Inteligência Artificial

Aula 2

Profª Bianca Zadrozny
http://www.ic.uff.br/~bianca/ia

Resolução de problemas por meio de busca

Capítulo 3 – Russell & Norvig
Seções 3.1, 3.2 e 3.3

Agentes de resolução de problemas

- Agentes reativos não funcionam em ambientes para quais o número de regras condição-ação é grande demais para armazenar.
- Nesse caso podemos construir um tipo de agente baseado em objetivo chamado de agente de resolução de problemas.

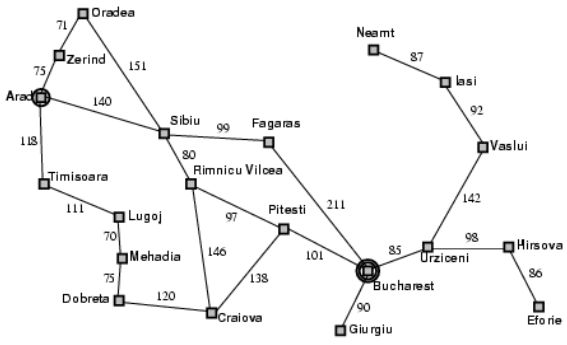
Busca

- Um agente com várias opções imediatas pode decidir o que fazer comparando diferentes sequências de ações possíveis.
- Esse processo de procurar pela melhor sequência é chamado de busca.
- Formular objetivo → buscar → executar

Exemplo: Romênia

- De férias na Romênia; atualmente em Arad.
- Vôo sai amanhã de Bucareste.
- Formular objetivo:
  - Estar em Bucareste
- Formular problema:
  - estados: cidades
  - ações: dirigir entre as cidades
- Encontrar solução:
  - sequência de cidades, ex., Arad, Sibiu, Fagaras, Bucareste.

Exemplo: Romênia



Formulação de problemas

Um problema é definido por quatro itens:

1. Estado inicial ex., "em Arad"
2. Ações ou função sucessor *S(x)* = conjunto de pares estado-ação
   - ex., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. Teste de objetivo, pode ser
   - explícito, ex., *x* = "em Bucharest"
   - implícito, ex., *Cheque-mate(x)*
   -
4. Custo de caminho (aditivo)
   - ex., soma das distâncias, número de ações executadas, etc.
   - *c(x,a,y)* é o custo do passo, que deve ser sempre ≥ 0

- Uma solução é uma sequência de ações que levam do estado inicial para o estado objetivo.
- Uma solução ótima é uma solução com o menor custo de caminho.

Agente de Resolução de Problemas

```
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 do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Supõe que ambiente é estático, observável, discreto e determinístico.

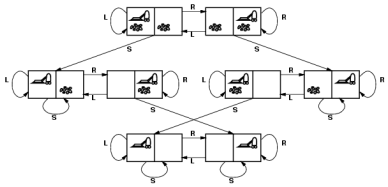
Espaço de estados

- O conjunto de todos os estados acessíveis a partir de um estado inicial é chamado de espaço de estados.
  - Os estados acessíveis são aqueles dados pela função sucessora.
- *O espaço de estados pode ser interpretado como um grafo em que os nós são estados e os arcos são ações*.

Selecionando um espaço de estados

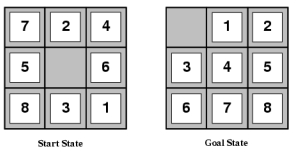
- O mundo real é absurdamente complexo
  → o espaço de estados é uma abstração
- Estado (abstrato) = conjunto de estados reais
- Ação (abstrata) = combinação complexa de ações reais
  - ex., "Arad → Zerind" representa um conjunto complexo de rotas, desvios, paradas, etc.
  - Qualquer estado real do conjunto "em Arad" deve levar a algum estado real "em Zerind".
- Solução (abstrata) = conjunto de caminhos reais que são soluções no mundo real
- A abstração é útil se cada ação abstrata é mais fácil de executar que o problema original.

Exemplo 1: Espaço de Estados do Mundo do Aspirador de Pó



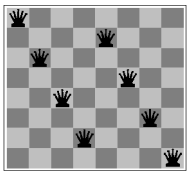
- **Estados:** Definidos pela posição do robô e sujeira (8 estados)
- **Estado inicial:** Qualquer um
- **Função sucessor:** pode-se executar qualquer uma das ações em cada estado (esquerda, direita, aspirar)
- **Teste de objetivo:** Verifica se todos os quadrados estão limpos
- **Custo do caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho

Exemplo 2: O quebra-cabeça de 8 peças



- **Estados:** Especifica a posição de cada uma das peças e do espaço vazio
- **Estado inicial:** Qualquer um
- **Função sucessor:** gera os estados válidos que resultam da tentativa de executar as quatro ações (mover espaço vazio para esquerda, direita, acima ou abaixo)
- **Teste de objetivo:** Verifica se o estado corresponde à configuração objetivo.
- **Custo do caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho

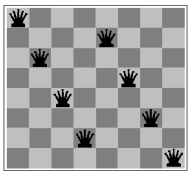
Exemplo 3:Oito rainhas
Formulação incremental



Quasi solução

- **Estados:** qualquer disposição de 0 a 8 rainhas
- **Estado inicial**: nenhuma rainha
- **Função sucessor**: colocar 1 rainha em qualquer vazio
- **Teste**: 8 rainhas no tabuleiro, nenhuma atacada
- 64x63x...57 = $3x10^{14}$ sequências para investigar

Exemplo 3:Oito rainhas
Formulação de estados completos



Quasi solução

- **Estados:** disposições de n rainhas, uma por coluna, nas n colunas mais a esquerda sem que nenhuma rainha ataque outra
- **Função sucessor**: adicionar uma rainha a qualquer quadrado na coluna vazia mais à esquerda, de tal modo que ela não seja atacada
- Tamanho do espaço de estados: 2.057

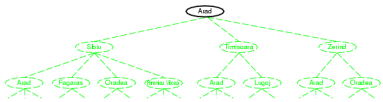
Problemas do mundo real

- Problema de roteamento
  - encontrar a melhor rota de um ponto a outro (aplicações: redes de computadores, planejamento militar, planejamento de viagens aéreas)
- Problemas de tour
  - visitar cada ponto pelo menos uma vez
- Caixeiro viajante
  - visitar cada cidade exatamente uma vez
  - encontrar o caminho mais curto
- Layout de VLSI
  - posicionamento de componentes e conexões em um chip
- Projeto de proteínas
  - encontrar uma sequência de aminoácidos que serão incorporados em uma proteína tridimensional para curar alguma doença.

Busca de soluções

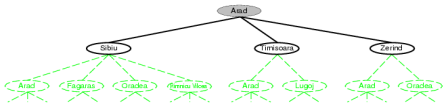
- Idéia: Percorrer o espaço de estados a partir de uma ***árvore de busca***.
- *Expandir* o estado atual aplicando a função sucessor, *gerando* novos estados.
- Busca: seguir um caminho, deixando os outros para depois.
- A *estratégia de busca* determina qual caminho seguir.

Exemplo de árvore de busca



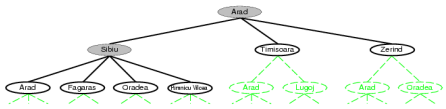
Estado inicial

Exemplo de árvore de busca



Depois de expandir Arad

Exemplo de árvore de busca



Depois de expandir Sibiu

Descrição informal do algoritmo geral de busca em árvore

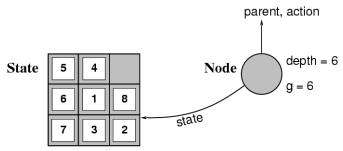
function TREE-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node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

Árvore de busca não é equivalente a espaço de estados!

- Há 20 estados no mapa da Romênia (espaço de estados), mas infinitos caminhos a percorrer. Portanto a árvore de busca, neste caso, tem tamanho infinito.
  - Caminho infinito: Arad-Sibiu-Arad-Sibiu-Arad-...

Estados vs. nós

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



- A função `Expand` cria novos nós, preenchendo os vários campos e usando a função sucessor do problema para gerar os estados correspondentes.
- A coleção de nós que foram gerados, mas ainda não foram expandidos é chamada de borda (ou fringe)
  - Geralmente implementados como uma fila.
  - A maneira como os nós entram na fila determina a estratégia de busca.

Algoritmo geral de busca em árvore

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** SOLUTION(*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(*node*, *action*, *s*)
    DEPTH[*s*] ← DEPTH[*node*] + 1
    add *s* to *successors*
  return *successors*

Estratégias de busca

- Uma estratégia de busca é definida pela escolha da ordem da expansão de nós
- Estratégias são avaliadas de acordo com os seguintes critérios:
  - completeza: o algoritmo sempre encontra a solução se ela existe?
  - complexidade de tempo: número de nós gerados
  - complexidade de espaço: número máximo de nós na memória
  - otimização: a estratégia encontra a solução ótima?
- Complexidade de tempo e espaço são medidas em termos de:
  - *b:* máximo fator de ramificação da árvore (número máximo de sucessores de qualquer nó)
  - *d:* profundidade do nó objetivo menos profundo
  - *m*: o comprimento máximo de qualquer caminho no espaço de estados (pode ser ∞)

Estratégias de Busca Sem Informação (ou Busca Cega)

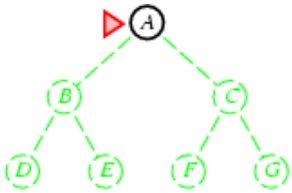
- Estratégias de busca sem informação usam apenas a informação disponível na definição do problema.
  - Apenas geram sucessores e verificam se o estado objetivo foi atingido.
- As estratégias de busca sem informação se distinguem pela ordem em que os nós são expandidos.
  - Busca em extensão (*Breadth-first*)
  - Busca de custo uniforme
  - Busca em profundidade (*Depth-first*)
  - Busca em profundidade limitada
  - Busca de aprofundamento iterativo

Estratégias de busca

- Estratégias são avaliadas de acordo com os seguintes critérios:
  - completeza: o algoritmo sempre encontra a solução se ela existe?
  - complexidade de tempo: número de nós gerados
  - complexidade de espaço: número máximo de nós na memória
  - otimização: a estratégia encontra a solução ótima?
- Complexidade de tempo e espaço são medidas em termos de:
  - *b:* máximo fator de ramificação da árvore (número máximo de sucessores de qualquer nó)
  - *d:* profundidade do nó objetivo menos profundo
  - *m*: o comprimento máximo de qualquer caminho no espaço de estados (pode ser ∞)

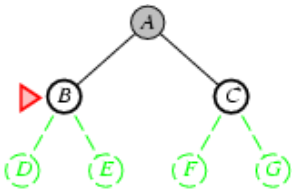
Busca em extensão

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



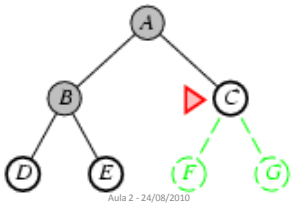
Busca em extensão

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



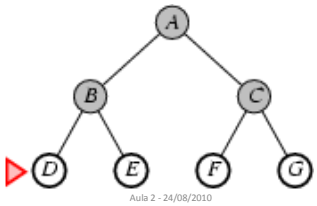
Busca em extensão

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



Busca em extensão

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



Propriedades da busca em extensão

- Completa? Sim (se *b* é finito)
- Tempo? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Espaço? $O(b^{d+1})$ (mantém todos os nós na memória)
- Ótima? Sim (se todas as ações tiverem o mesmo custo)

Requisitos de Tempo e Memória para a Busca em Extensão

- Busca com fator de ramificação b=10.
- Supondo que 10.000 nós possam ser gerados por segundo e que um nó exige 1KB de espaço.

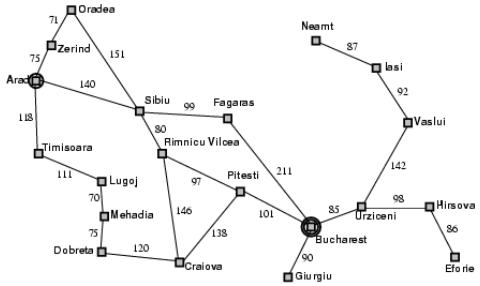
Profundidade	Nós	Tempo	Memória
2	1100	0.11 segundo	1 megabyte
4	111.100	11 segundos	106 megabytes
6	10^7	19 minutos	10 gigabytes
8	10^9	31 horas	1 terabyte
10	10^{11}	129 dias	101 terabytes
12	10^{13}	35 anos	10 petabytes
14	10^{15}	3.523 anos	1 exabyte

Busca de custo uniforme

- Expande o nó não-expandido que tenha o caminho de custo mais baixo.
- Implementação:
  - *borda* = fila ordenada pelo custo do caminho
- Equivalente a busca em extensão se os custos são todos iguais
- Completa? Sim, se o custo de cada passo ≥ ε
- Tempo? # de nós com $g \leq$ custo da solução ótima, $O(b^{\lceil C^*/\varepsilon \rceil})$ onde C^* é o custo da solução ótima
- Espaço? de nós com $g \leq$ custo da solução ótima, $O(b^{\lceil C^*/\varepsilon \rceil})$
- Ótima? Sim pois os nós são expandidos em ordem crescente de custo total.

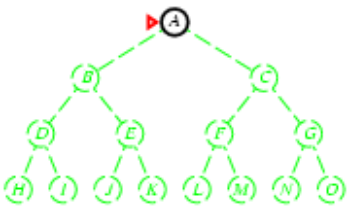
Exercício

- Aplicar busca de custo uniforme para achar o caminho mais curto entre Arad e Bucareste.



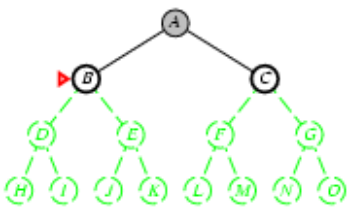
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



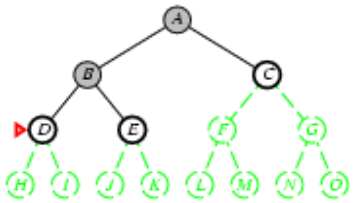
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



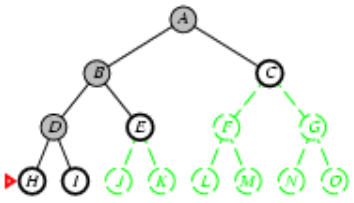
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



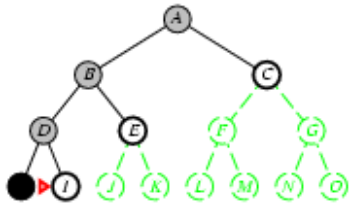
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



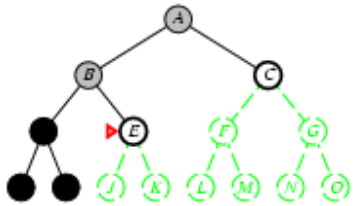
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



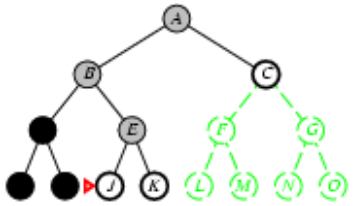
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



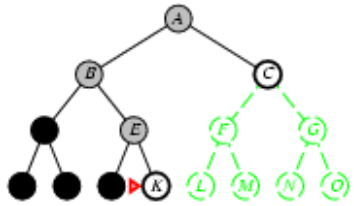
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



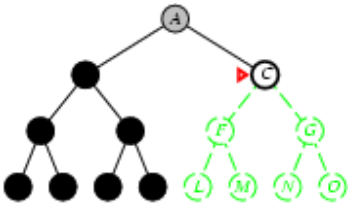
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



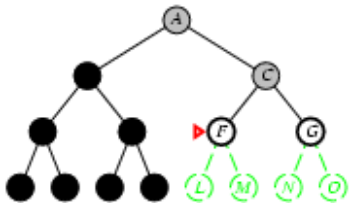
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



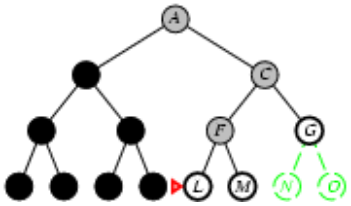
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



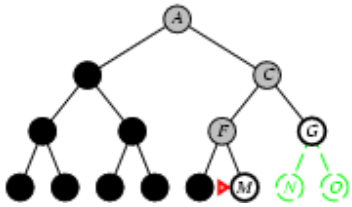
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



Propriedades da Busca em Profundidade

- Completa? Não: falha em espaços com profundidade infinita, espaços com loops
  - Se modificada para evitar estados repetidos é completa para espaços finitos
- Tempo? $O(b^m)$: péssimo quando m é muito maior que d.
  - mas se há muitas soluções pode ser mais eficiente que a busca em extensão
- Espaço? $O(bm)$, i.e., espaço linear!
  - 118 kilobytes ao invés de 10 petabytes para busca com b=10, d=m=12
- Ótima? Não

Busca em Profundidade Limitada

= busca em profundidade com limite de profundidade *l*, isto é, nós com profundidade *l* não tem 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 SOLUTION(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
```

Propriedades da Busca em Profundidade Limitada

- Completa? Não; a solução pode estar além do limite.
- Tempo? $O(b^l)$
- Espaço? $O(bl)$
- Ótima? Não

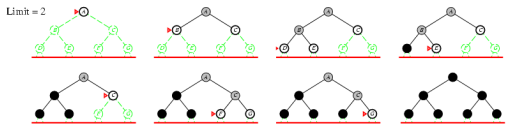
Busca de Aprofundamento Iterativo em Profundidade

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem
  for *depth* ← 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

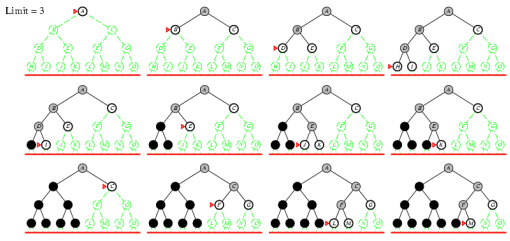
Busca de Aprofundamento Iterativo em Profundidade *l* =0

Busca de Aprofundamento Iterativo em Profundidade *l* =1

Busca de Aprofundamento Iterativo em Profundidade *l* =2



Busca de Aprofundamento Iterativo em Profundidade *l* =3



Busca de Aprofundamento Iterativo

- Número de nós gerados em uma busca de extensão com fator de ramificação *b*:

$$N_{BE} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b)$$

- Número de nós gerados em uma busca de aprofundamento iterativo até a profundidade *d* com fator de ramificação *b*:

$$N_{BAI} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- Para *b = 10, d = 5*,
  - $N_{BE} = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$
  - $N_{BAI} = 6 + 50 + 400 + 3.000 + 20.000 + 100.000 = 123.456$
- Overhead = (123.456 – 111.111)/111.111 = 11%

Propriedades da busca de aprofundamento iterativo

- Completa? Sim
- Tempo? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Espaço? $O(bd)$
- Ótima? Sim, se custo de passo = 1

Resumo dos algoritmos

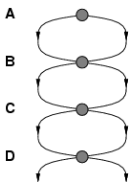
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

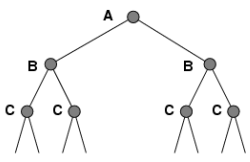
Estados repetidos

- O processo de busca pode perder tempo expandindo nós já explorados antes
  - Estados repetidos podem levar a loops infinitos
  - Estados repetidos podem transformar um problema linear em um problema exponencial

Estados Repetidos

- Não detectar estados repetidos pode transformar um problema linear em um problema exponencial.





Detecção de estados repetidos

- Comparar os nós prestes a serem expandidos com nós já visitados.
  - Se o nó já tiver sido visitado, será descartado.
  - Lista "closed" (fechado) armazena nós já visitados.
    - Busca em profundidade e busca de aprofundamento iterativo não tem mais espaço linear.
  - A busca percorre um grafo e não uma árvore.

```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Resumo

- A formulação de problemas usualmente requer a abstração de detalhes do mundo real para que seja definido um espaço de estados que possa ser explorado através de algoritmos de busca.
- Há uma variedade de estratégias de busca sem informação (ou busca cega).
- A busca de aprofundamento iterativo usa somente espaço linear e não muito mais tempo que outros algoritmos sem informação.