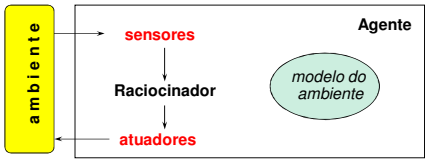
## Resolução de Problemas

**Inteligência Artificial**

## O que é um agente

- **Agente é qualquer entidade que:**
  - Percebe seu ambiente através de sensores (ex. Câmeras, microfone, teclado, ...)
  - Age sobre ele através de atuadores (ex. Vídeo, alto-falante, impressora, braços, ftp, ...)
- **Mapeamento: sequencia perceptiva => ação**




- *Um programa de IA pode ser visto como um **Agente Racional;***
- *Um agente de software recebe entradas do teclado, conteúdo de arquivos e pacotes de rede como sensores de entrada e age no ambiente mostrando resultados na tela, gravando em arquivos e enviando pacotes pela rede;*
- *A função de agente para um agente artificial será implementada por um programa de agente;*
- *É importante distinguir essas duas idéias:*
  - *A função de agente é uma descrição matemática abstrata;*
  - *o programa do agente é uma implementação concreta, rodando na arquitetura do agente.*

## IA – Modelo Simbólico - O que é um agente?

Um agente é tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores.

Um agente humano tem olhos, ouvidos e outros órgãos como sensores, e tem mãos, pernas, boca e outras partes do corpo que servem como atuadores.

Um agente robótico poderia ter câmeras e detectores da faixa de infravermelho funcionando como sensores e vários motores como atuadores.

O dicionário interpreta agente como:
- Alguém que atua;
- Alguém atuando ou fazendo negócios por outro;
- Procurador, delegado.

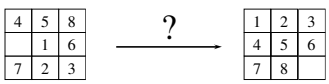
## IA – Modelo Simbólico - O que é um agente?

**Wooldrige & Jennings afirmam que agentes são sistemas que apresentam um comportamento determinado por um processo de raciocínio baseado na representação de suas atitudes, tais como crenças, comprometimentos e desejos.**

## Agente de Resolução de Problemas (1/2)



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




- **Já o agente baseado em objetivo...**
  - sabe, pois segue um **objetivo** explícito

## Agentes de resolução de problemas

- Pesquisadores de planejamento em IA tomam um sistema reativo por ser um sistema capaz de responder rapidamente à mudanças no seu ambiente (onde reativo é tomado como sinônimo de responsivo);
  - Mais recentemente, o termo tem sido usado para denotar sistemas que respondem diretamente ao mundo, no lugar de raciocinar explicitamente sobre ele;
- Sistemas reativos são mais difíceis de projetar que sistemas funcionais.
- Talvez a mais importante razão seja: deve fazer decisões locais continuamente, que tem consequências globais.
- 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.

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

- **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 sequencia de ações que leve a estados desejáveis (objetivos).
- **Questões:**
  - O que é um problema e como formulá-lo?
  - Como buscar a solução do problema?

## 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**

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

***Um problema em IA é definido em termos de...***

**1) Um espaço de estados possíveis, incluindo um estado inicial e um estado final (objetivo)**

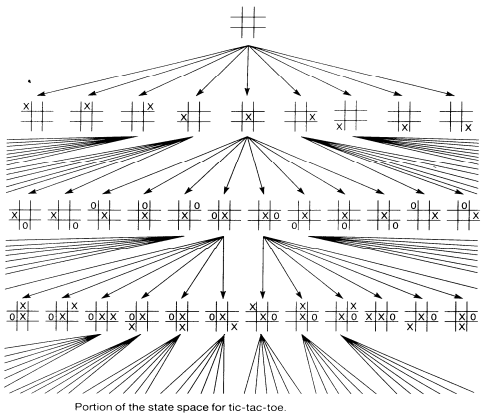
- exemplo 1: dirigir de Natal a Caicó
- exemplo 2: jogo de 8-números


**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*)

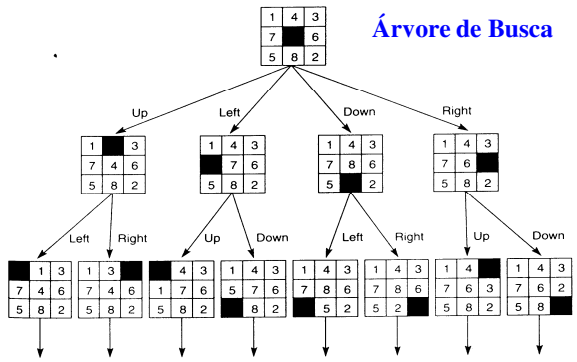
## Problemas e Soluções bem Definidos (2/2)

- **Espaço de Estados:**
  - Conjunto de todos os estados alcançáveis a partir do estado inicial por qualquer sequencia de ações;
- **Definição do objetivo:**
  - Propriedade abstrata
    - *ex., condição de xeque-mate no Xadrez*
  - Conjunto de estados finais do mundo
    - *ex., estar na cidade-destino*
- **Solução:**
  - Caminho (sequencia de *ações* ou *operadores*) que leva do estado inicial, a um estado final (objetivo).



Portion of the state space for tic-tac-toe.

## Árvore de Busca



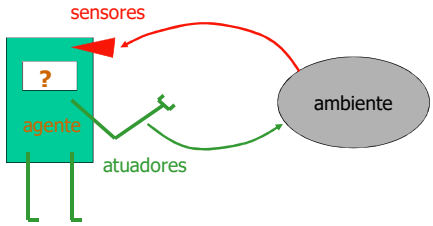

## *Exemplos de Formulação de problema*
## *Jogo de 8 números*

- **Espaço de estados** = todas as possíveis configurações do tabuleiro;
- **Estado inicial** = qualquer um dos estados possíveis;
- **Teste de término** = tabuleiro ordenado, com branco na posição [3,3];
- **Ações/operadores** = mover peças numéricas para espaços livres (em branco) (esquerda, direita, para cima e para baixo);
- **Custo do caminho** = número de passos da solução;
- **Custo de busca** = depende do computador, e da estratégia de busca utilizada.

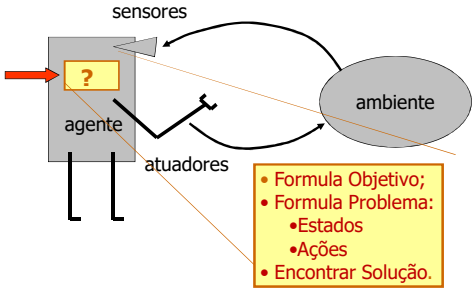
## *Solucionando o Problema:*
## *formulação, busca e execução*

- **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 sequencias de ações para alcançar um objetivo
  - *Solução* = caminho entre estado inicial e estado final.
- **Execução:**
  - Executar (passo a passo) a ***solução completa*** encontrada

## *Agente de Resolução de Problemas*




## *Agente de Resolução de Problemas*



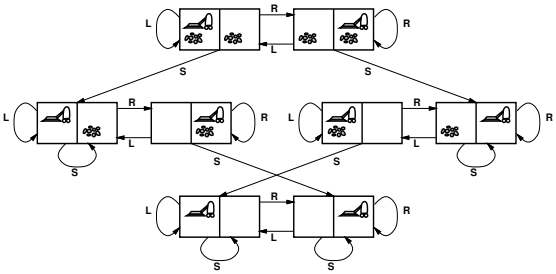

## *Estrutura de um Agente Inteligente*

- **Agente = arquitetura + programa**
- **Programa do Agente: a implementação de $f : \mathcal{P}^* \rightarrow \mathcal{A}$, que mapeia a percepção em ação;**

**função Skeleton-Agente (*Percepção*) retorna *Ação***
**Memória ← Atualiza_Memória (memória, *Percepção*)**
***Ação* ← Escolhe_Melhor_Ação (memória)**
**Memória ← Atualiza_Memória (memória, *Ação*)**
**retorna *Ação***

- **Arquitetura: um dispositivo que pode executar o programa agente (e.g., Um computador genérico, um dispositivo especial, robô, etc.)**

## Aspirador de Pó




## O Mundo do Aspirador de Pó

- **Percebe: o Local e o Estado (ex: [Sala A Suja], [Sala B, Limpa].**
- **Ações: Esquerda, Direita, Aspirar e Fazer Nada.**

## O Agente Aspirador de Pó

- **Função REFLEX ([*Local, Estado*]) retorna *Ação***
  - Se *Estado=Sujo* então retorna ***Aspirar;***
  - Senão, se *Local=A* então retorna ***Direita;***
  - Senão, se *Local=B* então retorna ***Esquerda.***


## Agentes Solucionadores de Problemas *formulação, busca e execução*

22

**função** Agente-Simples-SP(*p*) **retorna** uma *ação*
entrada: *p*, um dado perceptivo

*estado* ← Atualiza-Estado (*estado, p*)
se *s* (sequencia 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*

## Como um Agente se diferencia dos outros softwares? *(1/2)*

- **Os Agentes são autônomos, ou seja, eles agem em benefício do usuário;**
- **Os Agentes contém algum nível de inteligência, de regras fixas a máquinas de aprendizado que possibilitam que eles se adaptem a mudanças no ambiente;**
- **Os Agentes não agem somente reativamente, mas algumas vezes também agem proativamente;**
- **Um bom agente é o que toma as decisões certas, e isto depende da qualidade, acurácia, atualidade e quantidade de informações disponíveis para ele;**

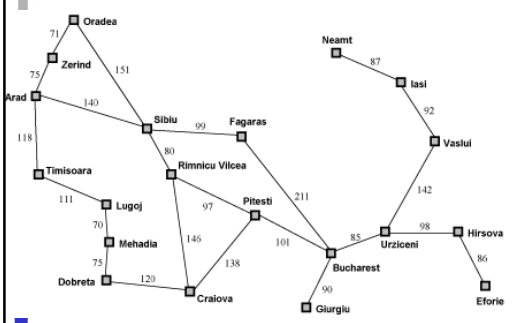
## Como um Agente se diferencia dos outros softwares? *(2/2)*

- **Os Agentes apresentam habilidade social, ou seja, eles se comunicam com o usuário, com o sistema, e outros agentes se necessário;**
- **Os Agentes podem também cooperar com outros agentes para conseguir desempenhar tarefas mais complexas que aquelas que eles poderiam fazer sozinho;**
- **Os Agentes podem migrar de um sistema para outro, para acessar recursos remotos, ou mesmo para encontrar outros agentes.**

## 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 e a solução mais barata

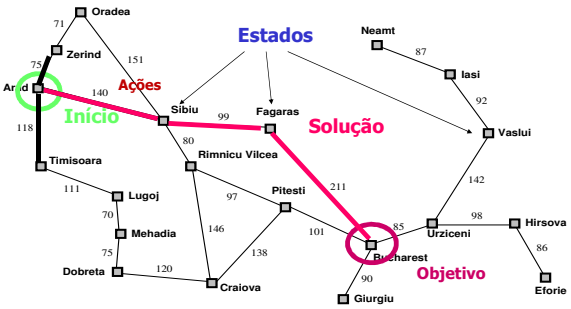
## Outro Exemplo: Ir de Arad a Bucharest




| Straight-line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

## Formulação de Problema

27

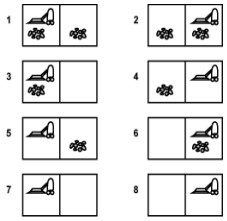



## Exemplo Romênia

- **Ida para Bucharest:**
  - **Estados** = Cada possível cidade do mapa
  - **Estado inicial** = Arad
  - **Teste de término** = Estar em Bucarest
  - **Operadores** = Dirigir de uma cidade para outra
  - **Custo do caminho** = Número de cidades visitadas, distância percorrida, tempo de viagem, grau de divertimento, etc

## Mais um Exemplo...

- **Aspirador de pó**
  - **estados** =
  - **estado inicial** =
  - **teste de término** =
  - **operadores** =
  - **custo da solução** =

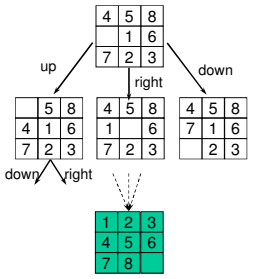



## Custo Diferente => Solução Diferente

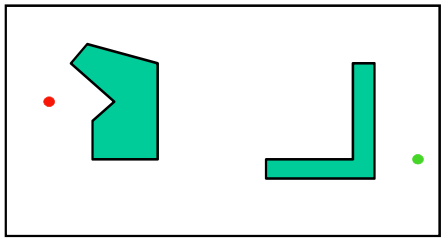
- **Função de *custo de caminho***
  - (1) número de cidades visitadas;
  - (2) distância entre as cidades;
  - (3) tempo de viagem, etc.
- **Solução mais barata:**
  - (1) Canudos, Belém do S. Francisco, Salgueiro, ...
  - (2) Canudos, Belém do S. Francisco, Salgueiro, ...
  - (3) Canudos, Juazeiro, Petrolina, Cabrobó, Salgueiro

## 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

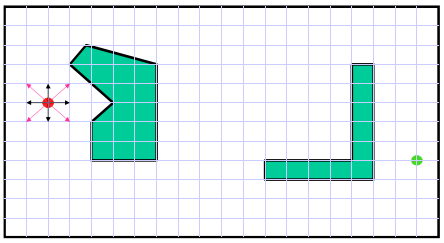



## Exemplo: Navegação de robô



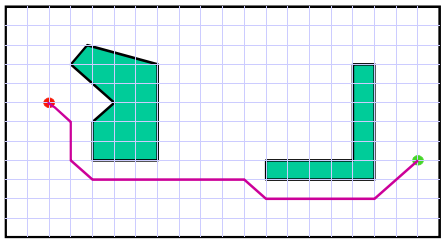
Qual é o espaço de estados?

## Exemplo: navegação de robô

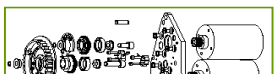


Custo de um passo horizontal/vertical = 1
Custo de um passo diagonal = $\sqrt{2}$

## Exemplo: navegação de robô



## Exemplo: Planejamento de Montagem

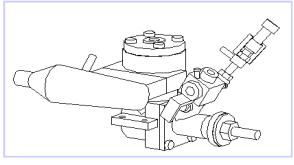


Estado Inicial

Função Complexa :
Deve encontrar a collision-free merging motion existe

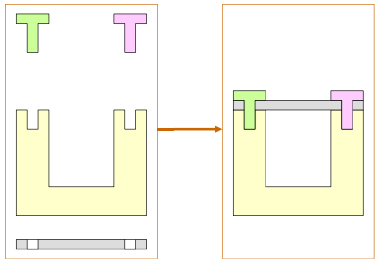
Função Sucessora :
• Mesclar duas sub montagens

Objetivo



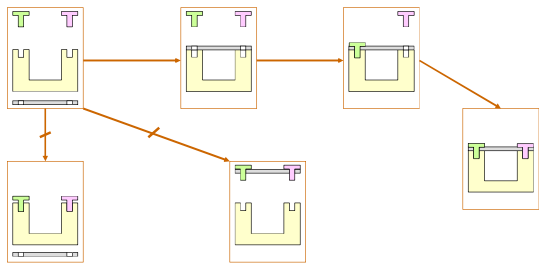
## Exemplo: Planejamento de Montagem





## Exemplo: Planejamento de Montagem

37



## Algumas Aplicações

38

## Aplicações de Busca: "Toy Problems"

39

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

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

- **Torre de Hanói**
- **Palavras cruzadas**
- **Canibais e missionários**

## Aplicações: Problemas Reais

40

- **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 Circuitos Integrados VLSI**
  - Cell layout;
  - Channel routing.

## Aplicações: Problemas Reais

- **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
- **etc...**

## Problemas com Informação Parcial

43

## Problemas com informação Parcial

- **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 sequencia qualquer de ações*
  - Solução: sequencia de ações.
- **Porém existem 3 outros tipos de problemas...**

44

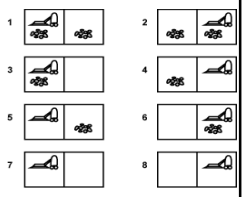
## Problemas com Informação Parcial

- **Sensorless ou conformant problem:**
  - Agente não sabe seu estado inicial (percepção deficiente);
  - Deve raciocinar sobre os conjuntos de estados;
  - Solução: sequencia de ações (via busca).
- **Problema de contingência:**
  - Efeito das ações não-determinísticas e/ou mundo parcialmente observável => novas percepções depois de ação
    - *ex. aspirador que suja ao aspirar 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.

45

## Problemas com Informação Parcial

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




## Buscando Soluções

**Busca Cega ou Não Informada**

## Resolução de Problemas

- **É necessário conhecimento sobre o problema e técnicas de manipular este conhecimento para obter a solução.**
- **Os agentes de Resolução de Problemas decidem o que fazer encontrando sequências de *ações* que levam a *estados* desejáveis.**

## Formulando Problemas

- O que é um **PROBLEMA**?
  - Resolver um problema é diferente de ter um método para resolvê-lo;
  - Antes de tentar buscar a solução de um problema, deve-se responder as seguintes perguntas:
    - *Quais são os dados?*
    - *Quais são as soluções possíveis?*
    - *O que caracteriza uma solução satisfatória?*

## Resolução de Problemas

- **Formulação de objetivo, baseada na situação atual é o primeiro passo na resolução de problemas.**
- **Além de formular um objetivo, pode-se decidir sobre os fatores que afetam a desejabilidade de diferentes caminhos que podem alcançar o objetivo.**

## 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 levará 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).

## Selecionando um espaço de estados

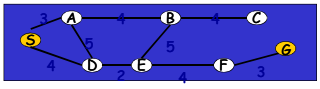
- **O mundo real é absurdamente complexo**
  → o espaço de estados é uma abstração
- **Estado (abstrato) = conjunto de estados reais**
- **Ação (abstrata) = combinação complexa de ações reais**
  - ex., "SP → Fortaleza" representa um conjunto complexo de rotas, desvios, paradas, etc.
  - Qualquer estado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

## Busca de soluções

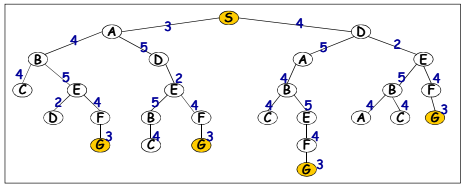
- **Idéia: Percorrer o espaço de estados a partir de uma *árvore de busca*.**
- ***Expandir* o estado atual aplicando a função sucessor, *gerando* novos estados.**
- **Busca: seguir um caminho, deixando os outros para depois.**
- **A *estratégia de busca* determina qual caminho seguir.**

## Do espaço do problema para árvore de busca

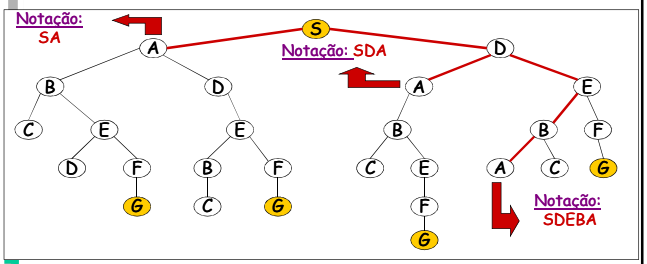
**Espaço do problema**



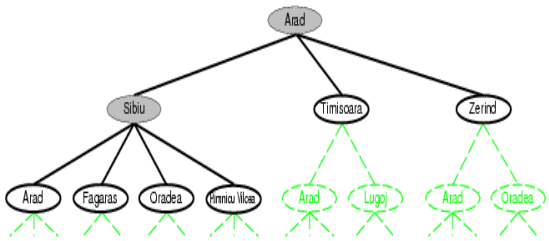

Árvore de busca associada (sem ciclos)



## Caminhos em árvores de busca

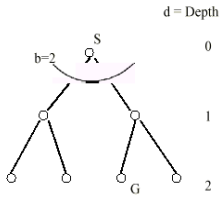


## Árvore de Busca: Exemplo




## Notação em árvores de pesquisa

- *Fator de ramificação -b (branching factor)*
- *Profundidade do nó - d*
- *caminhos parciais*
- *caminhos que não terminam num nó objetivo*
- *Caminhos completos*
- *caminhos que atingem a meta*
- *Nós abertos*
- *Nós que não são expandidos ( ex: folhas)*
- *Nós fechados*
- *Nós que já foram expandidos*



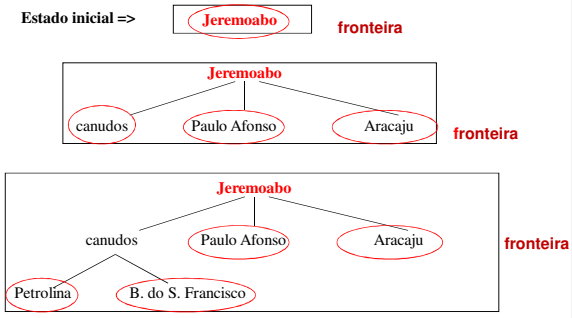

## Busca em Espaço de Estados: Geração e Teste

- ***Fronteira* do espaço de estados**
  - Os 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




## Busca em Espaço de Estados: Implementação

- **Espaços de Estados**
  - Podem ser representados como uma árvore onde os estados são nós e as operações são arcos.
- **Os nós da árvore podem guardar mais informação do que apenas o estado: estrutura de dados com pelo menos 5 componentes:**
  1. O estado correspondente
  2. O seu nó pai
  3. O operador aplicado para gerar o nó (a partir do pai)
  4. A profundidade do nó
  5. O custo do nó (desde a raiz)
  6. Nós-filhos
  7. Etc.

## 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

## *Definição de Problemas*

➢ ***Performance* da Resolução:**

**A eficiência de uma busca pode ser medida de três formas:**

- ➢Se ela encontra uma solução;
- ➢Se a solução encontrada é uma boa solução;
- ➢O custo total da busca, que é o custo do caminho mais o custo da busca (tempo e memória requeridos para encontrar uma solução).

## *Métodos de Busca*

- **Busca Exaustiva ou Cega:**
  - Não utilizam qualquer conhecimento específico do problema para determinar a prioridade com que os nós serão expandidos, por isso são chamados de busca cega;
  - Não sabe qual o **melhor** nó da fronteira a ser expandido = ou seja, desconhece o 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 ou Blind Search*

- **Estratégias para determinar a ordem de ramificação dos nós:**
  1. Busca em largura (ou Breadt First Search - BFS)
  2. Busca de custo uniforme (ou Uniform Cost Search – UCS)
  3. Busca em profundidade (ou Depth First Search - DFS)
  4. Busca com aprofundamento iterativo (Iterative Deepening Search - IDS)
- **Direção da ramificação:**
  1. Do estado inicial para um estado final
  2. De um estado final para o estado inicial
  3. Busca bi-direcional

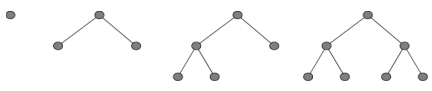
## *Critérios de Avaliação das Estratégias de Busca*

- **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 ou em Extensão*

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

**função** Busca-em-Largura (*problema*)
**retorna** uma solução ou falha
Busca-Genérica (*problema*, Insere-no-Fim)

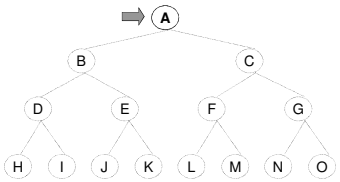


## *Busca em Largura*

- **Esta estratégia é *completa* (sempre encontra uma solução)**
- **É *ótima* ?**
  - Sempre encontra a solução mais "rasa"
  - Mas 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 somente por E*
- **Em outras palavras, é *ótima* se** custo de caminho **cresce com a profundidade do nó**
  - O que ocorre quando todos os operadores têm o mesmo custo (=1)

## Busca em Largura (ou Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no final da fila.

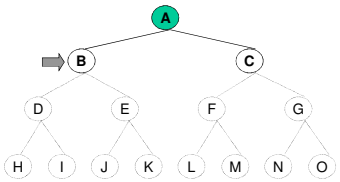



**Fronteira = (A)**



## Busca em Largura (Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no final da fila

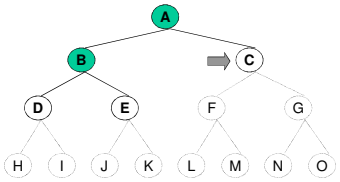



**Fronteira = (B,C)**



## Busca em Largura (Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., os novos sucessores são postos no final da fila

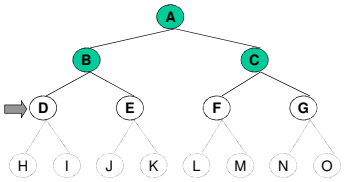



**Fronteira = (C,D,E)**



## Busca em Largura (Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no fim

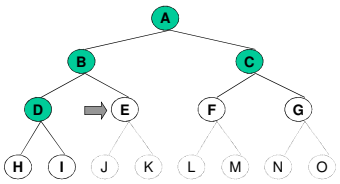



**Fronteira = (D,E,F,G)**



## Busca em Largura (Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no fim

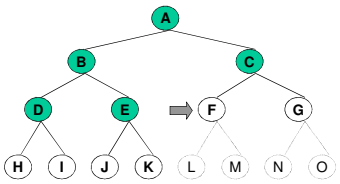



**Fronteira = (E,F,G,H,I)**



## Busca em Largura (Breadth-first)

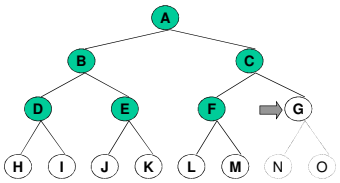
- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no fim




**Fronteira = (F,G,H,I,J,K)**

## Busca em Largura (Breadth-first)

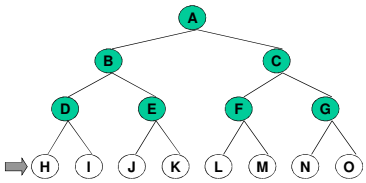
- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no fim




**Fronteira = (G,H,I,J,K,L,M)**

## Busca em Largura (Breadth-first)

- **Expande sempre o nó menos profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo FIFO**, i.e., novos sucessores são postos no fim



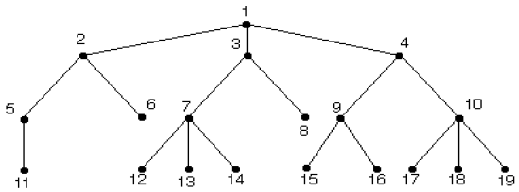

**Fronteira = (H,I,J,K,L,M,N,O)**

## Busca Breadth-First *(cont)*

| 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 |

- **Requisitos de tempo e memória para uma busca Breadth-First.**
- **Os valores acima apresentados assumem: (1) branching factor b=10; (2) 1000 nó/segundo; (3) 100 bytes/nó**

## Busca Breadth-First *(cont)*

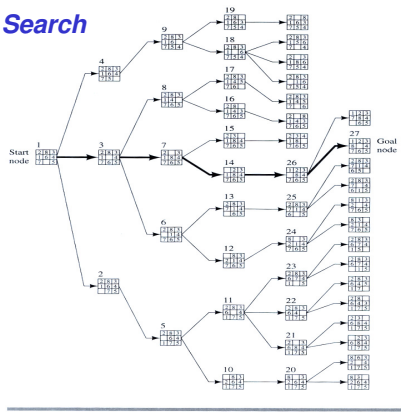



Árvore de Busca Breadth-First
(Números referem-se a ordem de visita na Busca)

## Busca em Largura ou busca em extensão

- **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:**
  - Problema mais crucial: a *fronteira* do espaço de estados deve permanecer em memória (RAM);
  - Logo, busca em largura somente apresenta bons resultados quando a profundidade da árvore de busca é pequena.

## Breadth-First Search



## Busca de Custo Uniforme (Dijkstra's Search)

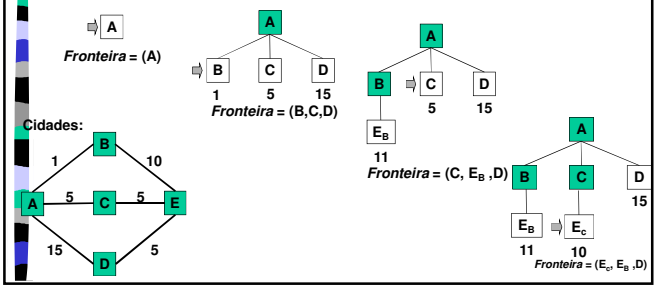
- **Concebido pelo cientista da computação Edsger W. Dijkstra em 1956;**
- **Estende a busca em largura:**
  - Expande o nó da fronteira com menor custo de caminho até o momento;
  - Cada operador pode ter um custo associado diferente, medido pela **função *g(n)*** que dá o **custo do caminho da origem ao nó n.**
- **Na busca em largura:** *g(n) = profundidade (n)*
- **Algoritmo:**

**função** Busca-de-Custo-Uniforme (*problema*)
  **retorna** uma solução ou falha

Busca-Genérica (*problema*, Insere-Ordem-Crescente)

## Busca de Custo Uniforme

- **Expande sempre o próximo nó ainda não expandido que possui caminho de menor custo**
  - ***Fronteira*** = **fila de nós ordenada pelo custo do caminho até cada nó**




Cidades:




## 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 $G_A$ 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, $G_A$, C}**
  - testa B, expande-o e guarda seu filho $G_B$ ordenadamente
- **F= {$G_B$, $G_A$, C}**
  - testa $G_B$ e para!

## Busca de Custo Uniforme

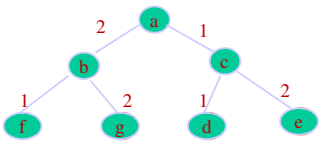
- **Esta estratégia é *completa!***
- **É *ótima* se**
  - g (sucessor(n)) ≥ g (n)
    - *custo de caminho* ***no mesmo caminho*** *não decresce*
    - *i.e., não tem operadores com* ***custo negativo***
  - caso contrário, teríamos que expandir todo o espaço de estados em busca da melhor solução.
    - *Ex. Seria necessário expandir também o nó C do exemplo, pois o próximo operador poderia ter custo associado = -13, por exemplo, gerando um caminho mais barato do que através de B*
- **Custo de tempo e de memória**
  - teoricamente, igual ao da Busca em Largura

**Se o custo dos nós de um mesmo nível for igual, o desempenho é equivalente ao da Busca em Largura**

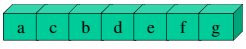
## Custo Uniforme - Complexidade

| Profundidade | Nós | Tempo | Memória |
|---|---|---|---|
| 0 | 1 | 1 milisegundo | 100 bytes |
| 2 | 111 | 0.1 segundos | 11 kilobytes |
| 4 | 11.111 | 11 segundos | 1 megabytes |
| 6 | $10^6$ | 18 minutos | 111 megabytes |
| 8 | $10^8$ | 31 horas | 11 gigabytes |
| 10 | $10^{10}$ | 128 dias | 1 terabyte |
| 12 | $10^{12}$ | 35 anos | 111 terabytes |
| 14 | $10^{14}$ | 3500 anos | 11.111 terabytes |

## Exemplo de UCF: Uniform Cost Search




***Resposta:***

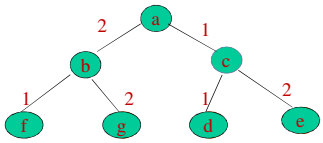


## Busca de Custo Uniforme

- Seja g(n) a soma dos custos das arestas da raiz ao nó n.
- Se g(n) for a função de custo geral, a melhor pesquisa se tornará a Pesquisa de Custo Uniforme, também conhecida como algoritmo de caminho mais curto de Dijkstra;
- Inicialmente, o nó raiz é colocado em aberto com um custo zero.
- A cada etapa, o próximo nó n a ser expandido é um nó aberto cujo custo g(n) é o mais baixo entre todos os nós abertos.

## Exemplo de Busca de Custo Uniforme

- **Suponha por exemplo uma árvore com diferentes custos de arestas, representada pelos números próximos às bordas**

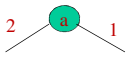



**Notações deste exemplo:**

**Nós gerados**

**Nós expandidos**

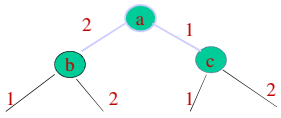
## Exemplo de Busca de Custo Uniforme




**Nós gerados**:

*Open list:*

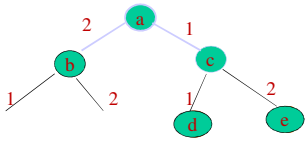

## Exemplo de Busca de Custo Uniforme




**Nós gerados** :

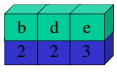

*Open list:*


## Exemplo de Busca de Custo Uniforme

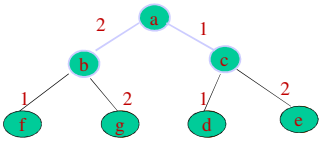



**Nós gerados**

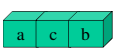

*Open list:*



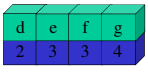

## Exemplo de Busca de Custo Uniforme



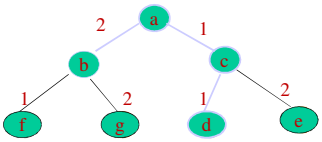

**Nós gerados**:




*Open list:*



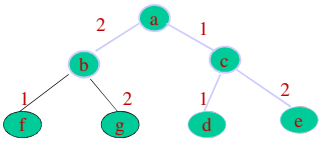
## Exemplo de Busca de Custo Uniforme




**Nós gerados**: a c b d

*Open list:* e f g / 3 3 4

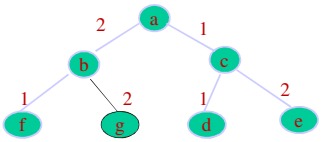
## Exemplo de Busca de Custo Uniforme




**Nós gerados**: a c b d e

*Open list:* f g / 3 4

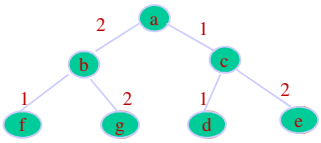
## Exemplo de Busca de Custo Uniforme




**Nós gerados**: a c b d e f

*Open list:* g / 4

## Exemplo de Busca de Custo Uniforme




**Nós gerados** a c b d e f g

*Lista aberta:*

## Busca de Custo Uniforme

- Considera-se a busca de custo uniforme como uma pesquisa de força bruta, porque não usa uma função heurística.
- **Questões levantadas:**
  - O custo uniforme sempre termina ?
  - Garante-se sempre encontrar o estado objetivo?

## Busca de Custo Uniforme- Finalização

- O algoritmo encontrará um nó objetivo, ou apresentará mensagem de que não há nó de meta nas seguintes condições:
  - O espaço de estado do problema é finito;
  - Deve existir um caminho para um objetivo cujo comprimento seja finito e de custo finito;
  - Não deve haver caminhos infinitamente longos de custo finito.
- Será assumido que todas as arestas têm um custo mínimo diferente de zero e para resolver um problema de rotas infinitas de nós com arestas de custo zero.

  **Então, o UCS acabará atingindo um objetivo de custo finito, se houver algum.**.

## Busca em Profundidade

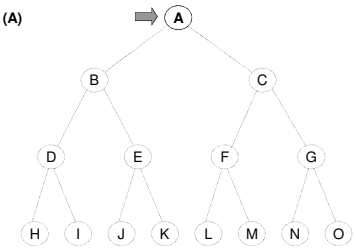
- **Ordem de ramificação dos nós:**
  - Sempre expande o nó no *nível mais profundo* da árvore:
    1. nó raiz
    2. primeiro nó de profundidade 1
    3. primeiro nó de profundidade 2, etc.
  - Quando um nó final não é solução, o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço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

**função** Busca-em-Profundidade (*problema*)
  **retorna** uma solução ou falha
  Busca-Genérica (*problema*, Insere-no-Começo)

## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

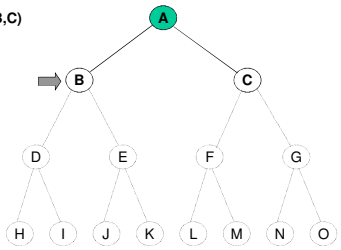
***Fronteira* = (A)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

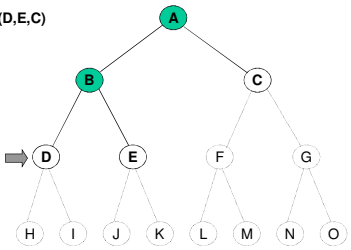
***Fronteira* = (B,C)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

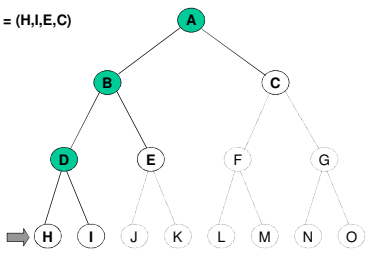
***Fronteira* = (D,E,C)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

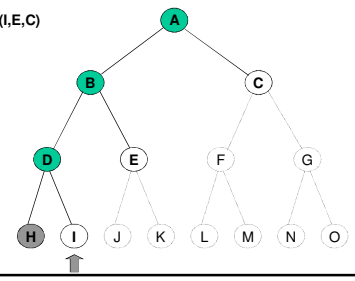
***Fronteira* = (H,I,E,C)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

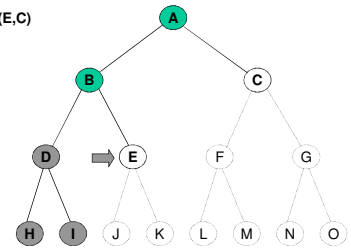
***Fronteira* = (I,E,C)**



## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

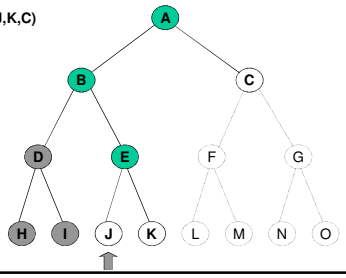
***Fronteira* = (E,C)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

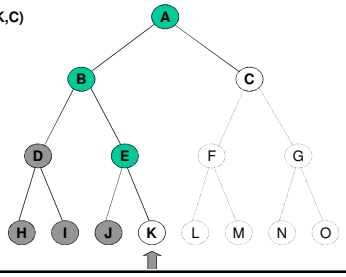
***Fronteira* = (J,K,C)**




## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

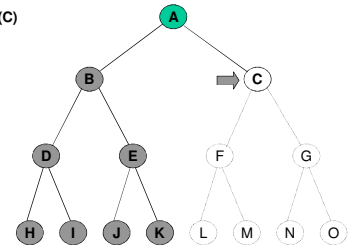
***Fronteira* = (K,C)**



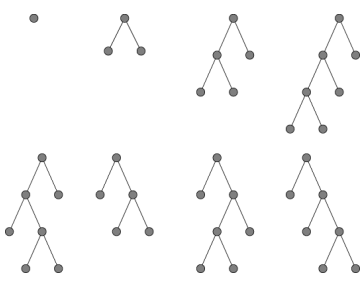

## Busca em Profundidade (Depth-first)

- **Expande sempre o nó mais profundo ainda não expandido;**
  - ***Fronteira*** é uma **fila do tipo LIFO**, i.e., novos sucessores são postos no início

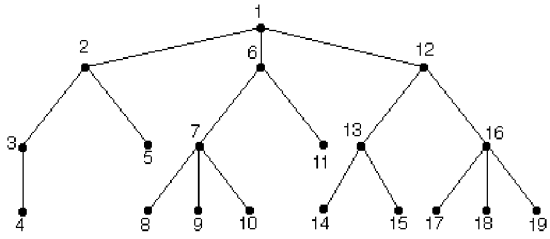
***Fronteira* = (C)**




## Busca em Profundidade



## Busca em profundidade




Árvore de busca - Dept-First
(Os números referem-se a ordem de visitação na Busca)

## 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, onde m pode ser maior que d (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 em Profundidade Limitada

- Impõe uma profundidade máxima para a expansão dos nós;
- Encontra a solução se esta estiver em uma profundidade menor ou igual ao limite estabelecido;
- Complexidade de tempo exponencial;
- Complexidade de espaço polinomial.

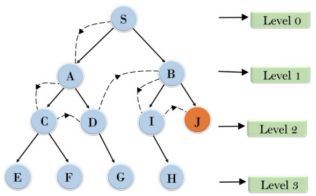
## Propriedades da Busca em Profundidade

- 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*.**
  - Em casos que temos 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**

**Obs:** b→ fator de ramificação, *m* → profundidade máxima da árvore de busca

## Propriedades da Busca em Profundidade Limitada

- Um algoritmo de busca limitada em profundidade é semelhante à busca em profundidade com um limite predeterminado;
- A pesquisa limitada em profundidade pode resolver a desvantagem do caminho infinito na pesquisa em profundidade;
- Neste algoritmo, o nó no limite de profundidade será tratado, pois não possui mais nós sucessores.




## Propriedades da Busca em Profundidade Limitada

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

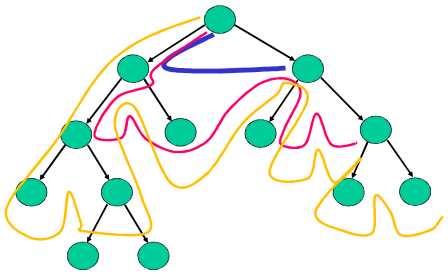
- **Obs:** b→ fator de ramificação, *l* → limite de profundidade

## Busca em Profundidade Iterativa

- **Aumenta o limite de profundidade a cada iteração;**
- **Encontra solução ótima;**
- **Complexidade de tempo exponencial;**
- **Complexidade de espaço polinomial**

## Aprofundamento Iterativo

115



## Busca com Aprofundamento Iterativo

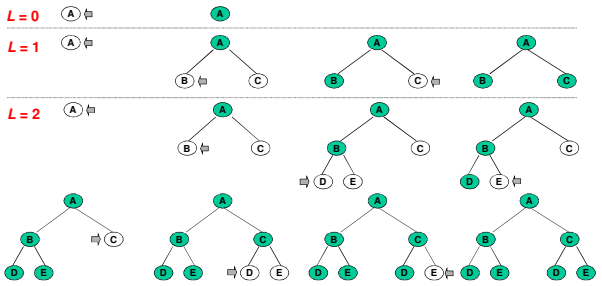
- Evita o problema de caminhos muito longos ou infinitos impondo um **limite máximo *(l)*** de profundidade para os caminhos gerados.
  - $l \geq d$, onde $l$ é o limite de profundidade e $d$ é a profundidade da primeira solução do problema
- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - Fixa profundidade = *i*, executa busca
  - Se não chegou a um objetivo, recomeça busca com profundidade = *i* + *n* (*n* qualquer)
  - Piora o tempo de busca, porém melhora o custo de memória!

## Busca com Aprofundamento Iterativo

- **Combina as vantagens de *busca em largura* com *busca em profundidade.***
- **É *ótima* e *completa***
  - Com n = 1 e operadores com custos iguais
- **Custo de memória:**
  - 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 de Aprofundamento Iterativo

- **Variação da Busca em Profundidade, que utiliza um limite de profundidade *L*, que inicia em 0 e vai sendo incrementado de 1, a cada iteração.**

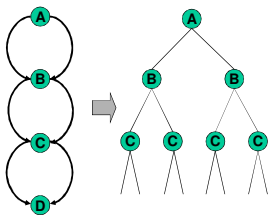



## Comparando Estratégias de Busca Exaustiva

| 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 |

- **Onde:**
  ***b*** (*branching*) - fator máximo de ramificação da árvore de busca;
  ***d*** (*depth*) – profundidade da solução de menor custo;
  ***m*** – profundidade máxima do espaço de estados (que pode ser infinita)

## Repetição de Estados na Busca



**Soluções possíveis (Custo x Eficácia)**

1. **Não retornar ao estado "pai"**
2. **Não retornar 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 $O(b^d)$
   - pode ser implementado mais eficientemente com *hash tables*
   - quando encontra nó igual, tem de escolher o melhor (menor custo de caminho até então)

## 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.
- **3 soluções com diferentes níveis de eficácia e custo de implementação...**

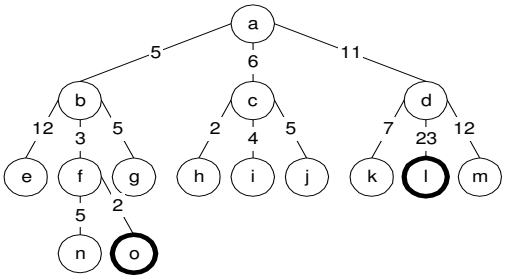
## Evitar Estados Repetidos: soluções

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 $O(b^d)$
   - pode ser implementado mais eficientemente com *hash tables*
   - quando encontra nó igual tem de escolher o melhor (menor custo de caminho até então)

## Exercício - Busca cega

- **Considerando a árvore de pesquisa seguinte, encontrar o(s) nó(s) objetivo(s) - (L e/ou O) aplicando os métodos de busca cega!**

## Exercício: busca cega




## Exercício: busca cega
## Solução

- ***Busca em Largura***
- **Procurar na árvore por níveis. Parar somente quando for achado o primeiro nó solução.**
  - *Completo e ótimo, se o custo de um caminho for uma função não decrescente do nível do nó.*

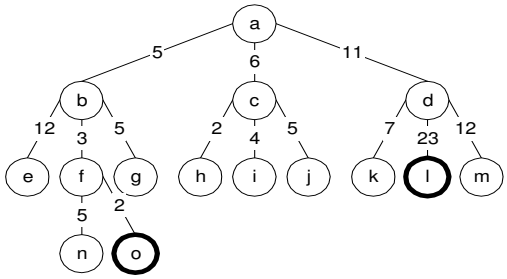
## Exercício: busca em largura
## Solução

- expansão

{A}
A {B C D}
B {C D E F G}
C {D E F G H I J}
D {E F G H I J K L M}
E {F G H I J K L M}
F {G H I J K L M N O}
G {H I J K L M N O}
H {I J K L M N O}
I {J K L M N O}
J {K L M N O}
K {L M N O}
L {M N O} pára

Breadth-First Search: A B C D E F G H I J K L
Solução encontrada: A D L
Nós expandidos: 12

## Exercício 4.2
## Busca cega




## Exercícios busca cega
## Solução

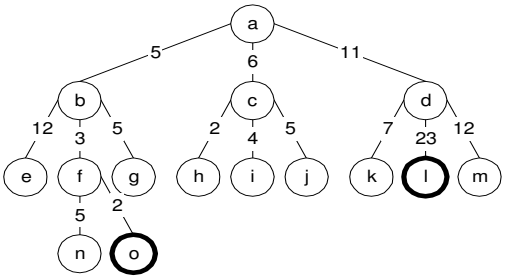
- ***Procura em Profundidade:***
  - Expandir um nó da árvore, o mais profundo possível;
  - Parar quando achar o primeiro nó solução.
    - *Completo mas não ótimo*

## Exercício: busca cega
## Solução

- **expansão**

**{ A }**
**A { B C D }**
**B {E F G C D}**
**E { F G C D}**
**F { N O G C D}**
**N { O G C D }**
**O { G C D }**
**Deap-First Search: A B E F N O**
**Solução encontrada: A B F O**
**Nós expandidos: 6**

## Exercício: busca cega




## Exercício: busca cega
## Solução

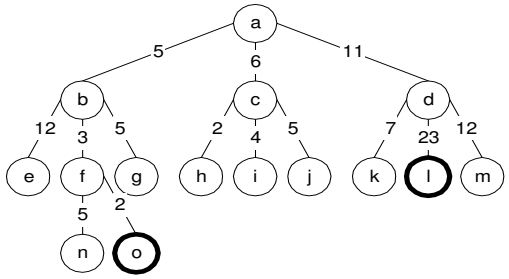
- ***Procura por Custo Uniforme***
  - Consiste em visitar o nó da lista cujo custo é o menor.
    - *Completo e ótimo*
- **Procura - a, b, c, f, h, g, i, o**
  - **Expansão -** ~~b=~~5, ~~e=6~~, d=11, e=17, ~~f=8~~, ~~h=8~~, ~~g=10~~, ~~i=10~~, j=11, n=13, ~~o=10~~

## Exercícios pesquisa cega
## Solução (Custo Uniforme)

- **expansão**

**{ A }**
**A { B (5) C (6) D( 11 }**
**B { C (6) F (8) G (10) D(11) E(17)}**
**C { F (8) H(8) G(10) I(10) D(11) J(11) E(17) }**
**F { H (8) G(10) I(10) O(10) D(11) J(11) F(13) E(17) }**
**H {G(10) I(10) O(10) D(11) J(11) F(13) E(17) }**
**G { I(10) O(10) D(11) J(11) F(13) E(17) }**
**I { O(10) D(11) J(!!) F(13) E(17) }**
**O { D(11) J(11) F(13) E(17) }**
**Pára, porque o caminho por D é maior que o nó objetivo já encontrado**
**Uniform Search: A B C F H G I O**
**Solução encontrada:A B F O Nós expandidos: 8**

## Exercício: busca cega

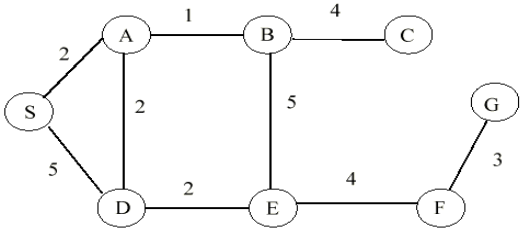



## Exercício: busca cega
## Solução

- ***Procura por Aprofundamento Iterativo***
  - Consiste em expandir a arvore um nível de cada vez
    - *Completo e ótimo, se o custo de um caminho for uma função não decrescente do nível do nó.*
  - **Procura**
    - *1 - a*
    - *2 - a b c d*
    - *3 - a b c d e f g h i j k l*

## Exercício 2

- *Dado o seguinte espaço de estados, construa a árvore de pesquisa e a expansão que atinge a meta G, pelo método de custo uniforme*




## Exercício 3

- Dado o grafo seguinte, encontrar o nó objetivo (G) aplicando os métodos de busca cega!