Resolução de Problemas através de Busca Cega

Prof. Júlio Cesar Nievola
PPGIa - PUCPR

Solução de Problemas

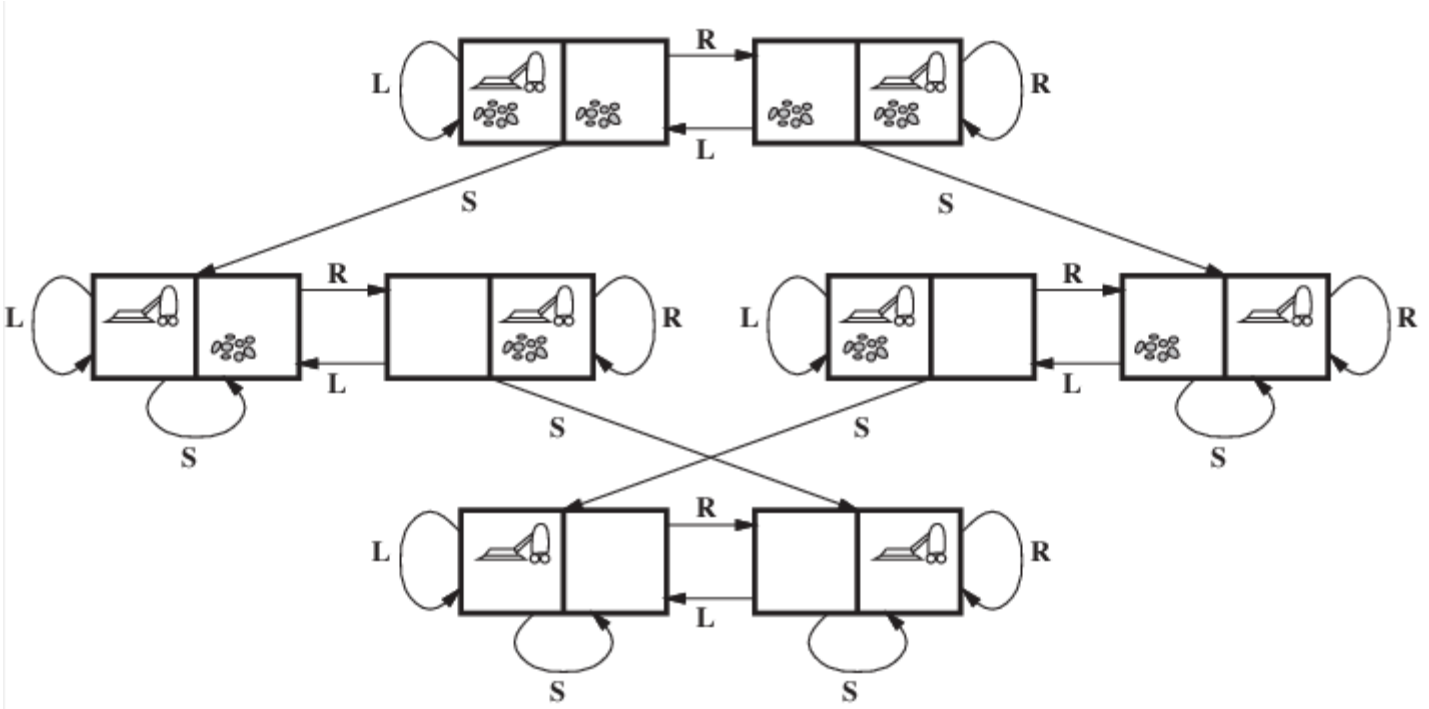
- Sistemas inteligentes devem agir de maneira a fazer com que o ambiente passe por uma sequência de estados que maximize a sua medida de desempenho.
- Exemplo: partir de Arad, na Romênia e chegar a Bucareste.

Elementos de um problema

- Formulação do objetivo
- Formulação do problema
  - Processo de decidir quais ações e estados devem ser considerados
- Busca
- Solução
- Execução

Formulação de problemas

Os 8 estados possíveis do mundo do vácuo simplificado.



Problemas e soluções bem-definidos - 1

- O estado inicial
- Operadores - função sucessor S
  - Conjunto de possíveis ações disponíveis ao sistema
- Espaço de estados
  - É o conjunto de estados alcançáveis a partir do estado inicial por uma série de ações
  - Um caminho no espaço de estados é uma sequência de ações de um estado a outro

Problemas e soluções bem-definidos - 2

- Teste do objetivo
  - É aplicado pelo sistema para determinar se a descrição do estado é um estado objetivo
- Custo do caminho **g**
  - É uma função que assinala um custo a um caminho

Tipos de dados: PROBLEMA

componentes: ESTADO_INICIAL, OPERADORES, TESTE_DO_OBJETIVO, FUNÇÃO_CUSTO_DO_CAMINHO

Medindo o desempenho da solução do problema

- A busca consegue encontrar uma solução para o problema?
- A solução encontrada é boa?
- Qual é o custo da busca em termos de tempo e memória necessários para encontrar a solução?
- Compromisso: solução ótima em tempo elevado ou vice-versa?

Exemplo: Romênia

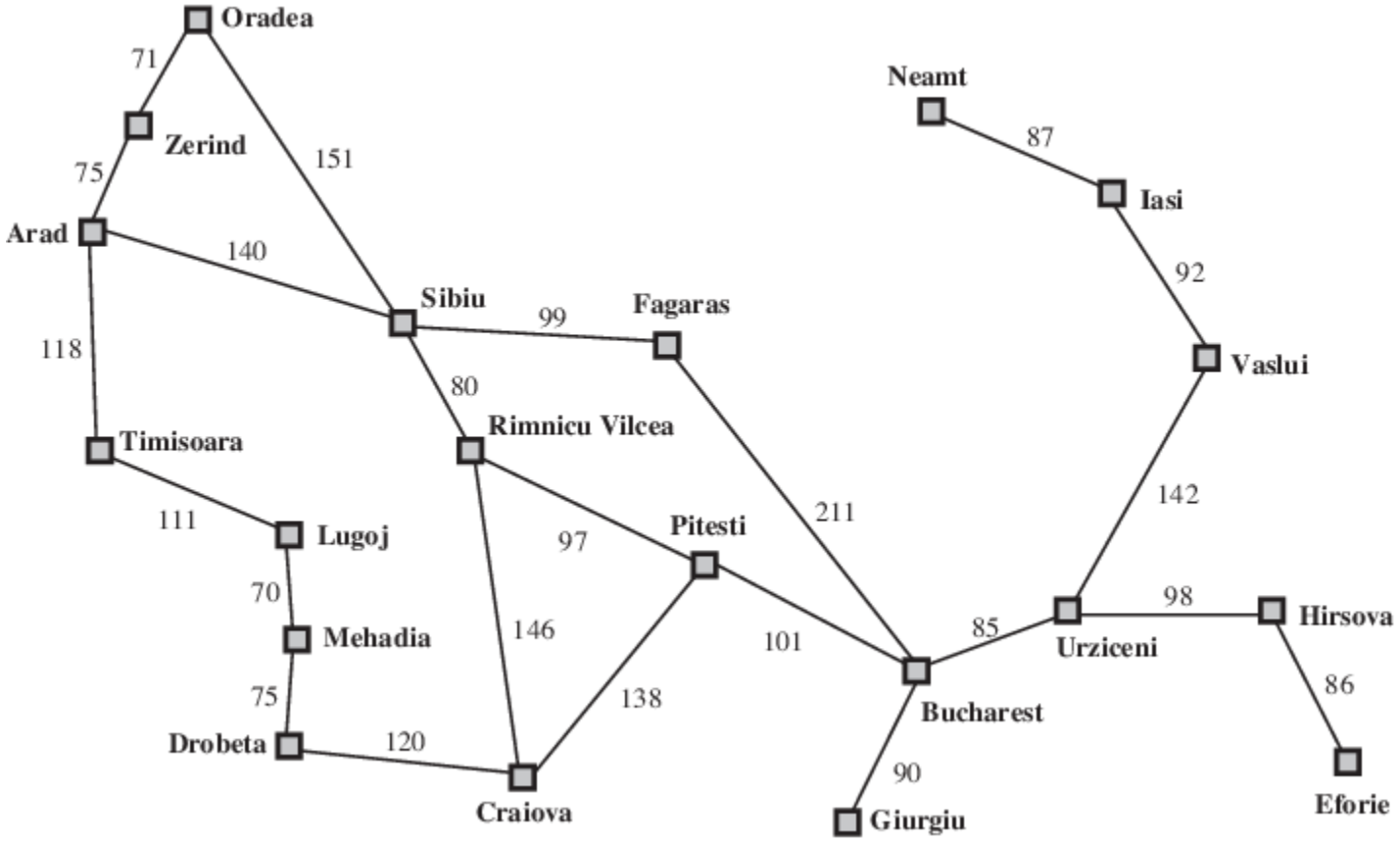
- Em férias na Romênia; atualmente em Arad.
- O voo de volta parte amanhã de Bucharest.

- Formular objetivo:
  - Estar em Bucharest.

- Formular problema:
  - Estados: várias cidades.
  - Ações: dirigir entre cidades.

- Encontrar a solução:
  - Sequência de cidaes, e.g., Arad, Sibiu, Fagaras, Bucharest.

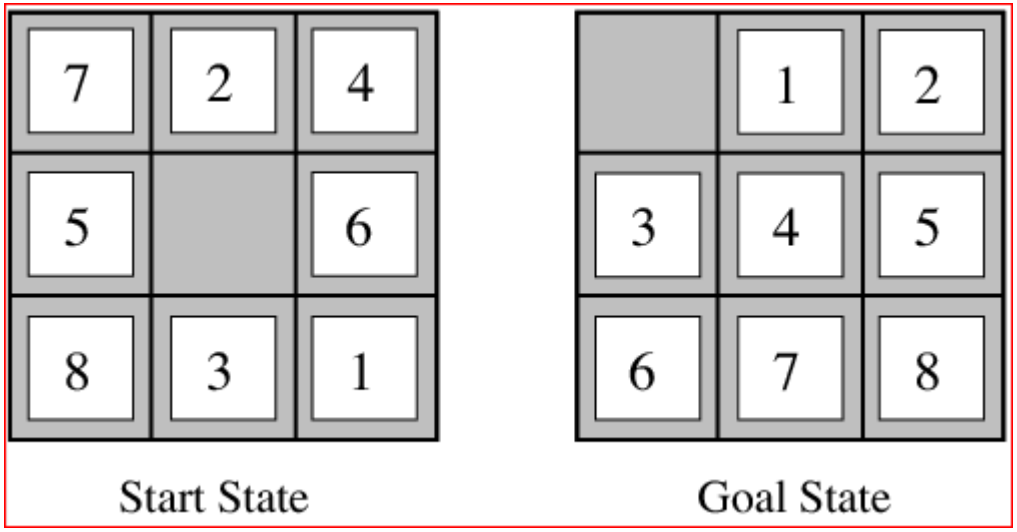
Escolhendo estados e ações: abstração



Problemas-exemplo

- Problemas de brinquedo (“toy problems”)
  - Por ter uma descrição exata e concisa permitem comparar o desempenho de diferentes algoritmos
- Problemas reais (“real-world problems”)
  - Sua solução tem utilidade real, mas a formulação pode privilegiar alguma(s) das propriedades

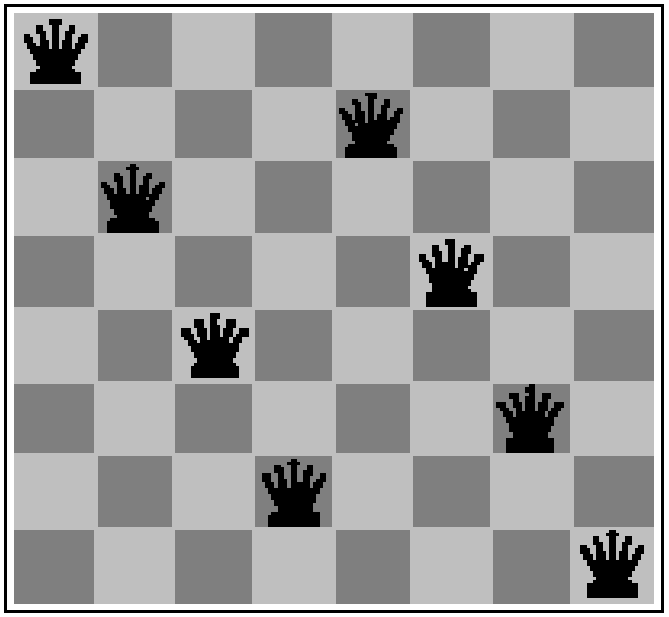
Problema de brinquedo: quebra-cabeças



Quebra-cabeças

- Estados
  - Localização de cada peça, inclusive o branco
- Operadores
  - Branco move à direita, esquerda, cima, baixo
- Teste do objetivo
  - Estado corresponde à configuração desejada
- Custo do caminho
  - Cada passo tem custo 1

Problema de brinquedo: Posicionamento de Rainhas



Posicionamento de Rainhas 1

- Estados
  - Qualquer arranjo de 0 a 8 rainhas no tabuleiro
- Operadores
  - Adicionar uma rainha a qualquer posição
- Teste do objetivo
  - 8 rainhas colocadas, nenhuma atacada
- Custo do caminho
  - Zero

Posicionamento de Rainhas 2

- Estados
  - Arranjo de 0 a 8 rainhas com nenhuma atacada
- Operadores
  - Colocar uma rainha na coluna mais à esquerda vazia, tal que não seja atacada
- Teste do objetivo
  - 8 damas colocadas, nenhuma atacada
- Custo do caminho
  - Zero

Posicionamento de Rainhas 3

- Estados
  - Arranjo de 8 rainhas, uma em cada coluna
- Operadores
  - Mover qualquer rainha atacada para outra posição na mesma coluna
- Teste do objetivo
  - 8 damas colocadas, nenhuma atacada
- Custo do caminho
  - Zero

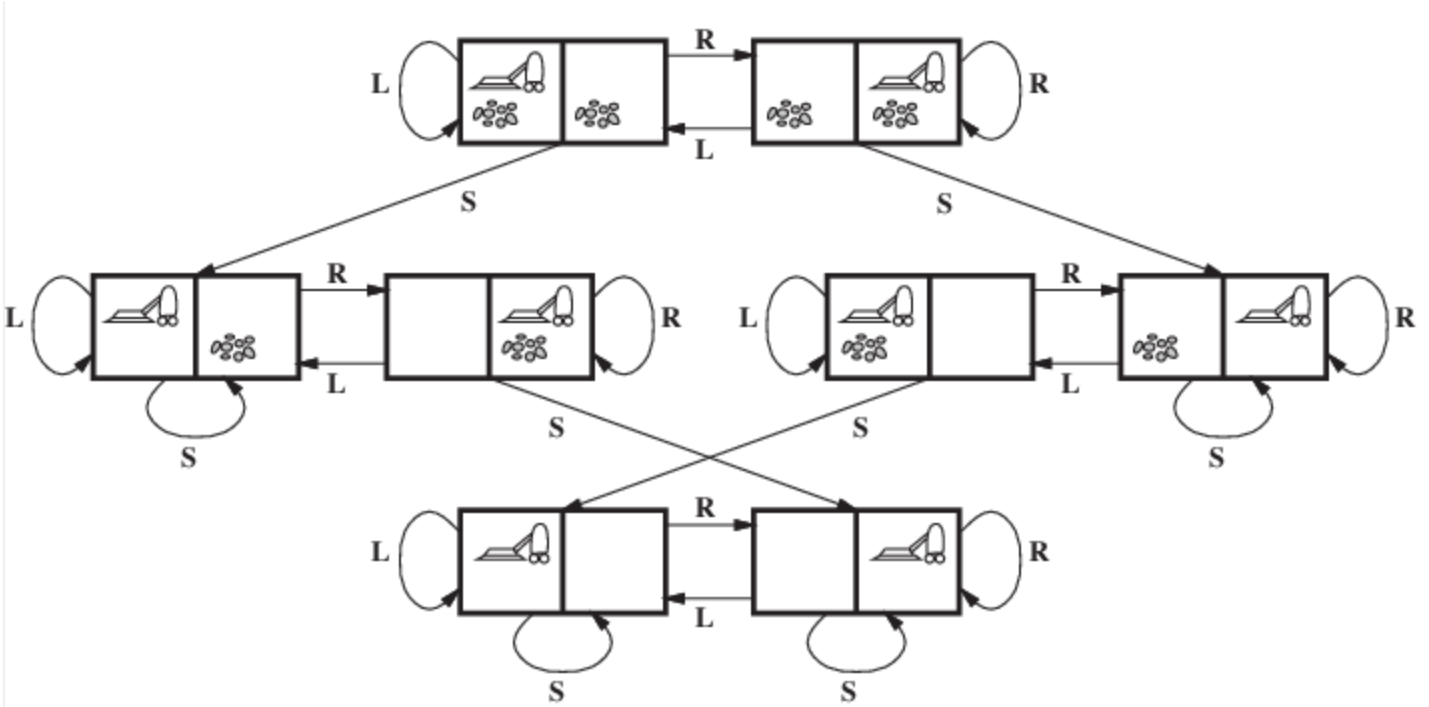
Problema de brinquedo: Cripto-aritmética

```
    FORTY          29786
+     TEN      +     850
+     TEN            850
 ---------      -------
   SIXTY          31486
```

Cripto-aritmética

- Estados
  - Um jogo com algumas letras substituídas por dígitos
- Operadores
  - Substituir todas as ocorrências de uma letra por um dígito novo
- Teste do objetivo
  - Jogo contém somente dígitos e a soma é correta
- Custo do caminho
  - Zero, todas as soluções tem a mesma validade

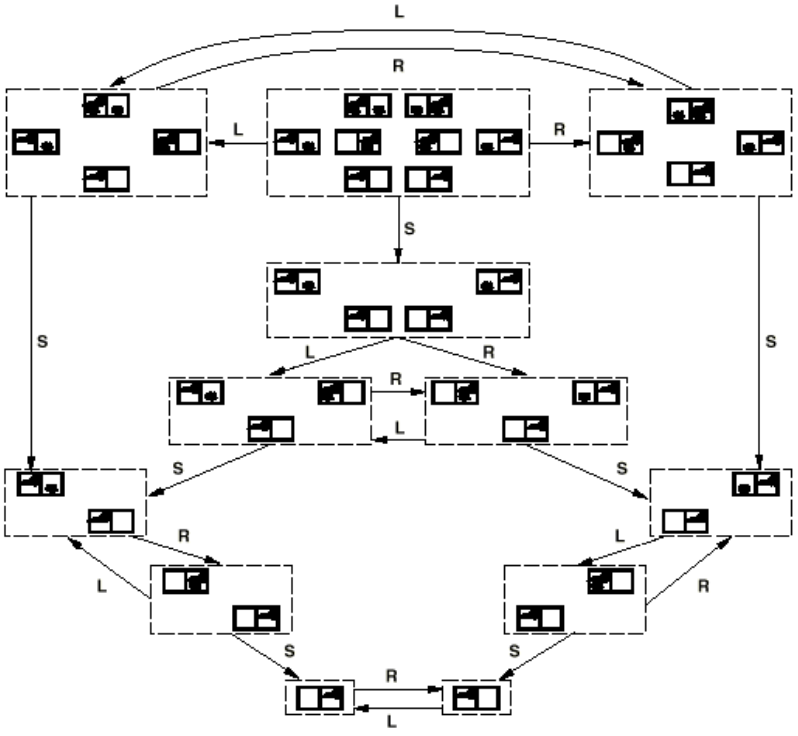
Problema de brinquedo: Mundo do vácuo



Mundo do vácuo

- Estados
  - Um dos oito estados mostrados
- Operadores
  - Mover à direita, à esquerda, sugar
- Teste do objetivo
  - Não haver sujeira em qualquer quadrado
- Custo do caminho
  - Cada ação custa 1

Problema de brinquedo: Mundo do vácuo com Murphy



Mundo do vácuo com Murphy

- Estados
  - Subconjuntos dos estados 1-8
- Operadores
  - Mover à direita, à esquerda, sugar
- Teste do objetivo
  - Todos os estados no conjunto não tem sujeira
- Custo do caminho
  - Cada ação custa 1

Problema de brinquedo: Missionários e canibais

- Estados
  - Sequência ordenada de 3 números, indicando número de missionários, canibais e barcos na margem direita
- Operadores
  - Levar um missionário ou um canibal, dois missionários ou canibais ou um de cada
- Teste do objetivo
  - Estado (0,0,0)
- Custo do caminho
  - Número de cruzamentos

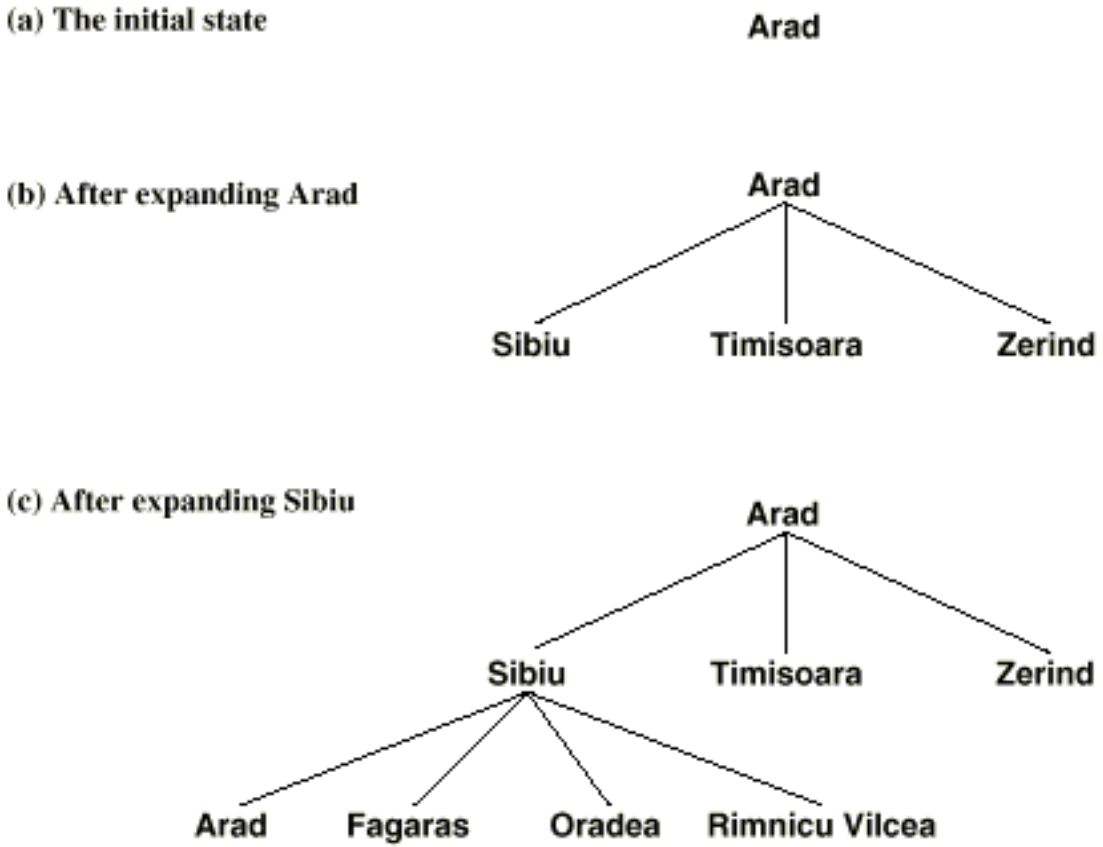
Problemas reais

- Planejamento de rotas aéreas
- Problema do caixeiro viajante
- Leiaute VLSI
- Navegação de robôs
- Seqüência de montagem

Gerando sequências de ações

- Na busca da solução deve-se buscar estados
- Para isto, aplicam-se os operadores ao estado atual, gerando novos estados
- Isto é chamado expansão dos estados
- A escolha do próximo estado a ser expandido é feito pela estratégia de busca
- No processo de busca constrói-se uma árvore de busca

Árvore parcial para busca da rota entre Arad e Bucareste



Descrição informal do algoritmo geral de busca

function BUSCA_GERAL(*problema, estratégia*) **return** uma solução ou falha

inicializar a árvore de busca usando o estado inicial do *problema*

loop do

if não há candidatos a expansão **then return** falha

escolher um nó folha para expansão segundo a *estratégia*

if o nó contém um estado objetivo **then return** a solução correspondente

else expandir o nó e adicionar os nós resultantes à árvore de busca

end

Estruturas de dados para árvores de busca 1

- Pode-se considerar que um nó é uma estrutura de dados com 5 componentes:
  - o estado representado (no espaço de estados)
  - o nó pai (que gerou o nó em questão)
  - o operador aplicado na geração do nó
  - a profundidade do nó
  - o custo do caminho a partir do estado inicial

Tipo de dados: nó

componentes: ESTADO, NÓ_PAI, OPERADOR, PROFUNDIDADE, FUNÇÃO_CUSTO

Estruturas de dados para árvores de busca 2

- Assume-se que a coleção de nós seja implementada como uma fila:
  - CRIAR_FILA(Elementos) cria uma fila com os elementos dados
  - VAZIA?(Fila) retorna verdadeiro somente se não há elementos na fila
  - REMOVE-PRIMEIRO(Fila) remove o elemento na frente da fila e retorna a mesma
  - ENFILEIRAR-FN(Elementos,Fila) insere um conjunto de elementos na fila

Algoritmo geral de busca

function BUSCA_GERAL(*problema,* ENFILEIRAR-FN) **return** uma solução, ou falha

nós → CRIAR-FILA(CRIAR-NÓ(ESTADO-INICIAL[*problema*]))

loop do

if *nós* está vazio **then return** falha

nó → REMOVE-PRIMEIRO(*nós*)

if TEST-OBJETIVO[*problema*] aplicado a STATE(nó) tem sucesso
then return *nó*

nós → ENFILEIRAR-FN (*nós*, EXPANDIR (*nó*, OPERADORES[*problema*]))

end

Estratégias de busca 1

- Busca não-informada ou busca cega
  - Não tem informações sobre o número de passos ou custo do caminho do estado atual ao objetivo
- Busca informada ou busca heurística
  - Dispõe de informações que auxiliam a determinar o provável melhor caminho entre o estado atual e o estado objetivo

Estratégias de busca 2

- Completude
  - A estratégia encontrará a solução se ela existir?
- Otimalidade
  - A estratégia encontra a melhor solução quando existem várias?
- Complexidade temporal
  - Quanto demora para encontrar a solução?
- Complexidade de espaço
  - Quanta memória é necessária para a busca?

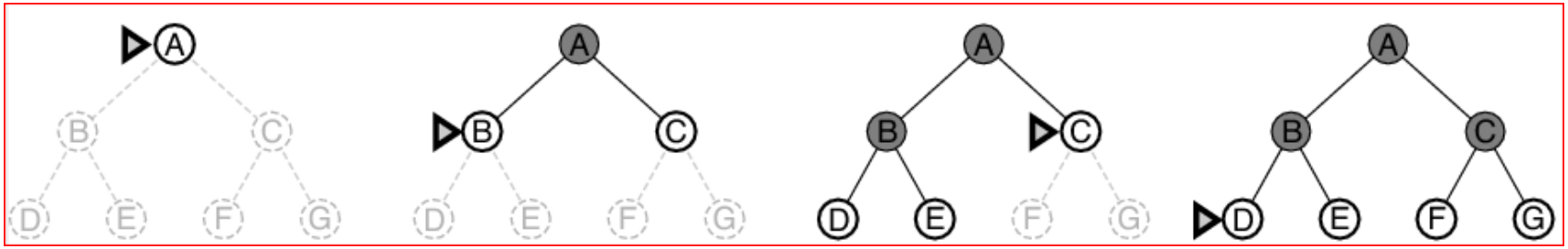
Busca primeiro em largura 1

- Todos os nós na profundidade *d* da árvore de busca são expandidos antes dos nós em profundidade *d+1*

function BUSCA-PRIMEIRO-LARGURA(*problema*)
return uma solução ou falha

BUSCA-GERAL (*problema*, ENFILEIRAR-NO-FIM)

Busca primeiro em largura 2



Busca primeiro em largura 3

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Busca em Largura

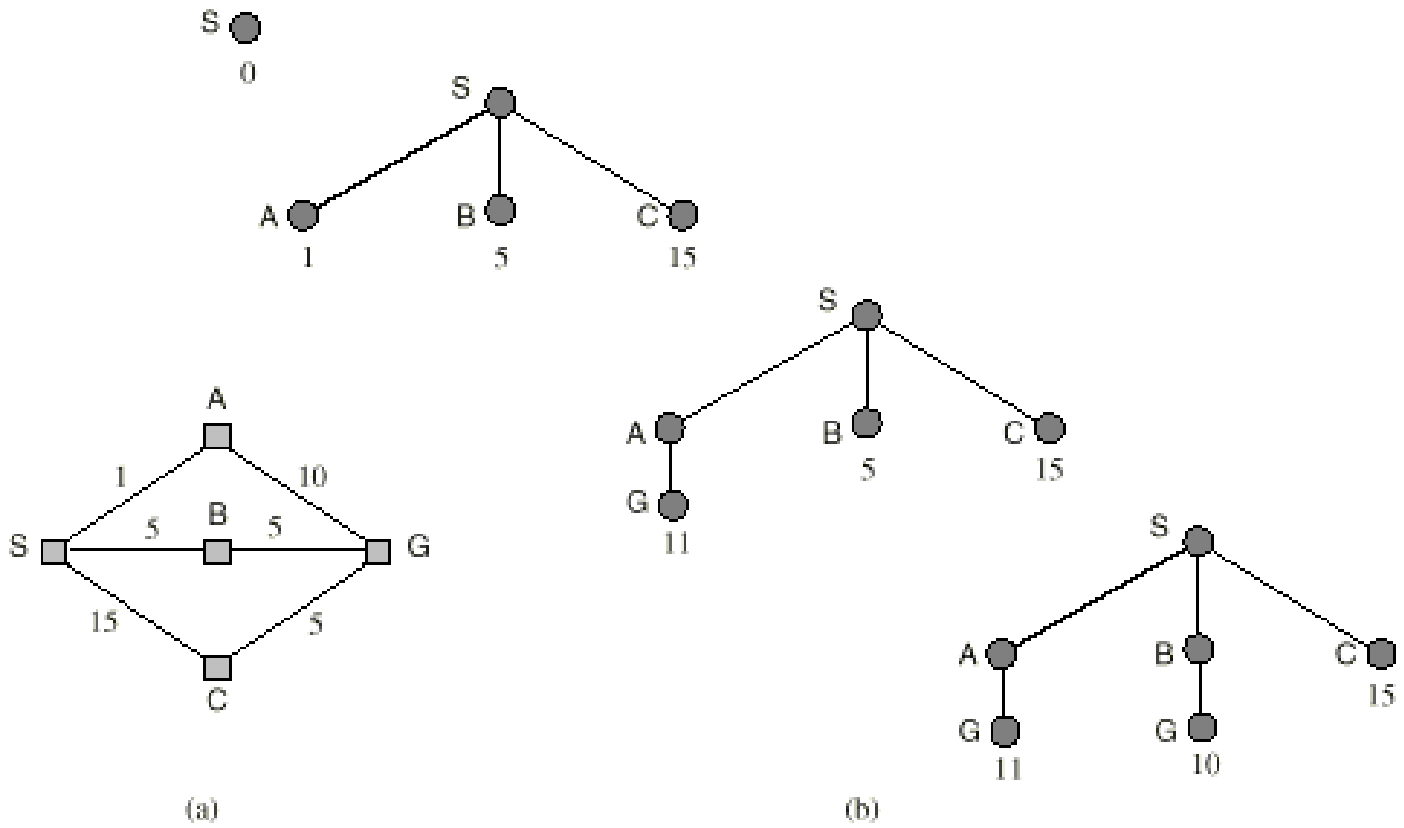
- **function** BUSCA-EM-LARGURA(*problema*) **returns** uma solução ou falha
  - *nó* ← um *nó* com ESTADO = *problema*.ESTADO-INICIAL, CUSTO-CAMINHO = 0
  - **if** *problema*.TESTE-OBJETIVO(*nó*.ESTADO) **then returns** SOLUÇÃO(*nó*)
  - *fronteira* ← fila FIFO com *nó* sendo o único elemento
  - *explorados* ← conjunto vazio
  - **loop do**
    - *if* VAZIO?(*fronteira*) **then return** falha
    - *nó* ← POP(*fronteira*) /*escolhe o nó mais superior na fronteira*/
    - adiciona *nó*.ESTADO a *explorados*
    - **for each** *ação* em *problema*.AÇÕES(*nó*, ESTADO) **do**
      - *filho* ← NÓ-FILHO(*problema*, *nó*, *ação*)
      - **if** *filho*.ESTADO não está em *explorados* ou *fronteira* **then**
        - **if** *problema*.TESTE-OBJETIVO(*filho*.ESTADO) **then return** SOLUÇÃO(*filho*)
        - *fronteira* ← INSERIR(*filho*, *fronteira*)

Busca de custo uniforme 1

- Modifica a busca primeiro em largura expandindo o nó de menor custo na fronteira (mesmo que já tenha atingido o estado objetivo)
- Encontra a melhor solução se o custo do caminho nunca decrescer ao longo do caminho:

$$g(\text{SUCESSOR}(n)) \geq g(n)$$

Busca de custo uniforme 2



Busca de Custo Uniforme

- **function** BUSCA-CUSTO-UNIFORME(*problema*) **returns** uma solução ou falha
  - *nó* ← um *nó* com ESTADO = *problema*.ESTADO-INICIAL, CUSTO-CAMINHO = 0
  - *fronteira* ← uma fila de prioridade ordenada por CUSTO-CAMINHO, com *nó* como único elemento
  - *explorados* ← conjunto vazio
  - **loop do**
    - *if* VAZIO?(*fronteira*) **then return** falha
    - *nó* ← POP(*fronteira*) /*escolhe o nó mais inferior em *fronteira*/
    - **if** *problema*.TESTE-OBJETIVO(*nó*.ESTADO) **then return** SOLUÇÃO(*nó*)
    - adicionar *nó*.ESTADO a *explorados*
    - **for each** *ação* em *problema*.AÇÕES(*nó*, ESTADO) **do**
      - *filho* ← NÓ-FILHO(*problema*, *nó*, *ação*)
      - **if** *filho*.ESTADO não está em *explorados* ou *fronteira* **then**
        - *fronteira* ← INSERIR(*filho*, *fronteira*)
      - **else if** *filho*.ESTADO está em *fronteira* com mais alto CUSTO-CAMINHO **then**
        - substituir o nó *fronteira* por *filho*

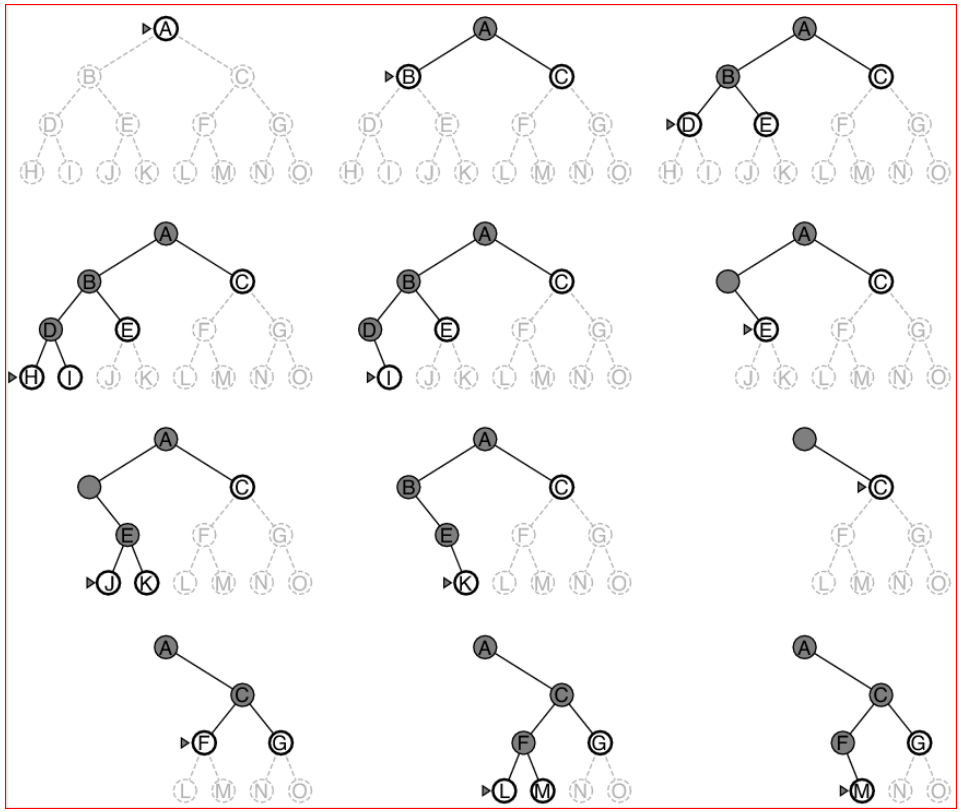
Busca primeiro em profundidade 1

- Expande sempre o nó no nível mais profundo da árvore de busca
- Somente quando atinge um nó não-objetivo sem expansão ocorre o retorno para um nível superior

function BUSCA-PRIMEIRO-PROFUNDIDADE(*problema*)
return uma solução ou falha

BUSCA-GERAL(*problema*, ENFILEIRAR-NA-FRENTE)

Busca primeiro em profundidade 2



Busca com profundidade limitada

- Coloca um limite na máxima profundidade de um caminho
- O limite pode ser implementado em um algoritmo especial de busca com profundidade limitada ou usando o algoritmo de busca geral com operadores que mantenham um registro da profundidade

Busca Profundidade Limitada

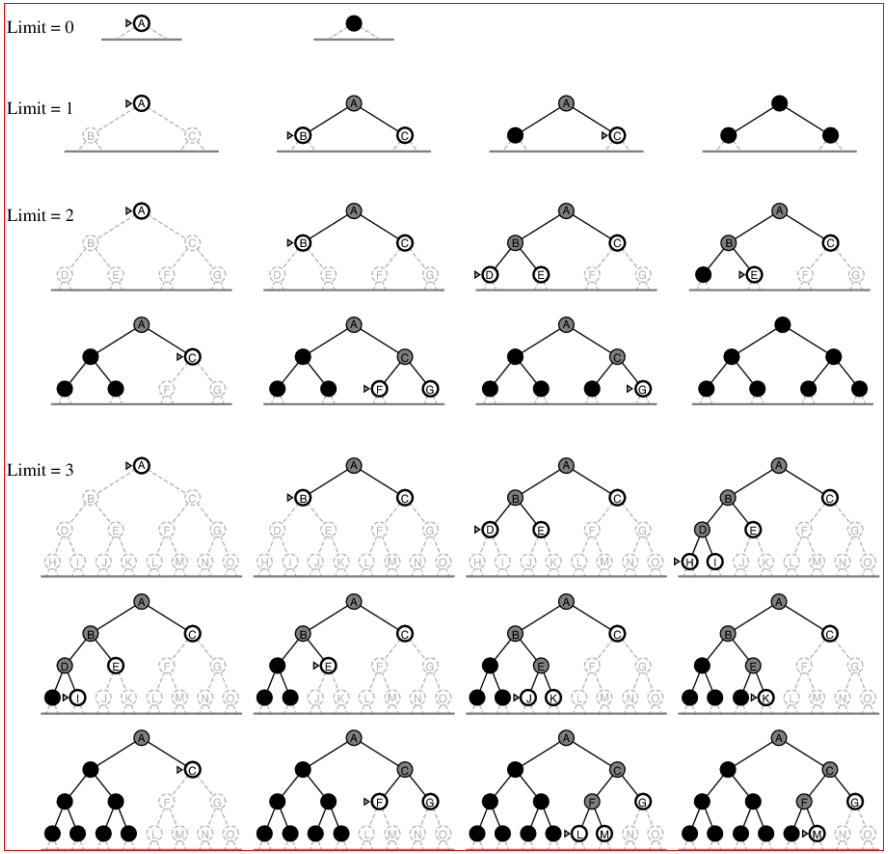
- **function** BUSCA-PROFUNDIDADE-LIMITADA(*problema*, *limite*) **returns** uma solução ou *falha/corte*
  - **return** DLS-RECURSIVO(CRIAR-NÓ(*problema*.ESTADO-INICIAL), *problema*, *limite*)
- **function** DLS-RECURSIVO(*nó*, *problema*, *limite*) **returns** uma solução ou *falha/corte*
  - **if** *problema*.TESTE-OBJETIVO(*nó*.ESTADO) **then return** SOLUÇÃO(*nó*)
  - ***else if*** *limite* = 0 **then return** *corte*
  - **else**
    - *ocorreu_corte?* ← falso
    - **for each** *ação* **in** *problema*.AÇÕES(*nó*.ESTADO) **do**
      - *filho* ← NÓ-FILHO(*problema*, *nó*, *ação*)
      - *resultado* ← DLS-RECURSIVO(*filho*, *problema*, *limite* – 1)
      - **if** *resultado* = *corte* **then** *ocorreu_corte* ← verdadeiro
      - **else if** *resultado* ≠ *falha* **then return** *resultado*
    - **if** *ocorreu_corte?* **then return** *corte* **else return** *falha*

Busca com aprofundamento iterativo 1

Segue testando com várias profundidades limites: 0, 1, 2, ... até encontra a solução

```
function BUSCA-APROFUNDAMENTO-ITERATIVO
    (problema)              return uma sequência solução
inputs: problema, um problema
   for profundidade 0 to ∞ do
      if BUSCA-PROFUNDIDADE-LIMITADA(problema,
           profundidade) tem sucesso then return
           resultado
    end
return falha
```

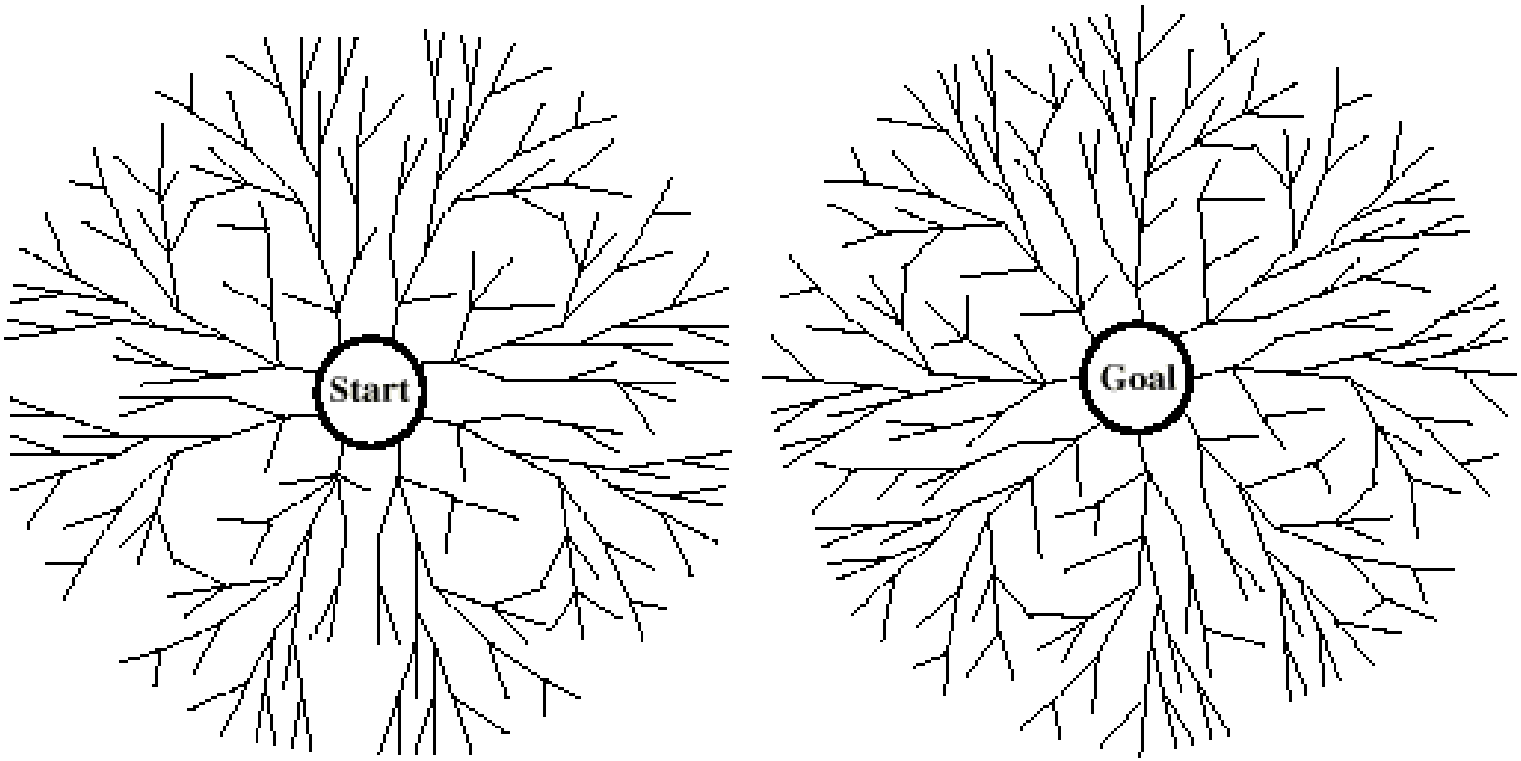
Busca com aprofundamento iterativo - 2



Busca Aprofundamento Iterativo

- **function** BUSCA-APROFUNDAMENTO-ITERATIVO(*problema*) **returns** uma solução ou falha
  - **for** *profundidade* = 0 **to** ∞ **do**
    - *resultado* ← BUSCA-PROFUNDIDADE-LIMITADA(*problema*, *profundidade*)
    - **if** *resultado* ≠ *corte* **then return** *resultado*

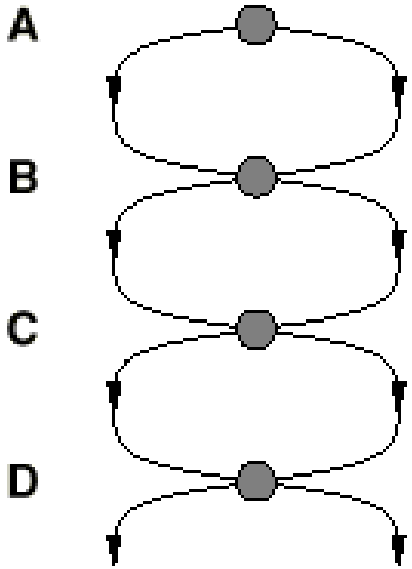
Busca bidirecional

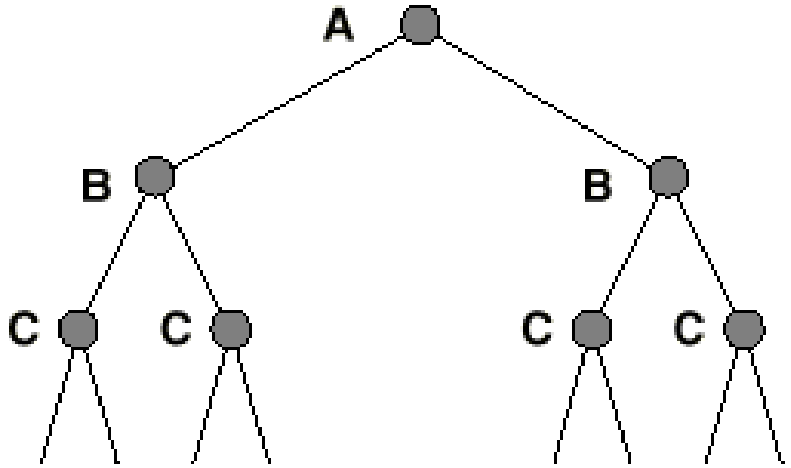


Comparação das estratégias

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

Evitando repetição de estados 1





Evitando repetição de estados 2

- Não retornar ao estado anterior
- Não criar caminhos que contenham ciclos
- Não gerar qualquer estado que já tenha sido gerado anteriormente

Sumário

- Formulação de problemas geralmente requer a abstração de detalhes do mundo real para definir um espaço de estado que pode ser explorado de maneira viável.
- Vários tipos de estratégias de busca não informada.
- Busca com aprofundamente iterative requer um espaço linear e não utiliza muito tempo a mais que os outros algoritmos não informados.