# RESOLUÇÃO DE PROBLEMAS (BUSCA)

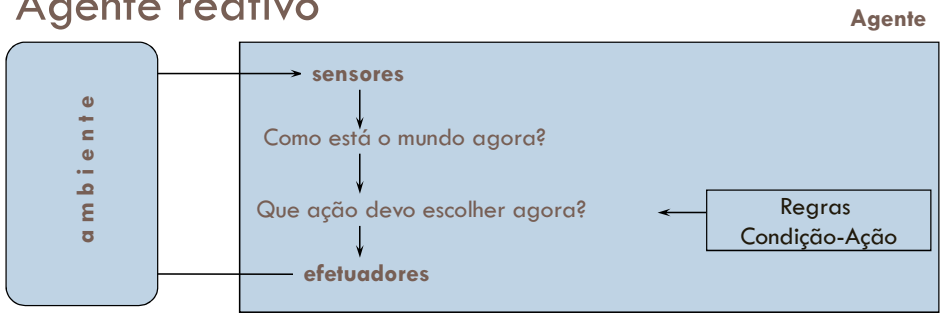
Prof. Marcilio Carlos Pereira de Souto – UFRN
Adaptado por Profa. Patricia Jaques - Unisinos

## Resolução de Problemas



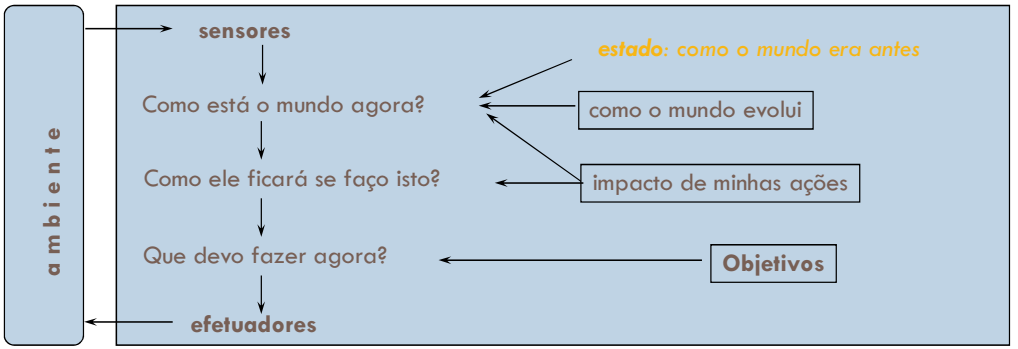
- Apresentação baseada nos capítulos 3 e 4 do livro:
  - **RUSSEL**, S. J. and **NORVIG**, P. Artificial intelligence : a modern approach. Upper Saddle River : Prentice-Hall, 1995. 932 p.

## Agente reativo




- Escolhe suas ações com base apenas nas percepções atuais
  - não pode pensar no futuro, não sabe "aonde vai"


## Agente baseado em objetivo




- Já o agente baseado em objetivo, sabe:
  - pois segue um **objetivo** explícito
  - sabe como o mundo evolui e, por isso, pode verificar se o resultado de suas ações levam ao objetivo

## Agente de Resolução de Problemas (2/2)

5

- Dentre as maneiras de implementar um agente baseado em objetivo existe o chamado Agente de Resolução de Problemas
  - serve para alguns tipos de problemas
  - requer pouco conhecimento explícito
  - basicamente busca uma seqüência de ações que leve a estados desejáveis (objetivos)
- Questões
  - O que é um problema e como formulá-lo?
  - Como buscar a solução do problema?

## Problemas e Soluções bem Definidos (1/2)

6

*Um problema na RP é definido em termos de...*

1) um espaço de estados possíveis, incluindo um estado inicial e um estado final (objetivo)
   - exemplo 1: dirigir de Natal a Caicó
   - exemplo 2: jogo de 8-números


2) um conjunto de ações (ou operadores) que permitem passar de um estado a outro
   - ex1. dirigir de uma cidade a outra
   - ex2. mover uma peça do jogo de n-números (*n-puzzle*)

## Problemas e Soluções bem Definidos (2/2)

7

- Espaço de Estados:
  - conjunto de todos os estados alcançáveis a partir do estado inicial por qualquer seqüência de ações.
- Formulação de objetivos:
  - propriedade abstrata
    - ex.: condição de xeque-mate no Xadrez
  - conjunto de estados finais do mundo
    - ex.: estar em na cidade-destino
- Solução:
  - caminho (seqüência de *ações* ou *operadores*) que leva do estado inicial a um estado final (objetivo).

## Solucionando o Problema:

### formulação, busca e execução

8

- Formulação do problema e do objetivo:
  - quais são os ***estados*** e as ***ações*** a considerar?
  - qual é (e como representar) o **objetivo**?
- Busca (solução do problema):
  - processo que gera/analisa seqüências de ações para alcançar um objetivo
  - *solução* = caminho (seqüência de ações) entre estado inicial e estado final.
- Execução:
  - Executar (passo a passo) a solução ***completa*** encontrada

# Agente Resolução de Problemas

## formulação, busca e execução



função **Agente-Simples-RP**(*p*) retorna uma *ação*

entrada: *p*, um dado perceptivo

*estado* ← **Atualiza-Estado** (*estado, p*)
se *s* (seqüência de ações) está vazia
então
  o (objetivo) ← **Formula-Objetivo** (*estado*)
  *problema* ← **Formula-Problema** (*estado, o*)
  *s* ← **Busca** (*problema*)
*ação* ← **Primeira** (*s, estado*)
*s* ← **Resto** (*s, estado*)
retorna *ação*

# Ambiente



- Estático:
  - Não muda enquanto o agente está realizando a resolução do problema
- Observável:
  - pois o estado inicial deve ser conhecido
- Discreto:
  - existe um nro. distinto e claramente definido de percepções em cada turno
- Determinístico:
  - próximo estado do agente pode ser completamente determinado pelo estado atual + ação

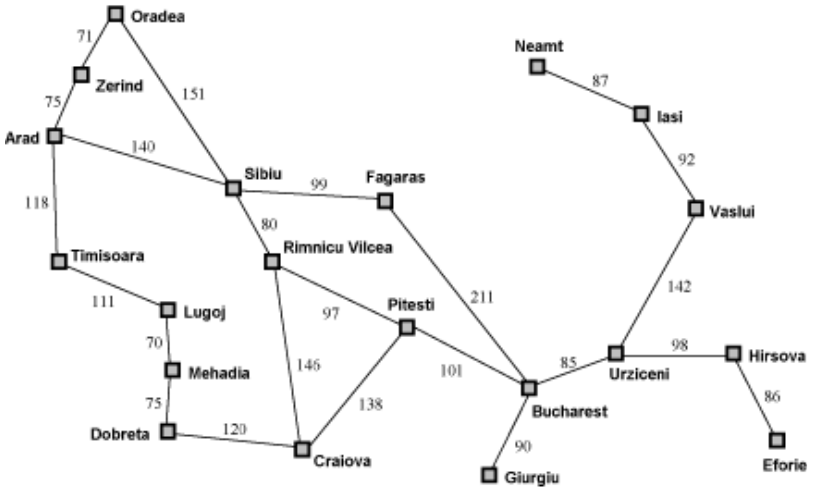
# Medida de Desempenho na Busca



- Desempenho de um algoritmo de busca:
  - 1. O algoritmo encontrou alguma solução?
  - 2. É uma boa solução?
    - **custo de caminho** (qualidade da solução)
  - 3. É uma solução computacionalmente barata?
    - **custo da busca** (tempo e memória)
- Custo total
  - custo do caminho + custo de busca
- Espaço de estados grande:
  - compromisso (conflito) entre a melhor solução (solução ótima) e a solução mais barata

# Outro Exemplo: Ir de Arad a Bucharest





# Exemplo Romênia

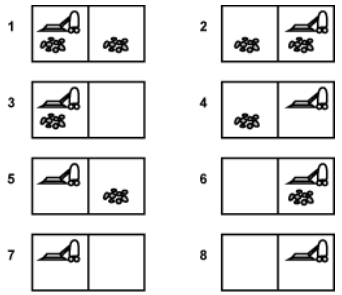
13

- Ida para Bucharest:
  - **estados** = cada possível cidade do mapa
  - **estado inicial** = Arad
  - **teste de término** = estar em Bucarest
  - **operadores** = dirigir de uma cidade para uma de suas cidades vizinhas
  - **custo do caminho** = distância percorrida

# Mais um Exemplo...

14

- Aspirador de pó
  - **estados** = 8 estados possíveis
  - **estado inicial** = qualquer estado escolhido
  - **teste de término** = verifica se todos os quadrados estão limpos
  - **operadores** = mover esquerda, mover direita, limpar
  - **custo da solução** = cada passo custa 1, assim o custo do caminho é o nro. de passos realizados




# Custo Diferente => Solução Diferente

15

- Função de *custo de caminho*

  (1) número de cidades visitadas,

  (2) distância entre as cidades,

  (3) tempo de viagem, etc.

- Solução mais barata:

  (1) Canudos, Belém do S. Francisco, Salgueiro, ...

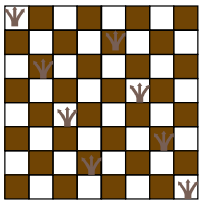
  (2) Canudos, Belém do S. Francisco, Salgueiro, ...

  (3) Canudos, Juazeiro, Pretrolina, Cabrobó, Salgueiro

# Importância da formulação: 8 rainhas

16

- Jogo das 8 Rainhas
  - dispor 8 rainhas no tabuleiro de forma que não possam se "atacar"
    - não pode haver mais de uma rainha em uma mesma linha, coluna ou diagonal
  - somente o custo da busca conta
    - não existe custo de caminho



- Existem diferentes estados e operadores possíveis
  - essa escolha pode ter conseqüências boas ou nefastas na complexidade da busca ou no tamanho do espaço de estados

# Importância da formulação: 8 rainhas

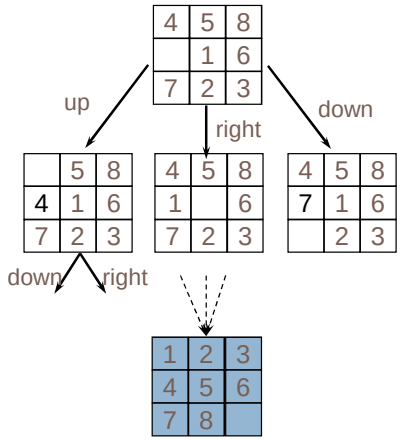


- Formulação A
  - estados: qualquer disposição com n (n ≤ 8) rainhas
  - estado inicial: nenhuma rainha no tabuleiro
  - operadores: adicionar uma rainha no tabuleiro (em qualquer quadrado)
  - teste objetivo: 8 rainhas estão no tabuleiro e nenhuma é atacada
  - $64^8$ seqüências possíveis: vai até o fim para testar se dá certo
- Formulação B
  - estados: disposição com n (n ≤ 8) rainhas sem ataque mútuo (teste gradual)
  - operadores: adicionar uma rainha na coluna vazia mais à esquerda em que não possa ser atacada
  - melhor (2057 possibilidades), mas pode não haver ação possível

# Importância da formulação: 8-números



- Jogo de 8 números:
  - **estados** = cada possível configuração do tabuleiro
  - **estado inicial** = qualquer um dos estados possíveis
  - **teste de término** = ordenado, com branco na posição [3,3]
  - **operadores** = mover branco (esquerda, direita, para cima e para baixo)
  - **custo da solução** = número de passos da solução






# ALGUMAS APLICAÇÕES

# Aplicações de Busca: "Toy Problems"



- Jogo das n rainhas
- Jogo dos n números (*n-puzzle*)
- Criptoaritmética

```
  send
+ more
--------
 money
```

- Torre de Hanoi
- Palavras cruzadas
- Canibais e missionários

## Aplicações: Problemas Reais

21

- Cálculo de rotas (pathfinding)
  - rotas em redes de computadores
  - sistemas de planejamento de viagens
  - planejamento de rotas de aviões
  - Caixeiro viajante
  - Jogos de computadores (rotas dos personagens)
- Alocação (Scheduling)
  - Salas de aula
  - Máquinas industriais (job shop)
- Projeto de VLSI
  - Cell layout
  - Channel routing

## Aplicações: Problemas Reais

22

- Navegação de robôs:
  - generalização do problema da navegação
  - robôs movem-se em espaços contínuos, com um conjunto (infinito) de possíveis ações e estados
    - controlar os movimentos do robô no chão, e de seus braços e pernas requer espaço multi-dimensional
- Montagem de objetos complexos por robôs:
  - ordenar a montagem das diversas partes do objeto
  - ex. motor elétrico
- etc...

23

# PROBLEMAS COM INFORMAÇÃO PARCIAL

## Problemas com informação Parcial

24

- Até agora só vimos problemas de estado único
  - o agente sabe em que estado está e pode determinar o efeito de cada uma de suas ações
    - sabe seu estado depois de uma seqüência qualquer de ações
  - Solução: seqüência de ações
- Porém existem 3 outros tipos de problemas...

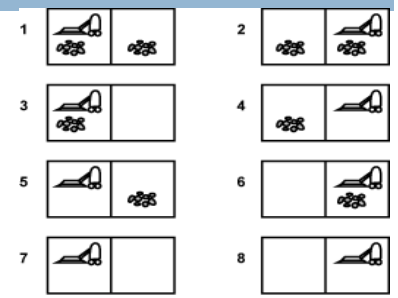
# Problemas com Informação Parcial



- Sensorless or conformant problem
  - Agente não sabe seu estado inicial (percepção deficiente)
  - Deve raciocinar sobre os conjuntos de estados
  - Solução: seqüência de ações (via busca)
- Problema de contingência
  - Efeito das ações não-determinístico e/ou mundo parcialmente observável => novas percepções depois de ação
    - ex. aspirador que suja ao sugar e/ou só percebe sujeira localmente
  - Solução: árvore de decisão (via planejamento)
- Problema exploratório (on-line)
  - Espaço de estados desconhecido
    - ex. dirigir sem mapa
  - Solução.... via aprendizagem por reforço

# Problemas com Informação Parcial

- Estado simples
  - Início: 5, Solução: [dir, suga]
- Conformant problem
  - Percepção deficiente
  - Início: {1,2,3,4,5,6,7,8}
  - Direita => {2,4,6,8}, Sugar => {4,8},...
  - Solução: [dir, suga, esq, suga]
- Problema de contingência
  - Efeito das ações não-determinístico
  - Início: [lado esq, sujo] = {1,3}
  - Solução? Sugar => {5,7}, Dir => {6,8}, Sugar no 6 => 8 mas sugar no 8 => 6
  - Solução: [sugar, dir, se sujo sugar]
  - Solução geral: [dir, se sujo suga, esq, se sujo suga]






# BUSCANDO SOLUÇÕES

Busca Cega

# Busca em Espaço de Estados



- Uma vez o problema bem formulado... o estado final deve ser “buscado”
- Em outras palavras, deve-se usar um método de busca para saber a ordem correta de aplicação dos operadores que lavará do estado inicial ao final
- Isto é feito por um processo de geração (de estados possíveis) e teste (para ver se o objetivo está entre eles)
- Uma vez a busca terminada com sucesso, é só executar a solução (= conjunto ordenado de operadores a aplicar)

# Busca em Espaço de Estados: Geração e Teste



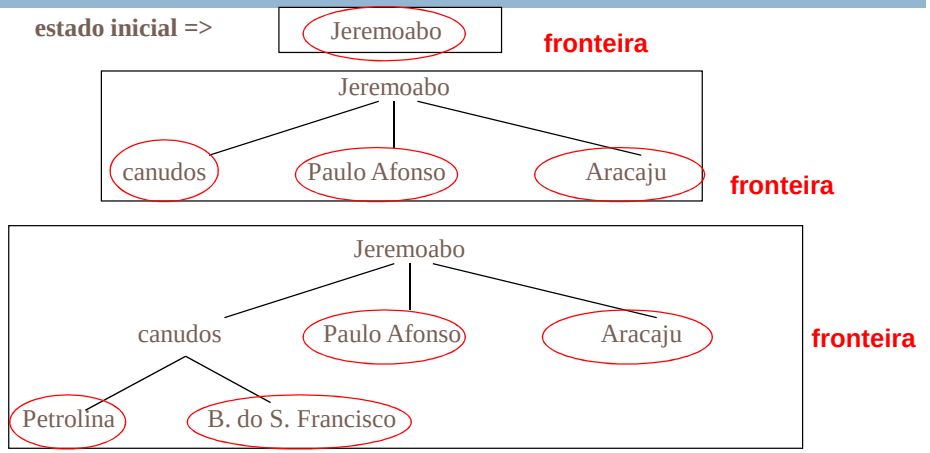
- *Fronteira* do espaço de estados
  - nós (estados) a serem expandidos no momento.
- Algoritmo:

Obs: começa com a fronteira contendo o estado inicial do problema.

1. Selecionar o primeiro nó (estado) da *fronteira* do espaço de estados;
   - se a fronteira está vazia, o algoritmo termina com falha.
2. Testar se o nó é um estado final (objetivo):
   - se "sim, então retornar nó - a busca termina com sucesso.
3. Gerar um novo conjunto de estados pela aplicação dos operadores ao nó selecionado;
4. Inserir os nós gerados na *fronteira*, de acordo com a estratégia de busca usada, e voltar para o passo (1).

# Exemplo: viajar de Jeremoabo a Cajazeiras






- Espaços de Estados: podem ser representados como uma árvore onde os estados são nós e as operações são arcos.

# Busca em Espaço de Estados: implementação



Algoritmo:

**Função-Insere:** controla a ordem de inserção de nós na fronteira do espaço de estados.

função **Busca-Genérica** (*problema*, **Função-Insere**)
retorna uma solução ou falha
*fronteira* ← **Faz-Fila** (**Faz-Nó** (**Estado-Inicial** [*problema*] ) )
loop do
se *fronteira* está vazia então retorna falha
*nó* ← **Remove-Primeiro** (*fronteira*)
se **Teste-Término** [*problema*] aplicado a **Estado** [*nó*] tiver sucesso
então retorna *nó*
*fronteira* ← **Função-Insere** (*fronteira*, **Operadores** [*problema*, *nó*])
end

# Métodos de Busca



- Busca exaustiva ou cega
  - Não sabe qual o **melhor** nó da fronteira a ser expandido = menor custo de caminho desse nó até um ***nó final*** (*objetivo*).
- Busca heurística - informada
  - Estima qual o melhor nó da fronteira a ser expandido com base em ***funções heurísticas*** => conhecimento

# Busca Cega

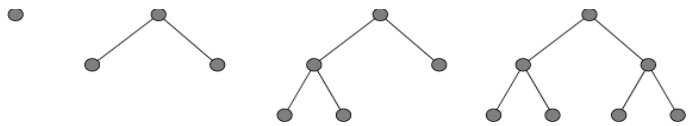
- Estratégias para determinar a ordem de ramificação dos nós:
  1. Busca em largura
  2. Busca de custo uniforme
  3. Busca em profundidade
  4. Busca com aprofundamento iterativo
- Direção da ramificação:
  1. Do estado inicial para um estado final
  2. De um estado final para o estado inicial
  3. Busca bi-direcional

## Critérios de Avaliação das Estratégias de Busca

- Completa?
  - a estratégia **sempre** encontra uma solução quando existe alguma?
- Ótima?
  - a estratégia encontra **a melhor solução** quando existem soluções diferentes?
    - menor custo de caminho
- Custo de tempo?
  - quanto **tempo** gasta para encontrar uma solução?
- Custo de memória?
  - quanta **memória** é necessária para realizar a busca?

# Busca em Largura

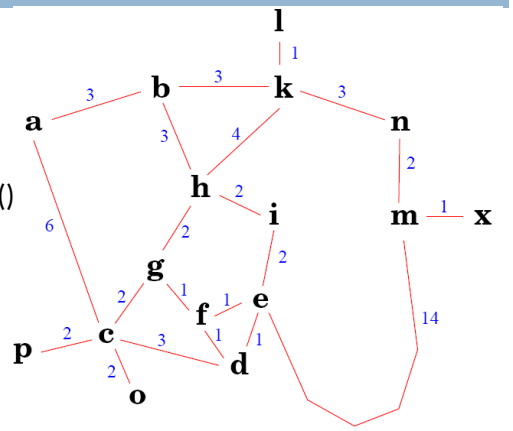
- Ordem de ramificação dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc…



# Busca em Largura



```
function BL(Estado inicial): Nodo
   Queue fronteira
   fronteira.add(new Nodo(inicial))
   while not fronteira.isEmpty() do
         Nodo n ← fronteira.remove()
         if n.getEstado().éMeta() then
                  return n
         end if
         fronteira.add(n.sucessores())
   end while
return null
```



## Evitar Geração de Estados Repetidos



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

## Evitar Estados Repetidos: soluções

- 1. Não retornar ao estado "pai"
- 2. Não retorna a um ancestral
- 3. Não gerar qualquer estado que já tenha sido criado antes (em qualquer ramo)
  - requer que todos os estados gerados permaneçam na memória: custo O(bd)
  - pode ser implementado mais eficientemente com hash tables
  - quando encontra nó igual tem de escolher o melhor (menor custo de caminho até então)

## Busca em Largura
## Como evitar estados repetidos

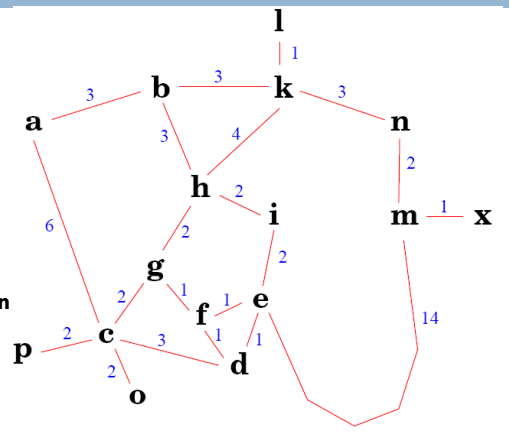


- Não inserir novamente na fronteira do pai do nó expandido
- Manter uma **lista fechada**:
  - Manter uma lista de nós que já foram expandidos, a qual chamaremos de **lista fechada**
  - Se o nó atual estiver na **lista fechada**, ele será descartado ao invés de ser expandido

## Busca em Largura



```
function BL(Estado inicial): Nodo
    Queue fronteira
    fronteira.add(new Nodo(inicial))
    while not fronteira.isEmpty() do
        Nodo n ← fronteira.remove()
        if n.getEstado().éMeta() then
            return n
        end if
        if n.getEstado() não está em fechado then
            fechado.add(n.getEstado())
            fronteira.add(n.sucessores())
        end if
    end while
return null
```



41

Se no final temos apenas o nó solução, como fazer para ter o caminho?

## Estrutura de dados Nó

42

- **Estado**: O Estado no espaço de estados a que o nó corresponde
- **Nó-pai**: o nó na árvore de busca que gerou esse nó.
- **Ação**: A ação que foi aplicada ao pai para gerar esse nó.
- **Custo do caminho**: O custo, tradicionalmente denotado por *g(n)*, do caminho desde o estado inicial até o nó indicado pelos ponteiros do pai.
- **Profundidade**: Número de passos ao longo do caminho, desde o estado inicial.

## Método de Expansão
## Obter sucessores de um nó

43

```
function sucessores (Nodo n, acao): lista de nós
   for (cada ação em  n.getEstado()) do
      s ← cria novo nó
      s.estado = resultado da ação em n
      s.pai = n
      s.acao = acao
      s.custo_caminho = n.custo_caminho + custo_passo (n, acao, s)
      s.profundidade = n.profundidade + 1
      adicionar s a sucessores
retornar sucessores
```

## Busca em Largura

- Esta estratégia é *completa*
  - o algoritmo encontra solução se ela existir
- É *ótima* ?
  - Sempre encontra a solução mais "rasa"
  - que nem sempre é a solução de menor **custo de caminho**, caso os operadores tenham valores diferentes
    - ex. ir para uma cidade D passando por B e C pode ser mais perto do que passando só por E
- Em outras palavras, é *ótima* se custo de caminho cresce com a profundidade do nó
  - isso ocorre quando todos os operadores têm o mesmo custo (=1)

# Busca em Largura

- Def. Fator de ramificação da árvore de busca:
  - número de nós gerados a partir de cada nó ($b$)
- Custo de tempo:
  - se o fator de ramificação do problema = $b$, e a primeira solução para o problema está no nível $d$,
  - então o número máximo de nós gerados até se encontrar a solução = $1 + b + b^2 + b^3 + \ldots + b^d$
  - **custo exponencial** = $O(b^d)$.
- Custo de memória:
  - todo nó gerado deve permanecer em memória (faz parte da fronteira ou é pai deste): então $O(b^d)$.
  - problema mais crucial: a *fronteira* do espaço de estados deve permanecer na memória
  - logo, busca em largura só dá bons resultados quando a profundidade da árvore de busca é pequena.

# Busca de Custo Uniforme (Dijkstra's Search)

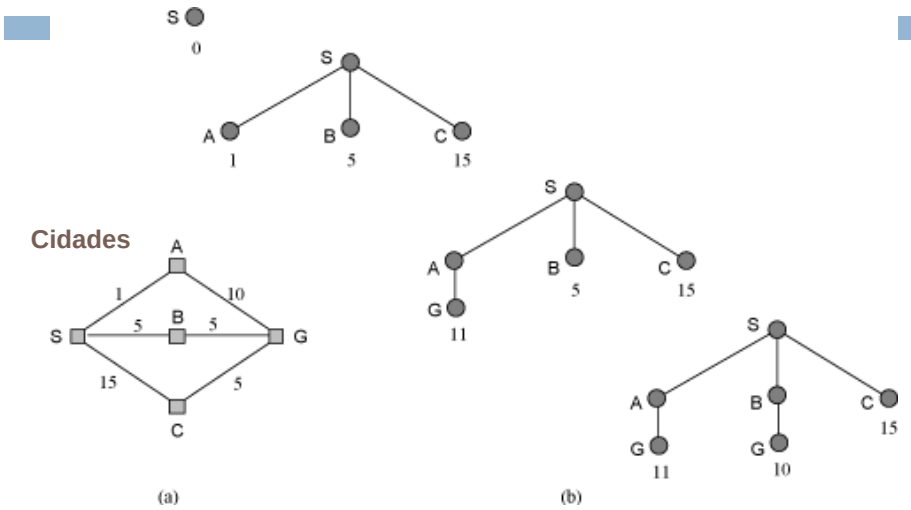
- Estende a busca em largura:
  - expande o nó da **fronteira** com **menor custo de caminho** até o momento
  - cada operador pode ter um custo associado diferente, medido pela função g(n) que dá o custo do caminho da origem ao nó n

# Busca de Custo Uniforme (Dijkstra's Search)



```
function Uniforme (Estado inicial): Nodo
    PriorityQueue(g) fronteira {fila ordenada por g}
    fronteira.add(new Nodo(inicial))
    while not fronteira.isEmpty() do
            Nodo n ← fronteira.remove()
            if n.getEstado().éMeta() then
                    return n
            end if
            if n.getEstado() não está em fechado then
                    fechado.add(n.getEstado())
                    fronteira.add(n.sucessores())
            end if
    end while
return null
```

# Busca de Custo Uniforme



# Busca de Custo Uniforme
# Fronteira do exemplo anterior



- F = {S}
  - testa se S é o estado objetivo, expande-o e guarda seus filhos A, B e C ordenadamente na fronteira
- F = {A, B, C}
  - testa A, expande-o e guarda seu filho GA ordenadamente
    - obs.: o algoritmo de geração e teste guarda na fronteira todos os nós gerados, testando se um nó é o objetivo apenas quando ele é retirado da lista!
- F= {B, GA, C}
  - testa B, expande-o e guarda seu filho GB ordenadamente
- F= {GB, GA, C}
  - testa GB e para!

# Busca de Custo Uniforme



- Esta estratégia é completa
  - sempre encontra uma solução
- É ótima se
  - custo de caminho no mesmo caminho não decresce
  - i.e., não tem operadores com custo negativo
- Custo de tempo e de memória
  - teoricamente, igual ao da Busca em Largura

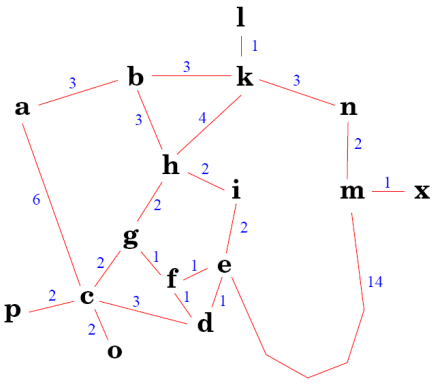
# Busca em Profundidade

- Ordem de ramificação dos nós:
  - sempre expande o nó no nível mais profundo da árvore:
  - 1. nó raiz
  - 2. primeiro nó de profundidade 1
  - 3. primeiro nó de profundidade 2, etc.
  - Quando um nó final não é solução, o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço de estados (backtracking)

# Busca em Profundidade

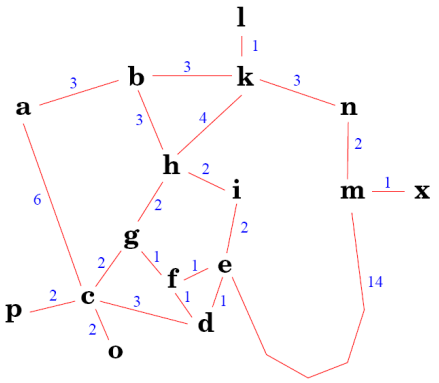


```
function BP(Estado inicial, int m): Nodo
    Stack fronteira
    fronteira.add(new Nodo(inicial))
    while not fronteira.isEmpty() do
        Nodo n ← fronteira.remove()
        if n.getEstado().éMeta() then
            return n
        end if
        if n.getProfundidade() < m then
            fronteira.add(n.sucessores())
        end if
    end while
return null
```



# Busca em Profundidade

53



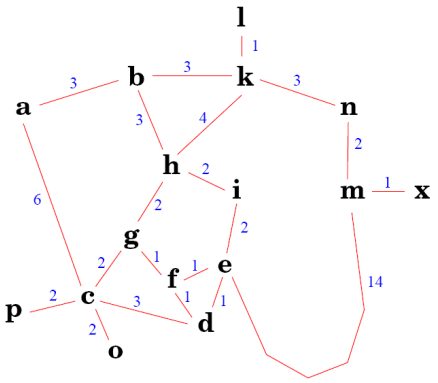


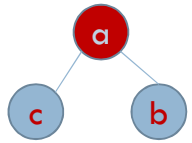
estado inicial: a
estado objetivo: i

~~a~~

# Busca em Profundidade

54



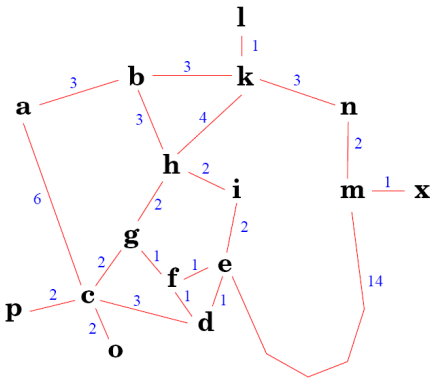



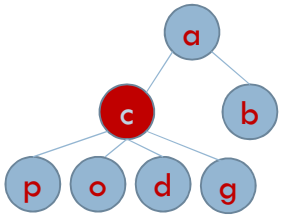

estado inicial: a
estado objetivo: i

c
b
~~a~~

# Busca em Profundidade

55



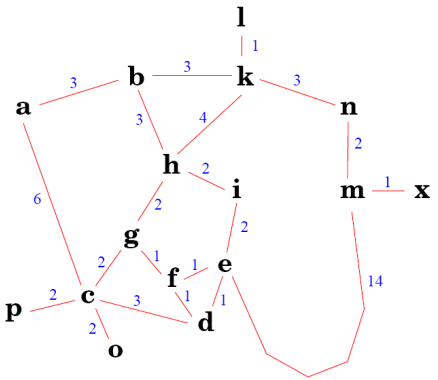



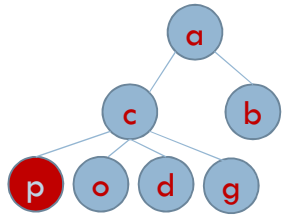

estado inicial: a
estado objetivo: i

p
o
d
g
~~c~~
b
~~a~~

# Busca em Profundidade

56



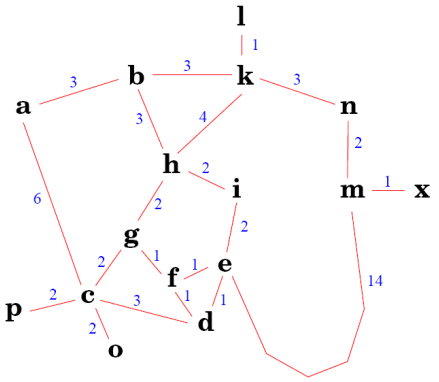



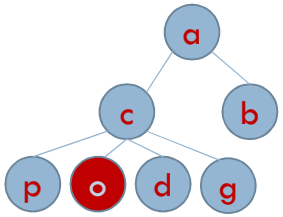

estado inicial: a
estado objetivo: i

~~p~~
o
d
g
~~c~~
b
~~a~~

# Busca em Profundidade

57



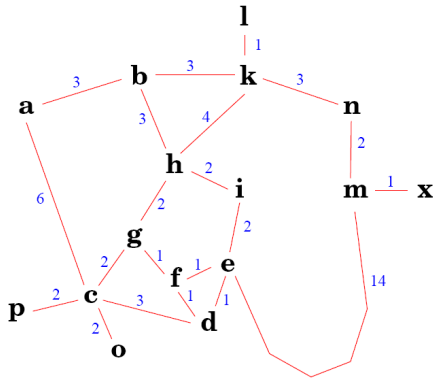



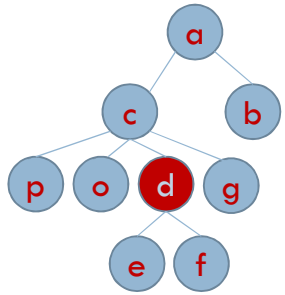

~~p~~ ~~o~~ d g ~~e~~ b ~~a~~

estado inicial: a
estado objetivo: i

# Busca em Profundidade

58



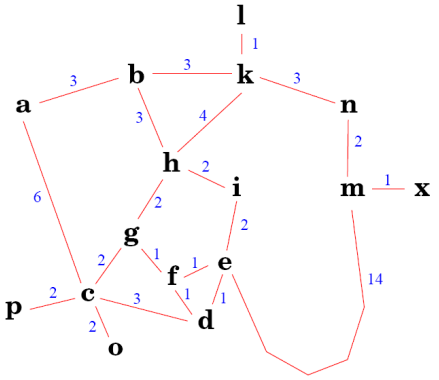



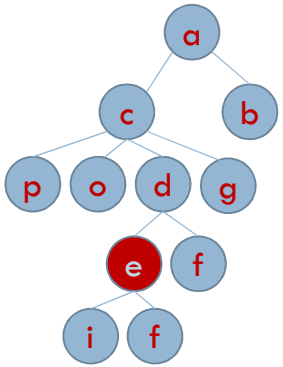

e f ~~p~~ ~~o~~ ~~d~~ g ~~e~~ b ~~a~~

estado inicial: a
estado objetivo: i

# Busca em Profundidade

59



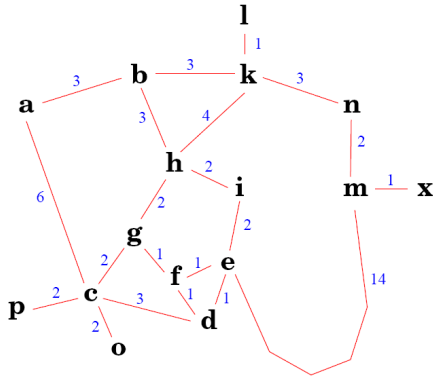



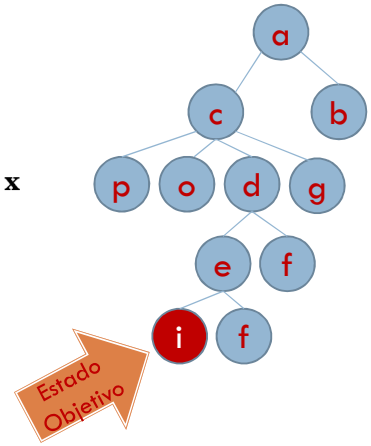

i f ~~e~~ f ~~p~~ ~~o~~ ~~d~~ g ~~e~~ b ~~a~~

estado inicial: a
estado objetivo: i

# Busca em Profundidade

60




i f ~~e~~ f ~~p~~ ~~o~~ ~~d~~ g ~~e~~ b ~~a~~

estado inicial: a
estado objetivo: i

## Busca em Profundidade

- Esta estratégia não é completa nem é ótima
  - Esta estratégia deve ser evitada quando as árvores geradas são muito **profundas** ou geram caminhos infinitos.
- Custo de memória:
  - necessita armazenar apenas b*m nós para um espaço de estados com fator de ramificação b e profundidade m (profundidade máxima de cada nó), onde m pode ser maior que d (profundidade da 1a. solução).
- Custo de tempo:
  - $O(b^m)$, no pior caso.
  - Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.

## Busca com Aprofundamento Iterativo

- Evita o problema de caminhos muito longos ou infinitos impondo um limite máximo (l) de profundidade para os caminhos gerados.
- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - fixa profundidade = i, executa busca
  - se não chegou a um objetivo, recomeça busca com profundidade = i + 1
  - piora o tempo de busca, porém melhora o custo de memória!

## Algoritmo
## Busca com Aprofundamento Iterativo

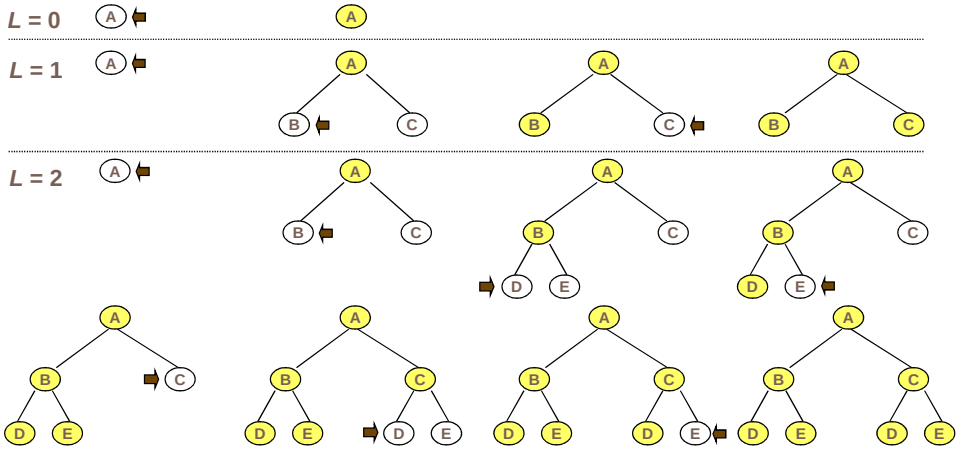


```
function BPI(Estado inicial): Nodo
  int p ← 1
  loop
      Nodo n ←  BP(inicial, p)
      if n <> null then
            return n
      end if
      p ← p + 1
  end loop
```

## Busca com Aprofundamento Iterativo

- Combina as vantagens de busca em largura com busca em profundidade.
  - baixo custo de memória
  - É ótima e completa
    - completo: quando fator de ramificação finito
    - ótimo: custo do caminho é uma função não decrescente da profundidade do nó
- Custo de memória:
  - semelhante a de busca em profundidade
  - necessita armazenar apenas b.d nós para um espaço de estados com fator de ramificação b e limite de profundidade d
- Custo de tempo:
  - $O(b^d)$
- Bons resultados quando o espaço de estados é grande e de profundidade desconhecida.

# Busca com Aprofundamento Iterativo




# Comparando Estratégias de Busca Exaustiva

| **Critério** | Largura | Custo Uniforme | Profun-didade | Aprofun-damento Iterativo |
|---|---|---|---|---|
| Tempo | $b^d$ | $b^d$ | $b^m$ | $b^d$ |
| Espaço | $b^d$ | $b^d$ | bm | bd |
| Otima? | Sim | Sim* | Não | Sim |
| Completa? | Sim | Sim | Não | Sim |

# Exercício em Laboratório



- Construa um programa em java para fazer a busca em largura (de Custo Uniforme) para o problema de rotas.
- Você pode fazer um programa genérico de busca ou um específico para o problema ao lado