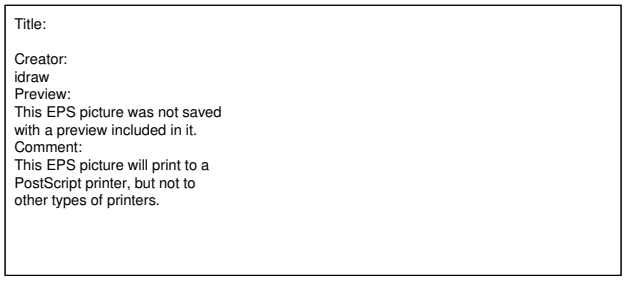
# Agentes Inteligentes

- Agentes Inteligentes - Agentes Racionais
- Estrutura dos Agentes Inteligentes
  - Agentes Simples Reflexos
  - Agentes com Representação do Mundo
  - Agentes Baseado em Objectivos
  - Agentes Baseados em Utilidade
- Propriedades dos Ambientes

# Agentes Inteligentes

- Agente: Apercebe-se do ambiente através de sensores e age nesse ambiente através de actuadores
- Sensores: Olhos, ouvidos, nariz, tacto, gosto, outros
- Actuadores: Pernas, braços, mãos, outros
- Agente robótico: cameras, sonares, sensores de infra-vermelhos, motores, rodas, etc.



## Agentes Inteligentes - Agentes Racionais

- Agente Racional é aquele que faz a acção correcta!
- Qual a acção correcta?
  - Aquela que o faz ser mais bem sucedido!
- Como e quando avaliar esse sucesso? (medida do sucesso)
- Exemplo: Agente aspirador!
- Agente Racional Ideal: “Para cada sequencia de percepções, faz a acção que é esperado maximizar a sua medida de performance (sucesso), dada o conhecimento que ele tem!”
- Mapeamento entre percepções e acções!

## Estrutura dos Agentes Inteligentes

- Agente exibe um comportamento - acção que é executada depois de uma dada sequência de percepções!
- Tarefa da IA:
  - Projectar o Programa e a Arquitectura para o Agente
- O que é um Agente?
  - Agente = Arquitectura + Programa
- Agentes de Software vs Agentes Físicos

## Estrutura dos Agentes - Descrição PAGE

| Agent Type | Percepts | Actions | Goals | Environment |
|---|---|---|---|---|
| Medical diagnosis system | Symptoms, findings, patient's answers | Questions, tests, treatments | Healthy patient, minimize costs | Patient, hospital |
| Satellite image analysis system | Pixels of varying intensity, color | Print a categorization of scene | Correct categorization | Images from orbiting satellite |
| Part-picking robot | Pixels of varying intensity | Pick up parts and sort into bins | Place parts in correct bins | Conveyor belt with parts |
| Refinery controller | Temperature, pressure readings | Open, close valves; adjust temperature | Maximize purity, yield, safety | Refinery |
| Interactive English tutor | Typed words | Print exercises, suggestions, corrections | Maximize student's score on test | Set of students |

## Programa de um Agente / Tipos de Agentes

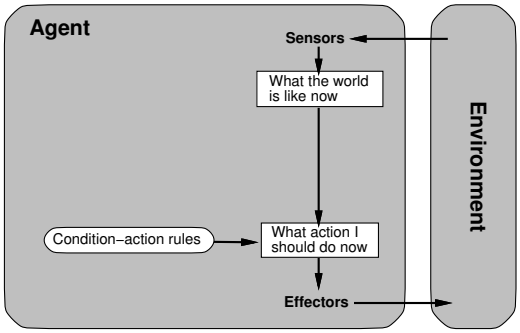
- Estruturas de Dados internas são actualizadas usando percepções e usadas para tomar a decisão das acções a executar (melhor acção)

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent's memory of the world

  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```

- Tipos de Agentes (Russel e Norvig):
  - Agentes reflexos simples
  - Agentes com representação do mundo
  - Agentes baseados em objectivos
  - Agentes baseados em utilidade

# Agentes Simples Reflexos

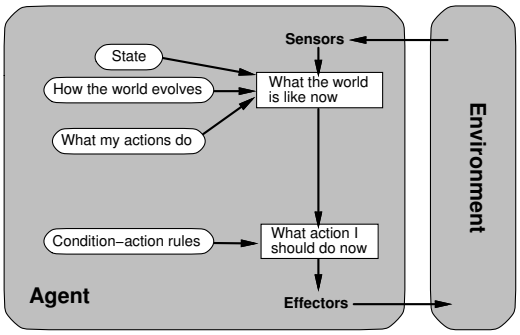
- Baseados em tabelas de regras condição-acção (regras if-then)




```
function SIMPLE-REFLEX-AGENT(percept) returns action
  static: rules, a set of condition-action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

# Agentes com Representação do Mundo

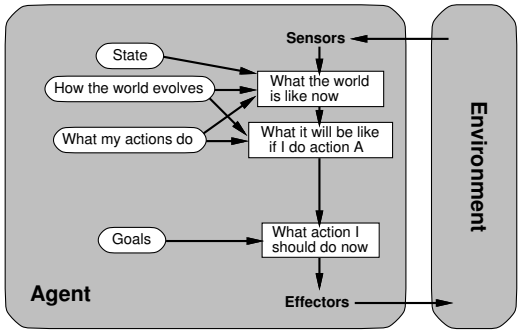
- Mantêm um estado interno (representação do mundo)




```
function REFLEX-AGENT-WITH-STATE(percept) returns action
  static: state, a description of the current world state
          rules, a set of condition-action rules

  state ← UPDATE-STATE(state, percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  state ← UPDATE-STATE(state, action)
  return action
```

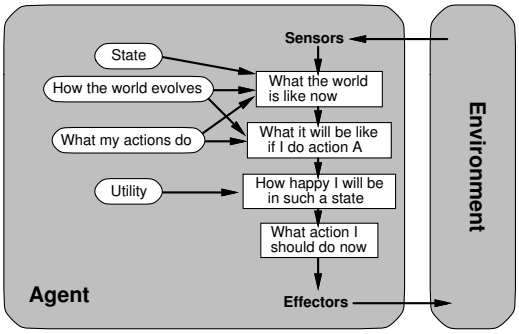
# Agentes Baseado em Objectivos

- Descrição do estado do mundo e do objectivo a atingir
- Exemplo: Chegar a Lisboa
- Resolução de problemas por Pesquisa, Planeamento




# Agentes Baseados em Utilidade

- Utilidade: Espécie de grau de felicidade do agente!
- Mapeia o estado actual num valor!



## Propriedades dos Ambientes

- Acessível vs Inacessível
  - Acessível se os sensores do agente detectam tudo o que é relevante do ambiente!
- Determinístico vs Não Determinístico
  - Determinístico se o próximo estado é determinado pelo anterior e pelas acções do agente!
- Episódico vs Não Episódico
  - Dividido em episódios! Episódios seguintes não dependem de acções em episódios anteriores!
- Estático vs Dinámico
  - Dinâmico se muda enquanto o agente está a pensar!
- Discreto vs Contínuo
  - Discreto se existe um número finito de percepções e acções!

## Propriedades dos Ambientes

| Environment | Accessible | Deterministic | Episodic | Static | Discrete |
|---|---|---|---|---|---|
| Chess with a clock | Yes | Yes | No | Semi | Yes |
| Chess without a clock | Yes | Yes | No | Yes | Yes |
| Poker | No | No | No | Yes | Yes |
| Backgammon | Yes | No | No | Yes | Yes |
| Taxi driving | No | No | No | No | No |
| Medical diagnosis system | No | No | No | No | No |
| Image-analysis system | Yes | Yes | Yes | Semi | No |
| Part-picking robot | No | No | Yes | No | No |
| Refinery controller | No | No | No | No | No |
| Interactive English tutor | No | No | No | No | Yes |

## Exercícios

- 1) Suponha um Robot autónomo com 2 rodas motrizes, 3 sensores de proximidade, 1 sensor de chão e 1 sensor de farol que se move num labirinto povoado por outros robots, tentando atingir a zona do mesmo onde se encontra o farol!
  - A) Em que consistem as percepções do agente?
  - B) Em que consistem as acções do agente?
  - C) Efectue uma classificação PAGE do ambiente.
  - D) Que tipo de arquitectura de agente lhe parece mais adequado nesta situação?
  - E) Suponha que o agente quer unicamente mover-se no labirinto sem bater em nenhum outro robot! Implemente um agente (o mais simples possível) capaz de o fazer!

## Resolução de Problemas por Pesquisa

- Formulação de Problemas
- Pesquisa Não Informada
- Pesquisa Informada
- Algoritmos Iterativos
- Jogos

## Resolução de Problemas - Pesquisa

- Como é que um agente pode agir, estabelecendo objectivos e considerando possíveis sequencias de acções para atingir esses objectivos!
- Resolução de Problemas:
  - Formulação de um problema como um problema de pesquisa
  - Pesquisa não Informada (estratégias de pesquisa)
  - Pesquisa Informada (pesquisa gulosa, algoritmo A*)
  - Algoritmos de Melhoria Iterativa
  - Jogos (em que é incluído um agente hostil!)

## Agente para Resolver Problemas

- "Problem Solving Agent": Procura encontrar a sequência de acções que leva a um estado desejável!
- Formulação do Problema:
  - Quais as acções possíveis? (qual o seu efeito sobre o estado do mundo?)
  - Quais os estados possíveis? (como representá-los?)
  - Como avaliar os Estados
- Problema de pesquisa
  - Solução: sequência de acções
- Fase final é a execução!
- Formular $\rightarrow$ Pesquisar $\rightarrow$ Executar

## Agente para Resolver Problemas

- Formulação do Problema:
  - Representação do Estado
  - Estado Inicial (Actual)
  - Teste Objectivo (define os estados desejados)
  - Operadores (Nome, Pré-Condições e Efeitos)
  - Custo da Solução

## Agente de Resolução de Problemas Simples

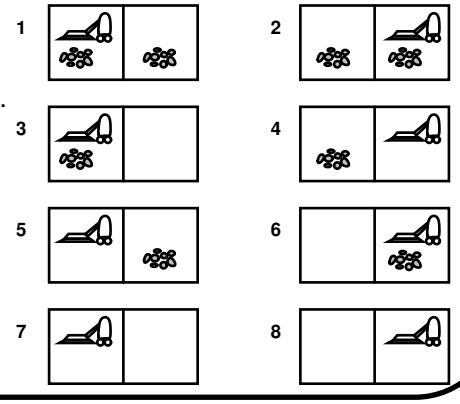
```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
          state, some description of the current world state
          g, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
      g ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH( problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

## Formulação do Problema

- Qual o conhecimento do agente sobre o estado do mundo e sobre as suas acções?
- Quatro tipos de problemas distintos:
  - Problemas de estado único (ambiente determinístico e acessível)
  - Problemas de múltiplos estados (ambiente determinístico mas inacessível)
  - Problemas de contingência (ambiente não determinístico e inacessível, é necessário usar sensores durante a execução, solução é uma árvore ou política)
  - Problemas de exploração (espaço de estados desconhecido)

## Exemplo: Problema do Aspirador

- 2 localizações, 3 Acções (left, right, suck), 8 Estados possíveis, Objectivo: limpar o lixo!
- Problema de:
  - Estado Único se…
  - Múltiplos Estados se…
  - Contingência se…
  - Exploração se...

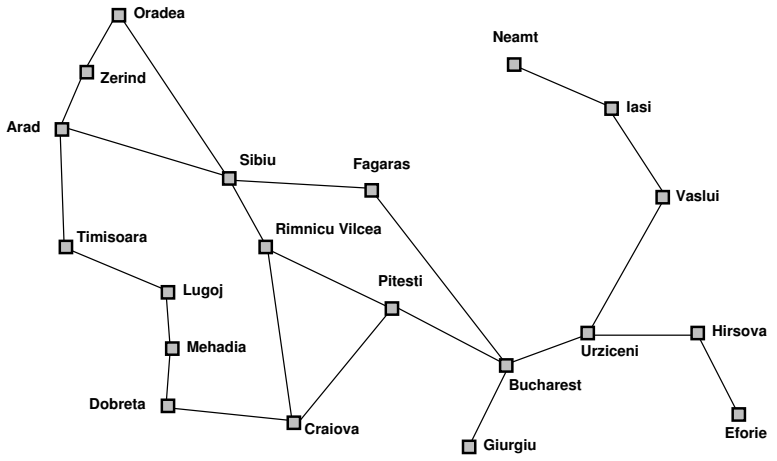


# Problemas Bem Definidos (estado único)

- Problema: Colecção de informação que o agente vai usar para decidir o que fazer!
- Formulação do Problema:
  - Espaço de Estados:
    - Estado Inicial
    - Conjunto de Acções possíveis (operadores, função sucessores)
  - Teste do Objectivo
  - Função de Custo da Solução
- **datatype** PROBLEM

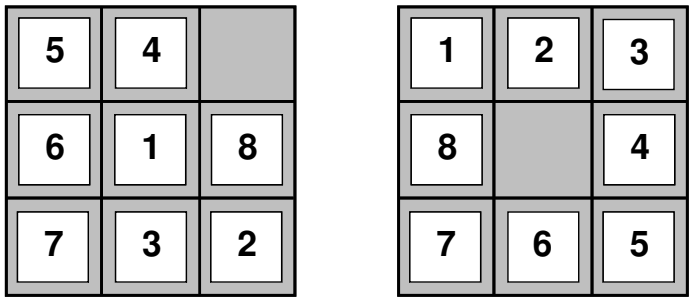
  **components:** INITIAL-STATE, OPERATORS, GOAL-TEST, PATH-COST-FUNCTION
- Solução: Caminho do estado inicial até ao objectivo
- Custo Total = Custo da Solução + Custo da pesquisa

# Exemplo: Problema do Mapa de Estradas da Roménia



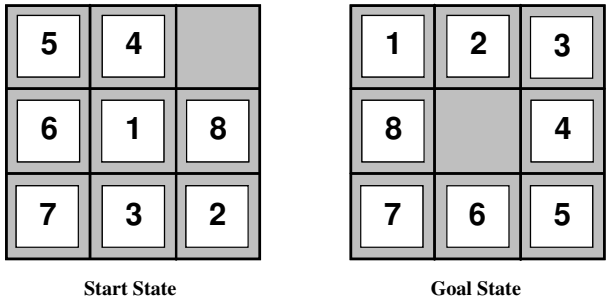
## Problema do Puzzle-8

- Estados, Operadores, Teste Objectivo, Custo da Solução



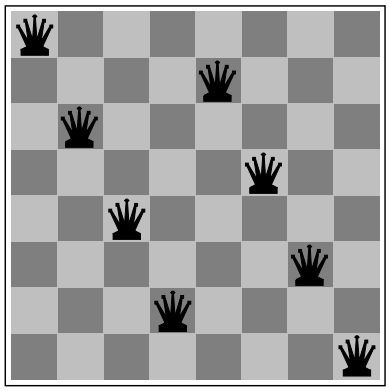

## Problema do Puzzle-8

- Estados: Localização dos 8 blocos (várias representações possíveis)
- Operadores: Buraco move-se para direita, esquerda, cima ou baixo
- Teste Objectivo: Representado na Figura
- Custo da Solução: Número de movimentos até ao objectivo



## Problema das N-Rainhas

- Estados, Operadores, Teste Objectivo, Custo da Solução



## Problema das N-Rainhas

- Teste Objectivo: 8 Rainhas no tabuleiro sem nenhum ataque
- Custo da Solução: 0
- Formulação 1:
  - Estado: Qualquer arranjo de 0 a 8 Rainhas no tabuleiro
  - Operador: Adicionar uma rainha em qualquer quadrado
    - Temos $64^8$ sequências possíveis!
- Formulação 2:
  - Estado: Arranjos de 0 a 8 Rainhas, uma em cada coluna, sem ataques!
  - Operador: Adicionar uma rainha na coluna mais à esquerda que estiver vazia, sem atacar nenhuma outra
    - Temos 2057 sequências possíveis!
- Formulação 3:
  - Estado: Arranjos de 8 Rainhas no tabuleiro, uma em cada coluna!
  - Operador: Movimentar rainha atacada para casa da mesma coluna

## Criptogramas

- Encontrar dígitos (todos diferentes), um para cada letra de forma a que a soma seja correcta!
- Estados: Puzzle com algumas letra substituídas por números
- Operadores: Substituir todas as ocorrências de uma letra por um dígito
- Teste Objectivo: Puzzle só contém dígitos e a soma está correcta!
- Custo da Solução: 0 (todas as soluções são iguais)

```
  FORTY
    TEN
+   TEN
-------
  SIXTY
```

```
C1 C2 C3 C4
   S E N D
+  M O R E
-----------
M O N E Y
```

## Criptogramas

- Soluções dos Criptogramas:
  - Criptograma 1: F=2, O=9, R=7, T=8, Y=6, E=5, N=0, S=3, I=1, X=4
    - Haverá mais soluções?
  - Criptograma 2: S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2
- Exercício: Inventar um criptograma!

```
  FORTY
    TEN
+   TEN
-------
  SIXTY
```

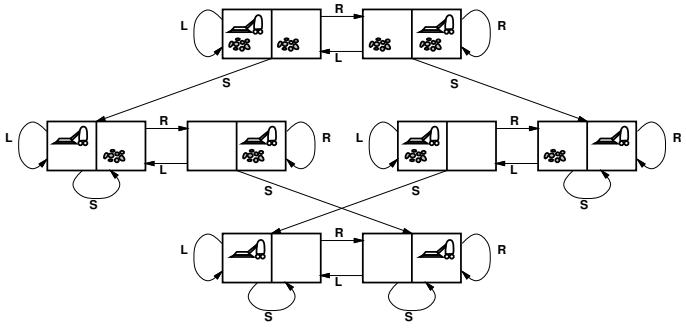
```
  29786
    850
+   850
-------
  31486
```

```
C1 C2 C3 C4
   S E N D
+  M O R E
-----------
M O N E Y
```

```
 1  0  1  1
   9 5 6 7
+  1 0 8 5
-----------
 1 0 6 5 2
```

## Exemplo: Mundo do Aspirador

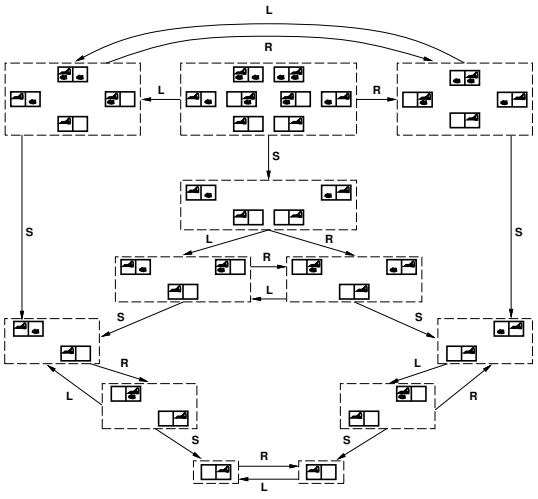
- Estado: 8 estados representados!
- Operadores: esquerda, direita, aspirar
- Teste Objectivo: Não há lixo em nenhum quadrado
- Custo da Solução: Cada acção custa 1




## Exemplo: Mundo do Aspirador sem Sensores!

- Problema de Múltiplos Estados
  - Agora temos em cada instante um conjunto de estados possíveis!
- Formulação do Problema
  - Conjunto de Estados: Subconjunto dos estados representados
  - Operadores: esquerda, direita e aspirar
  - Teste Objectivo: Todos os estados do conjunto não podem ter lixo
  - Custo da Solução: Cada acção custa 1

## Exemplo: Mundo do Aspirador sem Sensores!

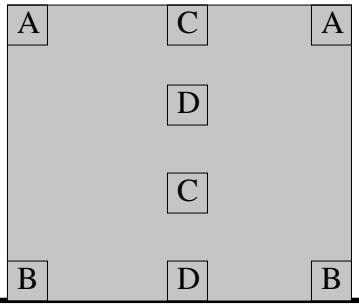



## Exercício: Missionários e Canibais

- Problema dos Missionários e Canibais
- Descrição:
  - 3 missionários e 3 canibais estão numa das margens do rio com um barco que só leva 2 pessoas. Encontrar uma forma de levar os 6 para a outra margem do rio sem nunca deixar mais canibais do que missionários numa das margens durante o processo!
- Formular este problema como um problema de pesquisa, definindo o estado inicial, os estados possíveis, os operadores, o teste objectivo e o custo da solução.

## Exercício: Quadrado Impossível

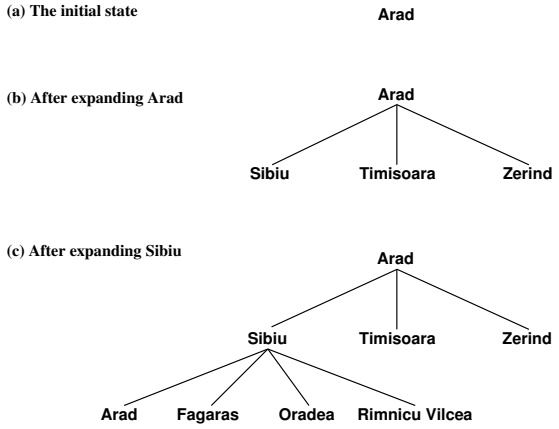
- Problema:
  - Dado o quadrado apresentado, ligar o A com o A, o B com o B, o C com o C e o D com o D sem cruzar nenhuma linha!
- Formular o problema do quadrado impossível como um problema de pesquisa e resolve-lo!




## Pesquisa da Solução

- Como se faz a pesquisa?
  - 1) Começar com o estado inicial
  - 2) Executar o teste do objectivo
  - 3) Se não tivermos encontrado a solução, usar os operadores para expandir o estado actual gerando novos estados (expansão)
  - 4) Executar o teste objectivo
  - 5) Se não tivermos encontrado a solução, escolher qual o estado a expandir a seguir (estratégia de pesquisa) e realizar essa expansão
  - 6) Voltar a 4)

## Pesquisa da Solução - Árvore de Pesquisa




## Pesquisa da Solução - Árvore de Pesquisa

- Árvore de pesquisa composta por nós. Nós folhas, ou não têm sucessores ou ainda não foram expandidos!
- Importante distinguir entre a árvore de pesquisa e o espaço de estados!

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

# Pesquisa da Solução - Estrutura de Dados

- Nó da Árvore (cinco componentes):
  - Estado a que corresponde
  - Nó que lhe deu origem (pai)
  - Operador aplicado para o gerar
  - Profundidade do nó
  - Custo do caminho desde o nó inicial
- **datatype** NODE

  **components:** STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST
- Fronteira: Conjunto de nós à espera de serem expandidos
  - Representada como uma fila

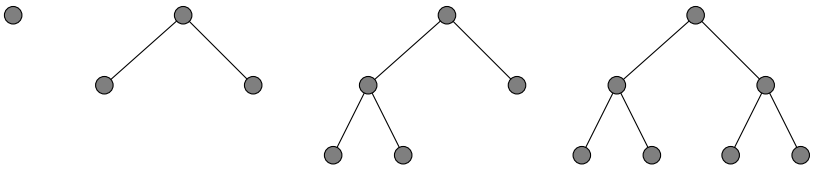
```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

# Estratégias de Pesquisa

- Critérios de Avaliação:
  - Complitude: Está garantido que encontra a solução?
  - Complexidade no Tempo: Quanto tempo demora a encontrar a solução?
  - Complexidade no Espaço: Quanta memória necessita para fazer a pesquisa?
  - Optimalidade: Encontra a melhor solução?
- Tipos de Estratégias de Pesquisa:
  - Pesquisa Não-Informada (cega)
  - Pesquisa Informada (heurística)

# Pesquisa Primeiro em Largura

- Pesquisa Primeiro em Largura (Breadth-first search)
  - Estratégia: Todos os nós de menor profundidade são expandidos primeiro
    - Bom: Pesquisa muito sistemática
    - Mau: Normalmente demora muito tempo e sobretudo ocupa muito espaço



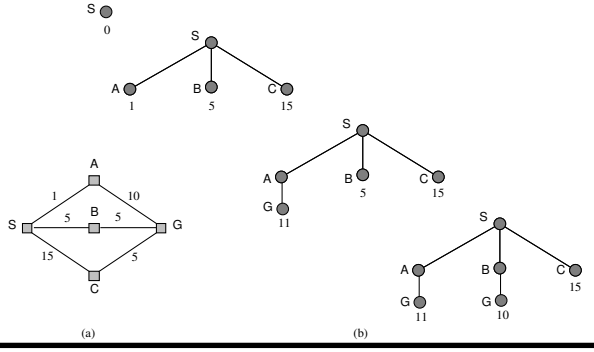
# Pesquisa Primeiro em Largura (2)

- Supondo factor de ramificação b então $n=1+b+b^2+b^3+ \ldots +b^n$
- Complexidade exponencial no espaço e no tempo: $O(b^d)$
- Em geral só pequenos problemas podem ser resolvidos assim!
- **function** BREADTH-FIRST-SEARCH(problem) **returns** a solution or failure
  GENERAL-SEARCH(problem,ENQUEUE-AT-END)

| 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 |

# Pesquisa de Custo Uniforme

- Estratégia: Expandir sempre o nó com menor custo da fronteira (medido pela função de custo da solução)
- Pesquisa Primeiro em Largura é igual a Pesquisa de Custo Uniforme se g(n)=Depth(n)

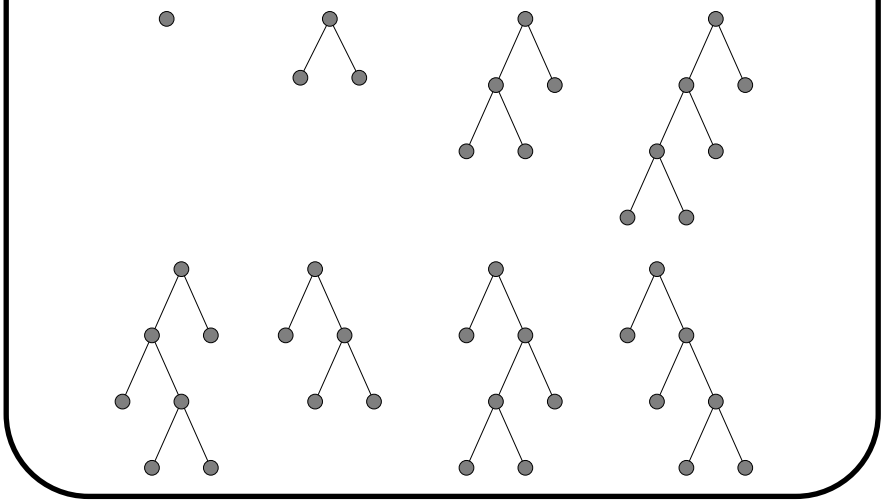



# Pesquisa Primeiro em Profundidade

- Pesquisa Primeiro em Profundidade (Depth-First Search)
  - Estratégia: Expandir sempre um dos nós mais profundos da árvore
    - Bom: Muito pouca memória necessária, bom para problemas com muita soluções
    - Mau: Não pode ser usada para árvores com profundidade infinita, pode ficar presa em ramos errados
  - Complexidade no tempo $O(b^m)$ e no espaço O(bm).
  - Por vezes é definida uma profundidade limite e transforma-se em Pesquisa com Profundidade Limitada
- **function** DEPTH-FIRST-SEARCH(problem) **returns** a solution or failure

  GENERAL-SEARCH(problem,ENQUEUE-AT-FRONT)

## Pesquisa Primeiro em Profundidade (2)

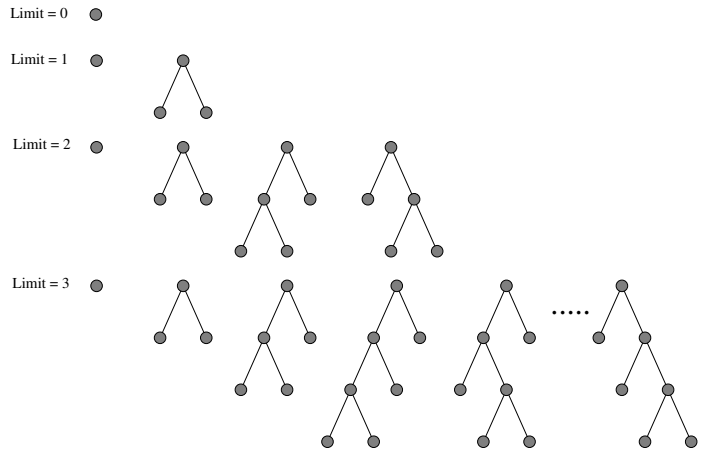


## Pesquisa em Profundidade Iterativa

- Estratégia: Executar pesquisa em profundidade limitada, iterativamente, aumentando sempre o limite da profundidade
- Complexidade no tempo $O(b^d)$ e no espaço $O(bd)$.
- Em geral é a melhor estratégia para problemas com um grande espaço de pesquisa e em que a profundidade da solução não é conhecida

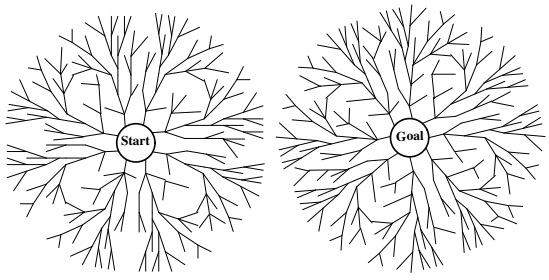
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

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

## Pesquisa em Profundidade Iterativa (2)




## Pesquisa Bidireccional

- Estratégia: Executar pesquisa para a frente desde o estado inicial e para trás desde o objectivo, simultaneamente
  - Bom: Pode reduzir enormemente a complexidade no tempo
  - Problemas: Será possível gerar os predecessores? E se existirem muitos estados objectivo? Como fazer o "matching" entre as duas pesquisas? Que tipo de pesquisa fazer nas duas metades?



# Comparação entre as Estratégias de Pesquisa

- Avaliação das estratégias de pesquisa! B é o factor de ramificação; d é a profundidade da solução, m é a máxima profundidade da árvore; l é a profundidade limite

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^m$ | $b^l$ | $b^d$ | $b^{d/2}$ |
| Space | $b^d$ | $b^d$ | $bm$ | $bl$ | $bd$ | $b^{d/2}$ |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes, if $l \geq d$ | Yes | Yes |

- Outro Problema: Evitar Estados Repetidos
  - Não voltar ao estado anterior
  - Não criar ciclos
  - Não usar nenhum estado repetido

# Exercícios - Pesquisa (1)

- Formular os seguintes problemas como problemas de pesquisa. Existem diversas possibilidades com diferentes níveis de detalhe.
  - A) Encontrar o número de telefone do Luís Paulo Reis, que vive em Espinho, dada uma pilha de 20 fichas ordenadas (uma por cidade), cada qual contendo um conjunto de nomes (ordenados por apelido) e respectivos telefones, ordenados alfabeticamente.
  - B) Como em A) mas supondo que você tinha esquecido o apelido
  - C) Colorir um mapa plano, usando 4 cores de forma a que dois vértices adjacentes não tenham a mesma cor
  - D) Resolver a paciência de cartas "solitário" supondo que todas as cartas estão descobertas no início
  - E) Num pequeno país da América do Sul, o objectivo é encontrar uma Farmácia. Todos os habitantes estão a dormir nas suas casas e não existe um mapa.

## Exercícios - Pesquisa (2)

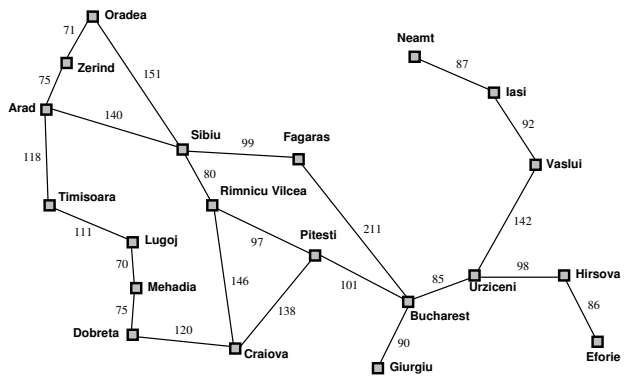
- Formular os seguintes problemas como problemas de pesquisa e resolve-los utilizando a estratégia primeiro em largura.
  - O problema da corrente consiste em juntar um conjunto de elos de corrente para formar uma corrente completa. Os operadores podem abrir um elo ou fechar um elo. Na forma mais usual, o estado inicial é composto por quatro correntes com três elos e o objectivo por uma corrente circular com 12 elos.
  - Suponha que um Pastor leva consigo, um Lobo, um Carneiro e uma Couve. Chegado ao Rio, dispõe unicamente de um barco capaz de o transportar e transportar mais um item. O objectivo do pastor é passar para o outro lado levando os três itens mas se deixar sozinhos numa das margens, o lobo e carneiro ou o carneiro e a couve vai ter problemas!
- Formular como problemas de pesquisa (o jogo 3x3 da corrente, o jogo dos pentaminós, problemas de Tangram, etc…)

## Pesquisa Informada - Melhor Primeiro

- Pesquisa Informada- Utiliza informação do problema para evitar que o algoritmo de pesquisa fique "perdido vagueando no escuro"!
- Estratégia de Pesquisa: Definida escolhendo a ordem de expansão dos nós!
- Pesquisa do Melhor Primeiro (Best-First Search)
  - Utiliza uma função de avaliação que retorna um número indicando a desirabilidade de expandir um nó
- Pesquisa Gulosa (Greedy-Search)
- Algoritmo A*

```
function BEST-FIRST-SEARCH( problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH( problem, Queueing-Fn)
```

# Pesquisa Informada - Exemplo

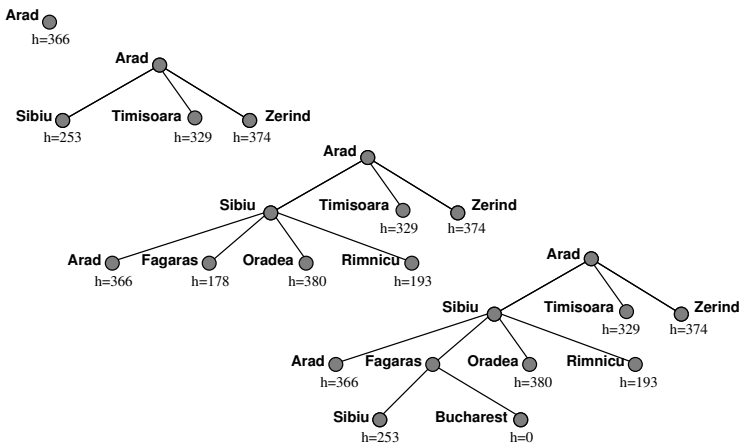



| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234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

# Pesquisa Gulosa (Greedy-Search)

- Estratégia: Expandir o nó que parece estar mais perto da solução
- h(n)= custo estimado do caminho mais curto do estado n para o objectivo
- **function** GREEDY-SEARCH(problem) **returns** a solution or failure
  **return** BEST-FIRST-SEARCH(problem,h)
- Exemplo: $h_{SLD}(n)$= distância em linha recta entre n e o objectivo
- Propriedades:
  - Completa? Não! Pode entrar em ciclos!
  - Susceptível a falsos começos.
  - Complexidade no tempo? $O(b^m)$ no espaço? $O(b^m)$ mas com uma boa função heurística pode diminuir consideravelmente
  - Óptima? Não! Não encontra sempre a solução óptima!
  - Necessário detectar estados repetidos!

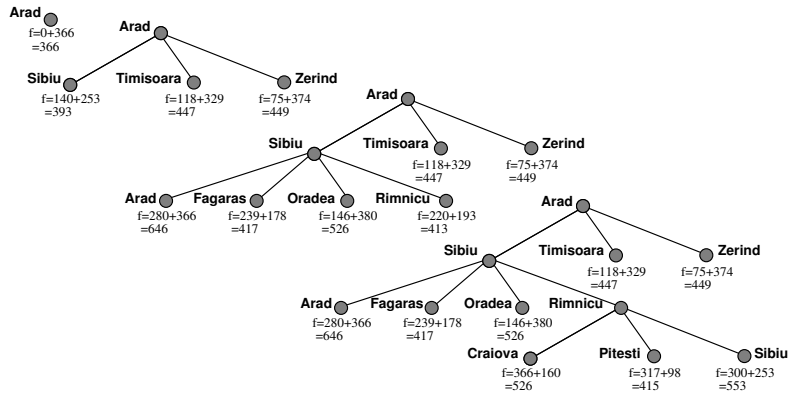
# Pesquisa Gulosa (Greedy-Search) (2)



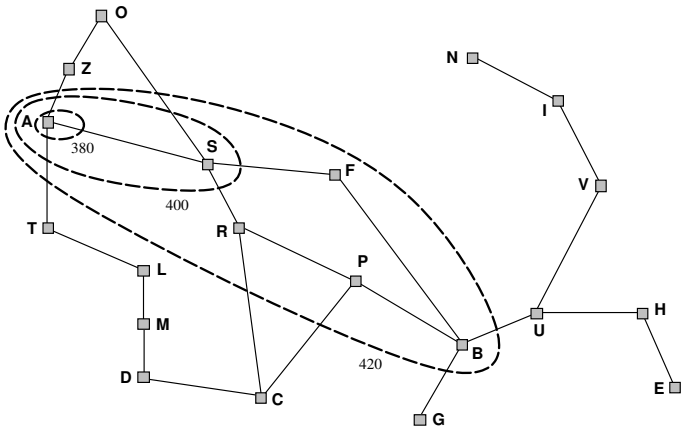

# Pesquisa A*

- Estratégia: O algoritmo A* combina a pesquisa gulosa com a uniforme minimizando a soma do caminho já efectuado com o mínimo previsto que falta até a solução. Usa a função:
  - f(n) = g(n) + h(n)
  - g(n) = custo até agora para chegar a n
  - h(n) = custo estimado para chegar ao objectivo
  - f(n)= custo estimado da solução mais barata que passa pelo nó n
  - h(n) não pode sobre-estimar o custo para chegar à solução!
- **function** A*-SEARCH(problem) **returns** a solution or failure
  **return** BEST-FIRST-SEARCH(problem,g+h)
  - Algoritmo A* é óptimo e completo!
  - Complexidade no tempo exponencial em (erro relativo de h*comprimento da solução)
  - Complexidade no espaço: Mantém todos os nós em memória!

# Pesquisa A* (2)

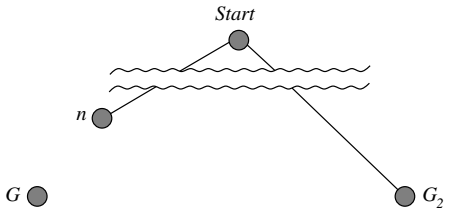



# Pesquisa A* (3)



## Prova da Optimalidade do A*

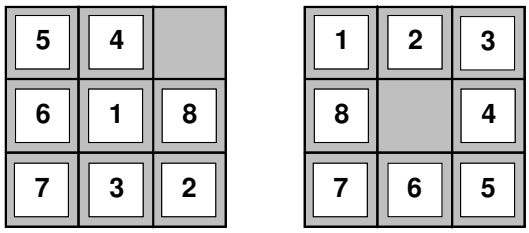
- Supondo que um objectivo sub óptimo G2 foi gerado e está na lista. Sendo n um nó ainda não expandido que leva até ao objectivo óptimo G1.




- f(G2) = g(G2) pois h(G2)=0
- f(G2) > g(G1) pois G2 é sub óptimo
- f(G2) >= f(n) pois h é uma heurística admissível
- Logo, o algoritmo A* nunca escolherá G2 para expansão!

## Funções Heurísticas - 8 Puzzle

- Solução típica em 20 passos com factor de ramificação médio: 3
- Número de estados: $3^{20} = 3.5*10^9$
- Nº Estados (sem estados repetidos) = 9! = 362880
- Heurísticas:
  - h1 = Nº de peças for a do sítio
  - h2 = Soma das distâncias das peças até às suas posições correctas



## Funções Heurísticas - 8 Puzzle (2)

- Relaxação de Problemas como forma de inventar heurísticas:
  - Peça pode-se mover de A para B se A é adjacente a B e B está vazio
  - a) Peça pode-se mover de A para B se A é adjacente a B
  - b) Peça pode-se mover de A para B se B está vazio
  - c) Peça pode-se mover de A para B

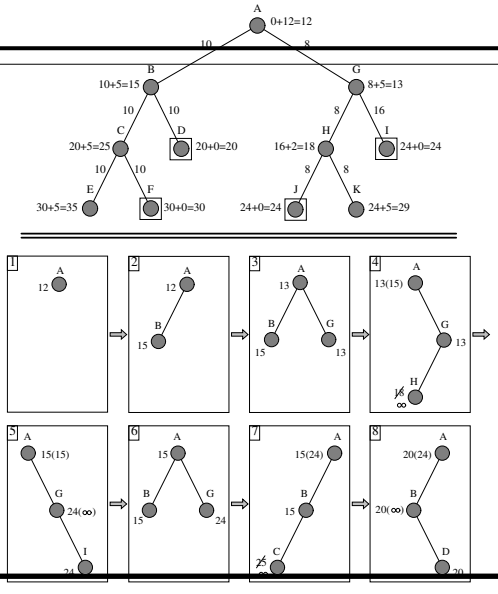
## Pesquisa com Memória Limitada - IDA*/SMA*

- IDA* - Pesquisa com Profundidade Iterativa (Iterative Deepening Search)
  - Estratégia: Utilização de um custo limite em cada iteração e realização de pesquisa em profundidade iterativa
  - Problemas em alguns problemas reais com funções de custo com muitos valores
- SMA* - Pesquisa Simplificada com Memória Limitada (Simplified Memory Bounded A*)
  - IDA* de uma iteração para a seguinte só se lembra de um valor (o custo limite)
  - SMA* utiliza toda a memória disponível, evitando estados repetidos
  - Estratégia: Quando necessita de gerar um sucessor e não tem memória, esquece um nó da fila que pareça pouco prometedor (com um custo alto).

## IDA* - Pesquisa com Profundidade Iterativa (Iterative Deepening Search)

## SMA*(Simplified Memory Bounded A*)

.



## SMA*(Simplified Memory Bounded A*) (2)

## Algoritmos de Melhoria Iterativa

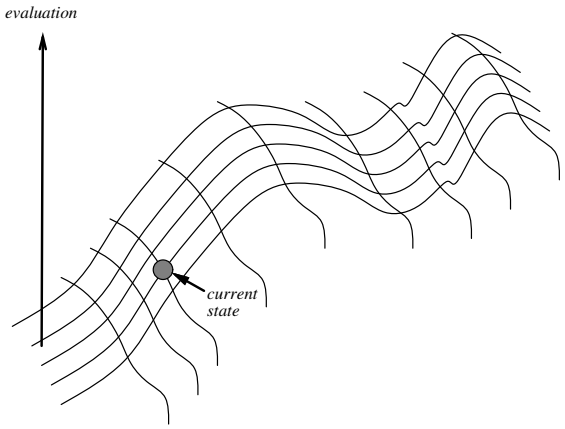
- Em muitos problemas de optimização, o caminho para o objectivo é irrelevante! O objectivo é ele mesmo a solução!
- Espaço de Estados = conjunto das configurações completas!
- Algoritmos Iterativos mantém um único estado (corrente) e tentam melhora-lo!
- Algoritmos de Melhoria Iterativa:
  - Pesquisa Subida da Colina (Hill-Climbing Search)
  - Arrefecimento Simulado (Simulated Annealing)
  - Pesquisa Tabu (Tabu Search)
  - Algoritmos Genéticos
- Estratégia: Começar como uma solução do problema e fazer alterações de forma a melhorar a sua qualidade

## Pesquisa Subida da Colina (Hill-Climbing Search)

- Problema: Dependendo do estado inicial pode ficar "preso" num mínimo local!

```
function HILL-CLIMBING( problem) returns a solution state
   inputs: problem, a problem
   static: current, a node
           next, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      next ← a highest-valued successor of current
      if VALUE[next] < VALUE[current] then return current
      current ← next
   end
```

## Pesquisa Subida da Colina (Hill-Climbing Search) (2)



# Arrefecimento Simulado ("Simulated Annealing")

- Estratégia: Escapar do mínimo local permitindo alguns "maus" movimentos mas gradualmente diminuindo a sua dimensão e frequência!

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T=0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

# Exercício

1) Suponha o seguinte jogo (para 1 jogador) em que o tabuleiro é constituído por 6 casas e 6 fósforos. O objectivo do jogo é colocar um fósforo em cada um dos quadrados (tal como é demonstrado na figura). Para tal, em cada jogada, o jogador pode:
   - Movimentar 1 fósforo de uma casa para outra casa que esteja à sua direita
   - Movimentar 2 fósforos de uma casa para outra casa que esteja à sua esquerda
   - Movimentar 2 ou 3 fósforos de uma casa para outra que esteja acima ou abaixo dela

a) Formule o problema como um problema de pesquisa

b) Partindo do estado representado abaixo, desenhe as árvores de pesquisa utilizando as estratégias primeiro em largura e primeiro em profundidade (considere primeiro os movimentos a partir do quadrado 1, depois a partir do 2 e assim sucessivamente).

c) Represente graficamente o espaço de estados (ignore estados em que um quadrado tenha mais de 3 fósforos).

d) Defina duas funções heurísticas que lhe permitam aplicar o algoritmo A* ao problema. Explique em que consiste o algoritmo A* e aplique-o para resolver o problema.

1 1 1 1 2 1
1 1 1 Estado Inicial 0 2 0 Alínea b)

## Exercícios - Resolução de Problemas por Pesquisa

- 1) Implemente usando uma linguagem convencional ou a linguagem Prolog, os algoritmos apresentados de pesquisa geral (e as estratégias de pesquisa descritas), o algoritmo de pesquisa do melhor primeiro: pesquisa gulosa e algoritmo A*.
- 2) Implementar os algoritmo da pesquisa subida da colina, arrefecimento simulado e pesquisa tabu utilizando uma linguagem convencional (C ou Pascal) ou Prolog.
- 3) Aplique os algoritmos implementados ao problema do problema das N-Rainhas e ao Puzzle-8 e compare os resultados obtidos.

## Jogos como Problemas de Pesquisa

- Agente Hostil (adversário) incluído no mundo!
- Oponente Imprevisível => Solução é um Plano de Contingência
- Tempo Limite => Pouco provável encontrar objectivo! É necessário uma aproximação
- Uma das áreas mais antigas da IA! Em 1950 Shannon e Turing criaram os primeiros programas de Xadrez!
- Xadrez:
  - Todos consideram que é necessário inteligência para jogar
  - Regras simples mas o jogo é complexo
  - Mundo totalmente acessível ao agente
  - Factor de ramificação médio de 35, partida com 50 jogadas => $35^{100}$ folhas numa árvore de pesquisa (embora só existam $10^{40}$ posições legais)
- Conceitos de corte na árvore de pesquisa e função de avaliação!

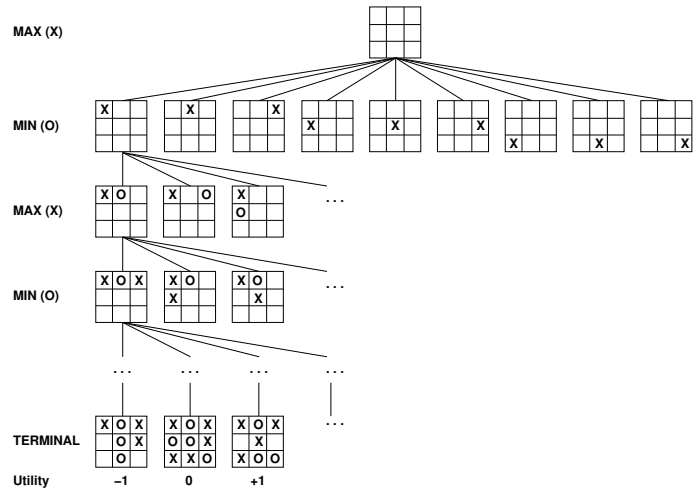
## Tipos de Jogos

- Tipos de Jogos:
  - Informação:
    - Perfeita: Xadrez, Damas, Go, Otelo, Gamão, Monopólio
    - Imperfeita: Poker, Scrabble, Bridge, King
  - Sorte/Determinístico:
    - Determinístico: Xadrez, Damas, Go, Otelo
    - Jogo de Sorte: Gamão, Monopólio, Poker, Scrabble, Bridge, King
- Plano de "Ataque":
  - Algoritmo para o jogo perfeito
  - Horizonte finito, avaliação aproximada
  - Cortes na árvores para reduzir custos

## Decisões Perfeitas em Jogos com 2 Adversários - MiniMax

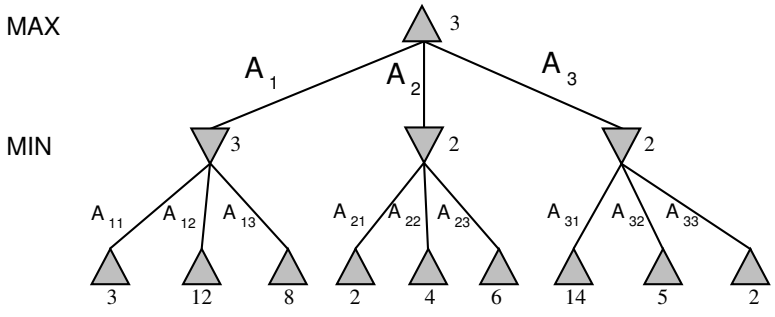
- Jogo: Problema de pesquisa com:
  - Estado Inicial (posição do tabuleiro e qual o próximo jogador a jogar)
  - Conjunto de Operadores (que definem os movimentos legais)
  - Teste Terminal (que determina se o jogo acabou ou seja está num estado terminal)
  - Função de Utilidade (que dá um valor numérico para o resultado do jogo, por exemplo 1-vitória, 0-empate, -1-derrota)
- Estratégia do algoritmo Minimax:
  - Gerar a árvore completa até aos estados terminais
  - Aplicar a função utilidade a esses estados
  - Calcular os valores da utilidade até a raiz da árvore, uma camada de cada vez
  - Escolher o movimento com o valor mais elevado!

## Minimax - Exemplo para o Jogo do Galo




## Minimax - Exemplo Geral

- Estratégia: Escolher o movimento que tem o maior valor minimax = melhor que se pode conseguir contra as melhores respostas do adversário!



# Algoritmo Minimax

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

# Propriedades do Minimax

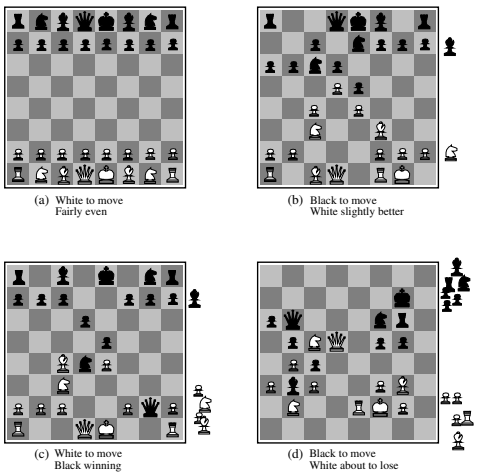
- Completo? Sim se a árvore for finita!
- Óptimo? Sim contra um adversário óptimo! Senão?
- Complexidade no Tempo? $O(b^m)$
- Complexidade no Espaço? O(bm) (exploração primeiro em profundidade)

- Problema: Inviável para qualquer jogo minimamente complexo.
- Exemplo: Para o xadrez (b=35, m=100), $b^m=35^{100}=2.5*10^{154}$ . Supondo que são analisadas 450 milhões de hipóteses por segundo => $2*10^{138}$ anos para chegar à solução!

# Recursos Limitados

- Supondo que temos 100 segundos e exploramos $10^4$ nós/segundo, podemos explorar $10^6$ nós por movimento
- Aproximação usual:
  - Teste de Corte: Profundidade Limite
  - Função de Avaliação: Desirabilidade estimada para a posição
- Exemplo (Xadrez):
  - Teste de Corte: Profundidade de Análise n
  - Função de Avaliação simples = soma dos valores das peças brancas em jogo menos a soma dos valores da peças negra em jogo!
  - Função de avaliação só deve ser aplicada a posições estáveis (em termos do seu valor). Por exemplo, posições com possíveis capturas devem ser mais exploradas…
  - Outro Problema: Problema do horizonte!

# Xadrez - Funções de Avaliação



(a) White to move
Fairly even

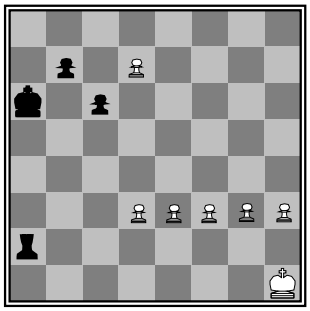
(b) Black to move
White slightly better

(c) White to move
Black winning

(d) Black to move
White about to lose

## Xadrez - Problema do Horizonte

- A partida está ganha para as brancas! Mas com um horizonte limitado não parece ser bem assim!



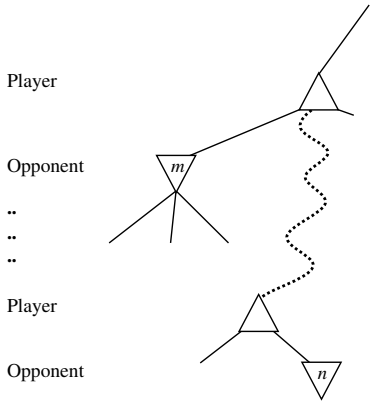
Black to move

## Cortes à Pesquisa

- MinimaxCutoff é idêntico ao MinimaxValue excepto:
  - Terminal-Test é substituído por Cutoff
  - Utility é substituída por Evaluation (que calcula uma avaliação da posição atingida)
- Será que funciona na prática?
  - Se $B^m=10^6$ com b=35 => m=4
- Um jogar de xadrez com profundidade 4 é absolutamente miserável!
  - Profundidade 4 => Jogador Novato
  - Profundidade 8 => PC, M.Bom Jogador Humano
  - Profundidade 12 => Deep Blue, Kasparov

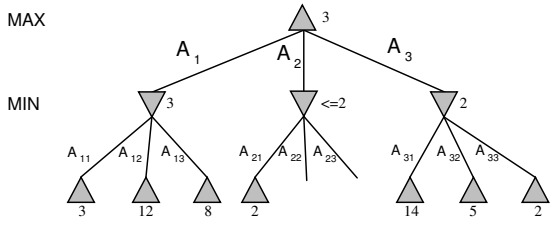
# Cortes Alfa-Beta

- $\alpha$ é o melhor valor (para Max) encontrado até agora no caminho corrente
- Se V for pior do que $\alpha$, Max deve evitá-lo => cortar o ramo
- $\beta$ é definido da mesma forma para Min




# Cortes Alfa - Beta (2)

- Cortes Alfa-Beta não afectam o resultado final
- Boa ordenação melhora a eficiência dos cortes
- Com ordenação perfeita: Complexidade no Tempo = $O(b^{m/2})$
  - Duplica a profundidade de pesquisa
  - Profundidade 8 => Bom jogador de Xadrez
- Bom exemplo do valor de raciocinar sobre que computações são relevantes



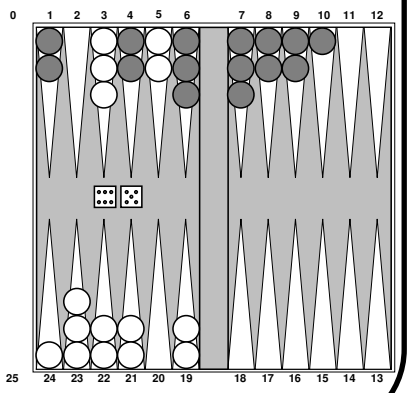
## Cortes Alfa - Beta (3)

## Jogos Determinísticos

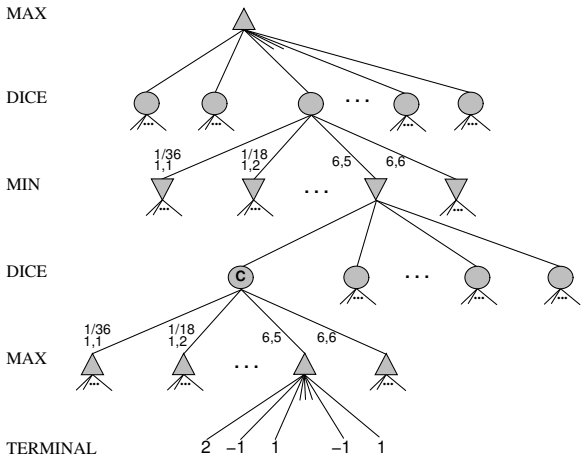
- Damas: Chinook  acabou com o reinado de 40 anos do campeão humano Marion Tinsley em 1994. Usava uma base de dados para finais de partida definindo a forma perfeita de vencer para todas as posições involvendo 8 ou menos peças (no total de 443748401247 posições)
- Deep Blue derrotou o campeão do mundo humano Gary Kasparov num jogo com 6 partidas em 1997. Deep Blue pesquisa 200 milhões de posições por segundo e usa uma função de avaliação extremamente sofisticada e métodos (não revelados) para estender algumas linhas de pesquisa para além da profundidade 40!
- Otelo: Campeões humanos recusam-se a competir com computadores pois não têm qualquer hipótese! (b entre 5 e 15)
- Go: Campeões humanos recusam-se a competir com computadores pois estes não conseguem jogar razoavelmente (b>300).

# Jogos de Azar

- Em muitos jogos, ao contrário do xadrez, existem eventos externos que afectam o jogo, tais como tirar uma carta ou lançar um dado!
- Exemplos: Jogos de cartas, Gamão, Scrabble, ...
- Árvore de pesquisa deve incluir nós de probabilidade!
- Decisão é efectuada com base no valor esperado!
- ExpectiMiniMax




# Jogos de Azar (2)



## Sumário - Jogos

- Trabalhar com jogos é extremamente engraçado (e perigoso)
  - Fácil testar novas ideias!
  - Fácil comparar agentes com outros agentes e com humanos!
- Jogos ilustram diversos pontos interessantes da IA:
  - Perfeição é inatingível => é necessário aproximar!
  - É boa ideia pensar sobre o que pensar!
  - Incerteza restringe a atribuição de valores aos estados!
- Jogos funcionam para a IA como a Formula 1 para a construção de automóveis!

## Exercícios - Jogos

- 1) Implementar o algoritmo Minimax (sem e com cortes alfa-beta) utilizando uma linguagem convencional ou o Prolog.
- 2) Formular o jogo dos Pentaminós (tabuleiro 8x8 para 2 jogadores) como um jogo e projectar um agente inteligente capaz de o jogar
- 3) Construir agentes capazes de jogar 4 em Linha, Naikey, Damas, Otelo e Xadrez.
- 4) Descrever e/ou implementar descrições do estado, geradores de movimentos e funções de avaliação para os seguintes jogos: Gamão, Monopólio, Scrabble.
- 5) Construir um agente capaz de jogar os jogo de cartas Viúva Negra (Hearts), King, Sobe e Desce, Poker, Lerpa e Bridge.