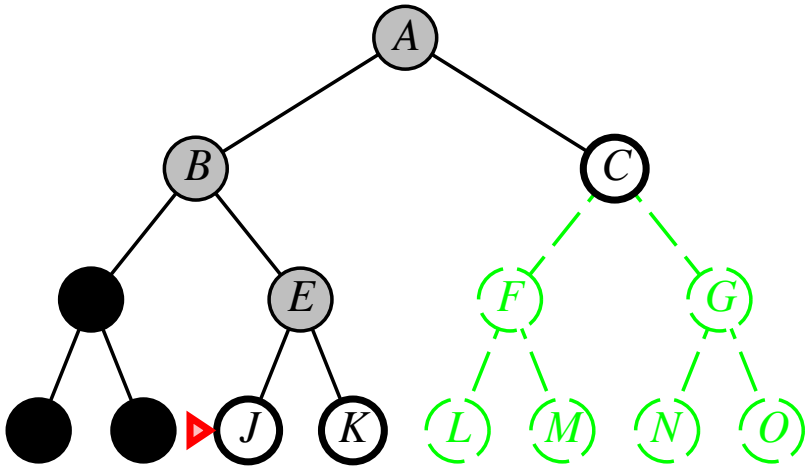


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

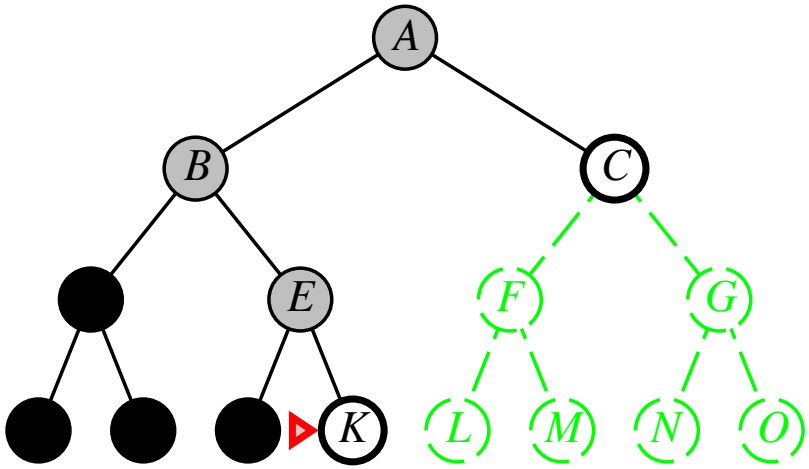


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

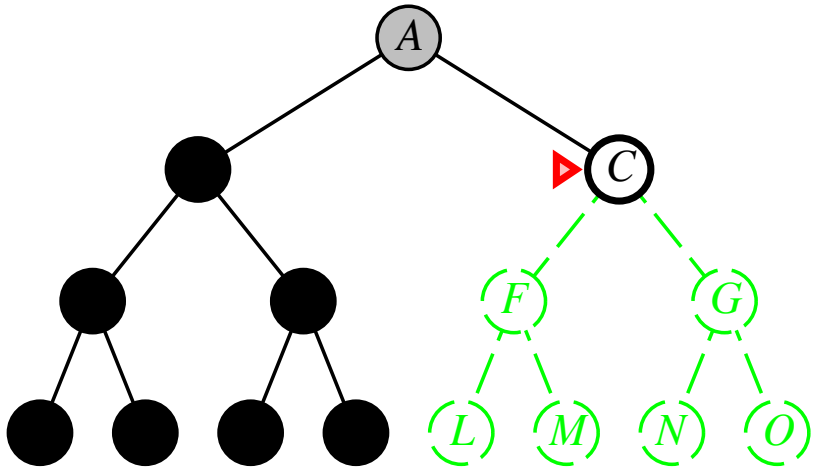


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

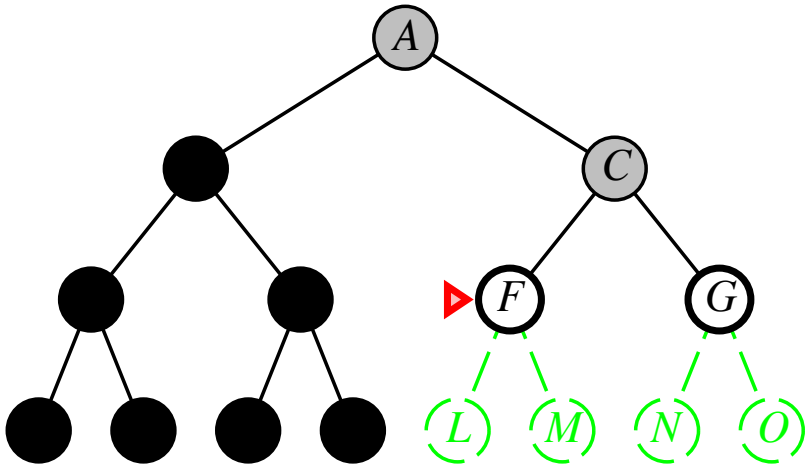


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

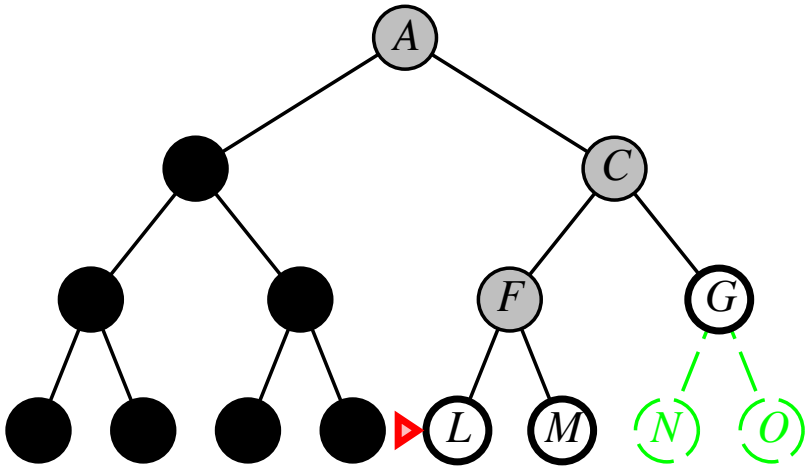


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila

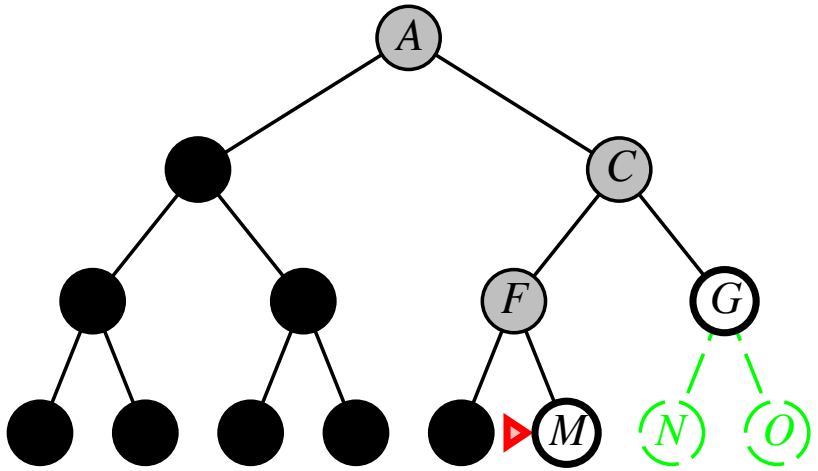


Pesquisa em Profundidade (Depth-first search)

Expandir o nó mais profundo que ainda não foi expandido

Implementação:

fringe = fila LIFO, i.e., colocar o sucessor na frente da fila



Propriedades da pesquisa em profundidade

Completa??

Propriedades da pesquisa em profundidade

Completa?? Não: Falha num espaço de profundidade infinita, espaço com ciclos

Modificar para evitar estados repetidos ao longo de um caminho
$\Rightarrow$ completo em espaços finitos

Tempo??

Propriedades da pesquisa em profundidade

Completa?? Não: Falha num espaço de profundidade infinita, espaço com ciclos

Modificar para evitar estados repetidos ao longo de um caminho
$\Rightarrow$ completo em espaços finitos

Tempo?? $O(b^m)$: terrível se m é muito maior que d
mas se as soluções são densas (há muitas e juntas) pode ser muito mais rápido que a pesquisa em largura.

Espaço??

Propriedades da pesquisa em profundidade

Completa?? Não: Falha num espaço de profundidade infinita, espaço com ciclos

Modificar para evitar estados repetidos ao longo de um caminho
$\Rightarrow$ completo em espaços finitos

Tempo?? $O(b^m)$: terrível se m é muito maior que d
mas se as soluções são densas (há muitas e juntas) pode ser muito mais rápido que a pesquisa em largura.

Espaço?? $O(bm)$, i.e., espaço linear!

Óptimo??

Propriedades da pesquisa em profundidade

Completa?? Não: Falha num espaço de profundidade infinita, espaço com ciclos

Modificar para evitar estados repetidos ao longo de um caminho
$\Rightarrow$ completo em espaços finitos

Tempo?? $O(b^m)$: terrível se m é muito maior que d
mas se as soluções são densas (há muitas e juntas) pode ser muito mais rápido que a pesquisa em largura.

Espaço?? $O(bm)$, i.e., espaço linear!

Óptimo?? Não

Pesquisa em profundidade limitada (Depth-limited search)

= pesquisa em profundidade com profundidade limitada a l,
i.e., nós à profundidade l não têm sucessores.

Implementação recursiva:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Pesquisa em profundidade iterativa (Iterative deepening search)

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Pesquisa em profundidade iterativa $l = 0$

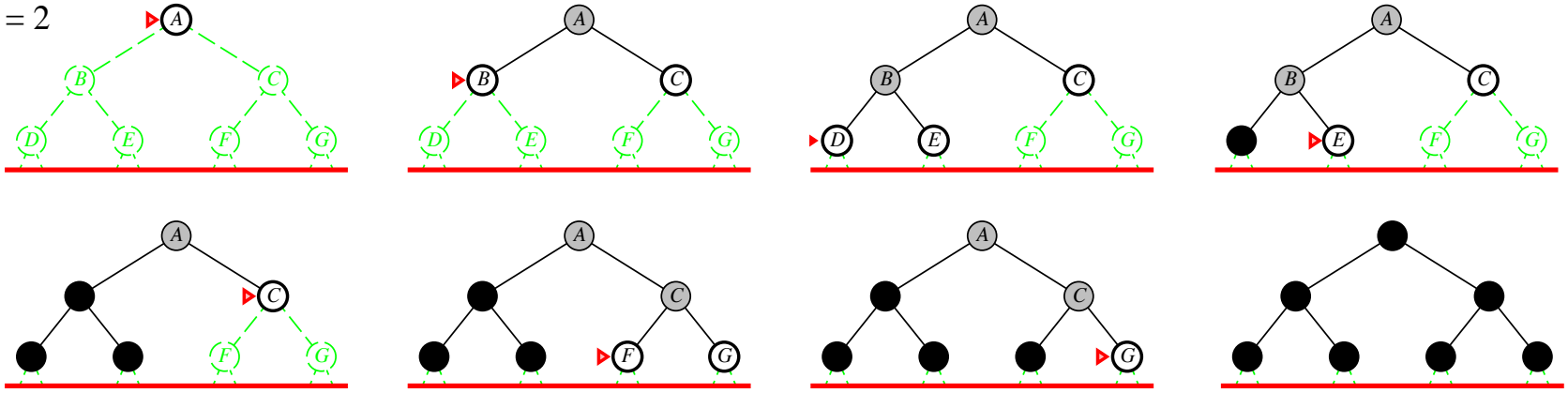
Limit = 0

Pesquisa em profundidade iterativa $l = 1$

Limit = 1

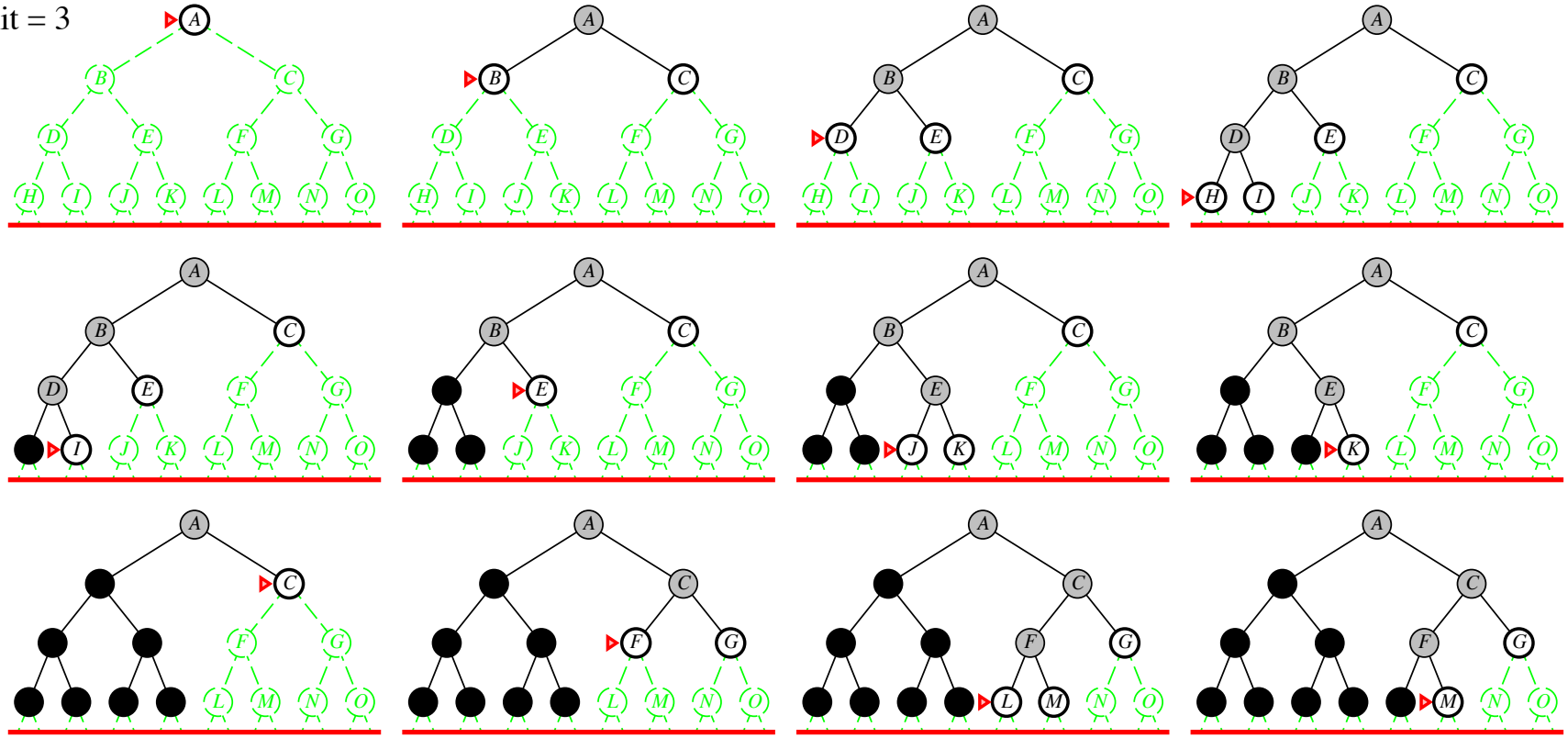
Pesquisa em profundidade iterativa $l = 2$

Limit = 2



Pesquisa em profundidade iterativa $l = 3$

Limit = 3



Propriedades da pesquisa em profundidade iterativa

Completa??

Propriedades da pesquisa em profundidade iterativa

Completa?? Sim

Tempo??

Propriedades da pesquisa em profundidade iterativa

Completa?? Sim

Tempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espaço??

Propriedades da pesquisa em profundidade iterativa

Completa?? Sim

Tempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espaço??$O(bd)$

Óptima??

Propriedades da pesquisa em profundidade iterativa

Completa?? Sim

Tempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espaço??$O(bd)$

Óptima?? Sim, se o custo dos passos é igual a 1

Pode ser modificado para explorar árvores de custo uniforme

Comparação para $b = 10$ e $d = 5$, solução na folha direita mais distante.:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

O desempenho de IDS é melhor porque os outros nós à profundidade d não foram expandidos

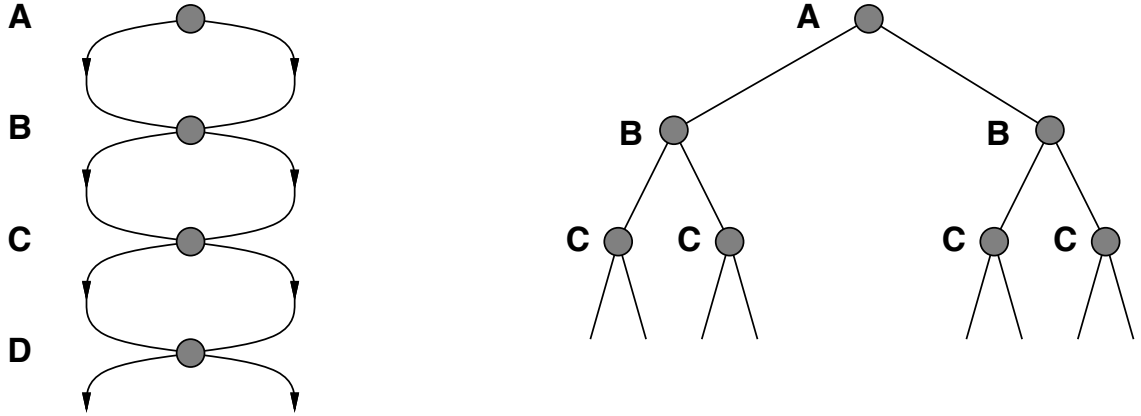
O BFS pode ser modificado para aplicar o teste de estado final quando um nó é **gerado**

Resumo dos algoritmos

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Estados repetidos

A falha na detecção de estados repetidos pode transformar um problema linear num problema exponencial!



Pesquisa em grafo

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Resumo

A formulação de problemas como problemas de pesquisa no espaço de estados requer a abstracção de detalhes do mundo real para definir o espaço de estados de forma a que se possa explorar.

Há várias estratégias de pesquisa não informada:

A pesquisa em profundidade iterativa (Iterative deepening search) usa espaço linear
e não muito mais tempo que outras pesquisas não informadas

A pesquisa em grafo pode ser exponencialmente mais eficiente que a pesquisa em árvore.