

Universidade de Aveiro

*Inteligência Artificial (LEI)*

*Introdução à Inteligência Artificial (MIECT)*

# Tópicos de Inteligência Artificial

Ano lectivo 2020/2021

Regente: Luís Seabra Lopes

# Definição de “Inteligência” - I

- Segundo [www.dictionary.com](http://www.dictionary.com), “inteligência” é:
  - Capacidade de adquirir e aplicar conhecimento
  - Capacidade de pensar e raciocinar
  - O conjunto de capacidades superiores da mente.

# Definição de “Inteligência” - II

- O estudo da inteligência envolve [Albus,1995]:
  - Como adquirir, representar e armazenar conhecimento
  - Como gerar comportamento inteligente
  - Como surgem e são utilizadas as motivações, emoções e prioridades
  - Como as percepções (sinais) dão origem a entidades simbólicas
  - Como raciocinar sobre o passado
  - Como planejar acções no futuro
  - Como surgem fenómenos como a ilusão, crença, esperança, amor, etc.

# Definição de “Inteligência Artificial”

- “Inteligência Artificial” é a disciplina que estuda as teorias e técnicas necessárias ao desenvolvimento de “artefactos” inteligentes. [Nilsson, 1998]
- Direcções seguidas [Russell & Norvig, 1995]:

|                          |                      |
|--------------------------|----------------------|
| Pensar como o ser humano | Pensar racionalmente |
| Agir como o ser humano   | Agir racionalmente   |

# História até à “Inteligência Artificial”

- Século IV a.C. – Aristóteles estabelece os fundamentos da lógica e do pensamento puramente racional.
- Séculos XVI-XVII – Bacon e Locke estabelecem os fundamentos do “empirismo”: “Nada está na compreensão que não tenha estado primeiro nos sentidos”.
- Séculos XIX-XX – Duas correntes na psicologia: “comportamentalismo” e “cognitivismo”.
- 1940 – início da era dos computadores
- 1943 – McCulloch & Pitts propõem um modelo de computação vagamente inspirado no cérebro humano: redes de unidades chamadas “neurónios” podiam implementar qualquer função.
- 1951 – primeiro programa que joga xadrez
- 1956 – a expressão “Inteligência Artificial” foi usada pela primeira vez.

# História da “Inteligência Artificial” - I

- 1958 – McCarthy usa lógica de primeira ordem para representar conhecimento numa espécie de “sistema pericial”.
- 1959 – *GPS: General Problem Solver* – aqui surge um assunto hoje clássico – pesquisa para resolução de problemas
- 1962 – Rosenblatt propõe o uso do “perceptrão” (rede de neurónios) para aprendizagem e reconhecimento de padrões
- 1966 – *CLS: Concept Learning System* – primeiro sistema de aprendizagem simbólica

# História da “Inteligência Artificial” - II

- 1970 – Surge o Prolog, uma linguagem de programação em lógica
- 1971 – DENDRAL: primeiro sistema pericial (reconstruía a estrutura de moléculas orgânicas)
- 1986 – Robôs comportamentalistas
- 1986 – Ressurgimentos das redes neuronais
- 1997 – O IBM Deep Blue vence uma partida de xadrez contra Kasparov
- 1997 – Primeiro campeonato mundial de futebol robótico

# Tópicos de Inteligência Artificial

- Agentes
  - Noção de agente
  - Objectivo da Inteligência Artificial
  - Agentes reactivos e deliberativos
  - Propriedades do mundo de um agente
  - Arquitecturas de agentes
- Representação do conhecimento
- Técnicas de resolução de problemas



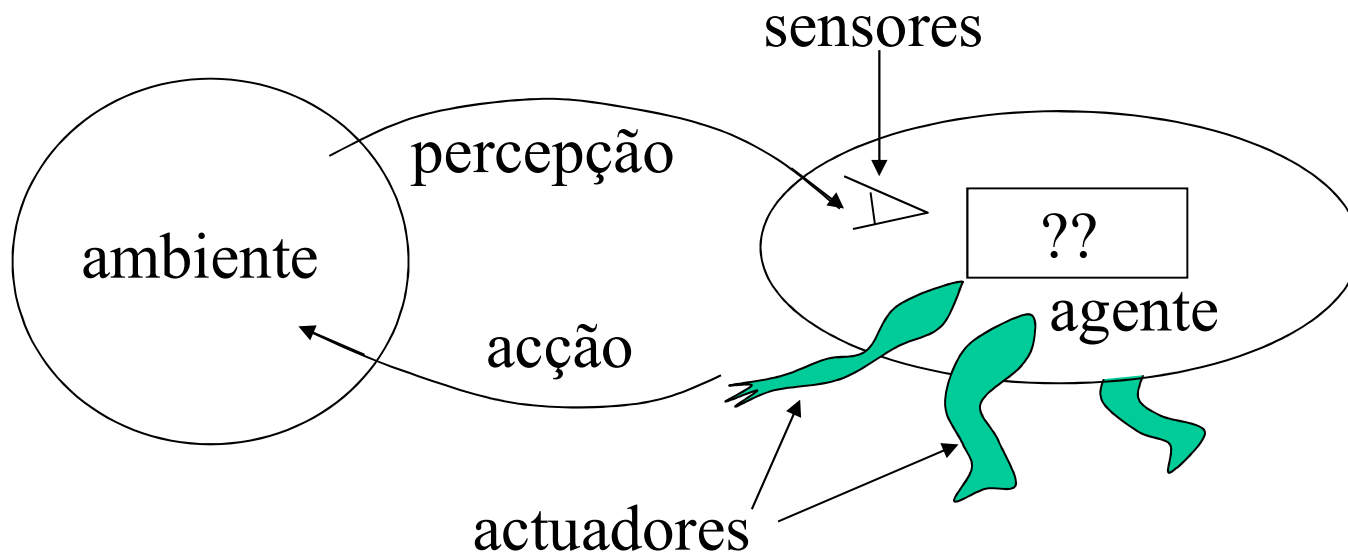
# Definição de “Agente”

- Nesta disciplina estudamos técnicas úteis no desenvolvimento de “agentes inteligentes”
- Segundo [www.dictionary.com](http://www.dictionary.com), um “agente” pode ser uma:
  - Entidade com poder ou autoridade de agir
  - Entidade que actua em representação de outrem

# Definição de “Agente”

- Agente – uma entidade com capacidade de obter informação sobre o seu ambiente (através de “sensores”) e de executar acções em função dessa informação (através de “actuadores”). [Russell & Norvig, 1995]
- Exemplos:
  - Agente físico: robô anfitrião
  - Agente de software: agente móvel de pesquisa de informação na internet

# Agente



# Exemplos de agentes

| Tipo de agente                            | Percepção                       | Acção   | Objectivos                                 | Ambiente                         |
|---|---------------------------------|---|--|----------------------------------|
| Sistema de diagnóstico médico             | Sintomas, respostas do paciente | Perguntas, testes, tratamentos                | Saúde do paciente, custo mínimo            | Paciente, hospital               |
| Sistema de análise de imagens de satélite | Imagem                          | Devolver uma categorização da cena            | Categorização correcta                     | Imagens de um satélite em órbita |
| Braço robótico para em embalagem          | Imagem, sinal de força          | Colocar peças em caixas                       | Colocar as peças na posição correcta       | Alimentador de peças, caixas     |
| Controlador de refinaria                  | Temperatura, pressão            | Abrir e fechar válvulas; ajustar temperatura  | Pureza, segurança                          | Refinaria                        |
| Tutor de inglês interactivo               | Palavras introduzidas           | Propôr exercícios, corrigi-los, dar sugestões | Maximizar o resultado dos alunos num teste | Conjunto de alunos               |

# Voltando à definição de “Inteligência Artificial”

- “Inteligência Artificial” é a disciplina que estuda as teorias e técnicas necessárias ao desenvolvimento de “artefactos” inteligentes. [Nilsson, 1998]
- Direcções seguidas [Russell & Norvig, 1995]:

|                          |                      |
|--------------------------|----------------------|
| Pensar como o ser humano | Pensar racionalmente |
| Agir como o ser humano   | Agir racionalmente   |

# Agir como o ser humano

## – o Teste de Turing

- “Comportamento inteligente” – a capacidade de um artefacto obter desempenho comparável ao desempenho humano em todas as actividades cognitivas. [Turing, 1950]
- Teste de Turing – é uma definição operacional de comportamento inteligente de nível humano:
  - Consiste em submeter o artefacto a um interrogatório realizado por um ser humano através de um terminal de texto.
  - Se o humano não conseguir concluir se está a interrogar um artefacto ou outro ser humano, então, esse artefacto é inteligente.
- Os sistemas deste tipo serão o objectivo principal da “Inteligência Artificial”?

# A “sala chinesa” de Searle

- Um humano, que apenas fala uma língua ocidental, documentado com um conjunto de regras escritas num livro nessa língua, e dispondo de folhas de papel, está fechado numa sala.
- Através de uma abertura na sala, o humano recebe folhas de papel com símbolos indecifráveis.
- De acordo com as regras, e em função do que recebe, o humano escreve outros símbolos (que igualmente desconhece) nas folhas brancas e envia-as para o exterior da sala.
- No exterior, no entanto, o que se observa é folhas de papel com mensagens escritas em caracteres chineses a serem introduzidas na sala e respostas inteligentes a essas mensagens a serem devolvidas do interior da sala.

# O argumento de Searle

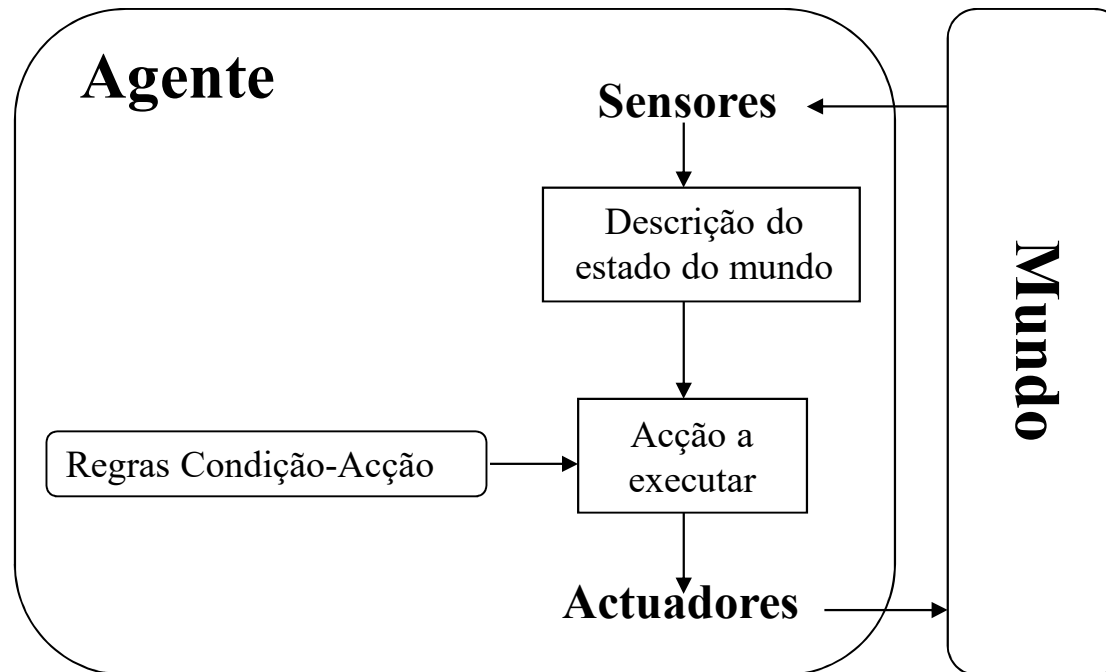
- O humano não percebe chinês
- A sala não percebe chinês
- O livro de regras e as folhas de papel também não percebem chinês
- Logo, não há qualquer compreensão de chinês naquela sala
  
- No entanto, podemos contra-argumentar: embora, individualmente, os componentes do sistema (a sala, o humano, o livro, as folhas de papel) não compreendam chinês, o sistema no seu conjunto compreende chinês



# Tipos e arquiteturas de agentes

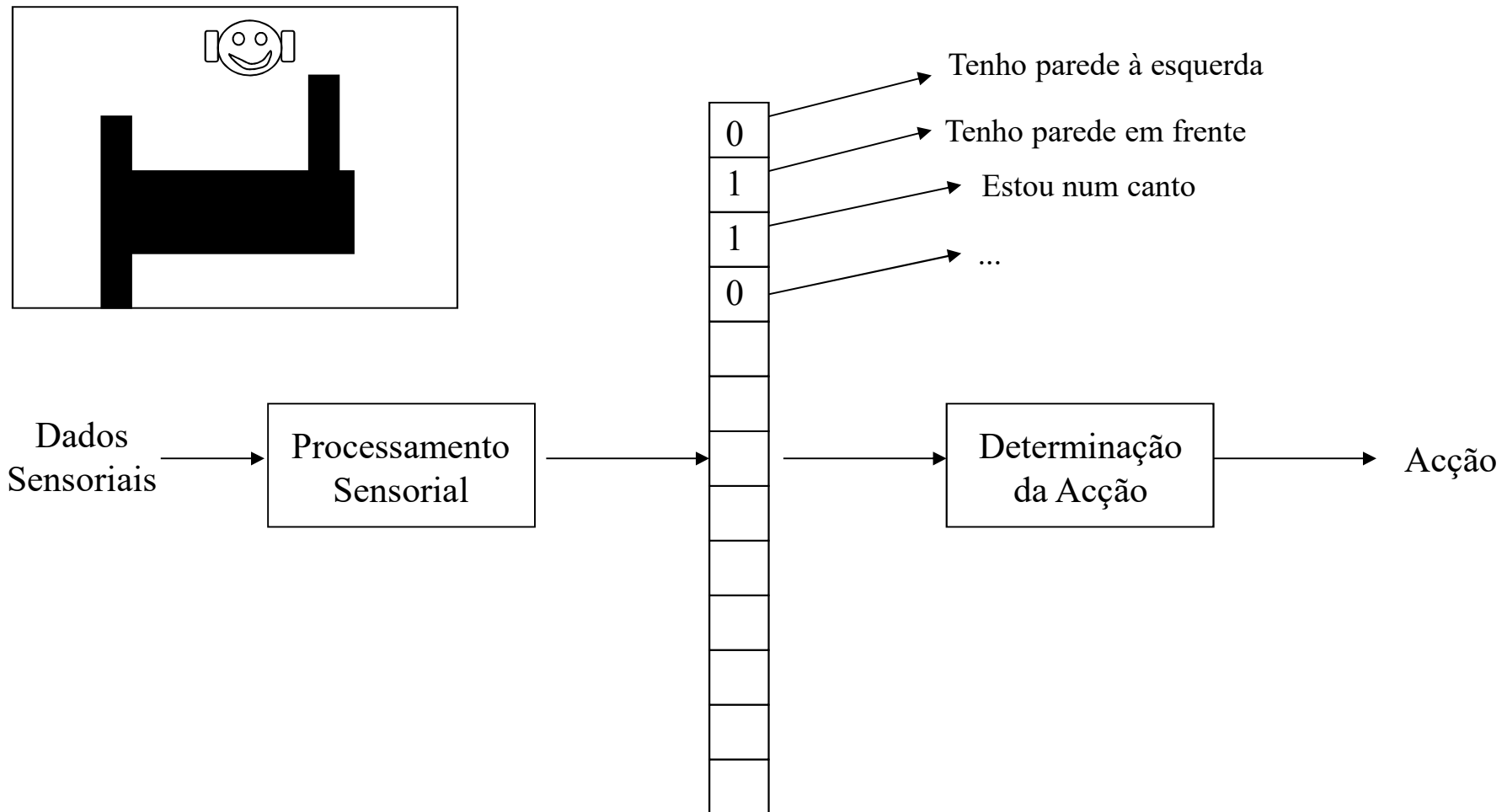
- Tipos de agentes
  - Reativos simples
  - Reativos com estado
  - Deliberativos orientados por objetivos
  - Deliberativos orientados por funções de utilidades
- Arquiteturas
  - Subsunção
  - Três torres
  - Três camadas
  - CARL

# Agente reactivo: simples

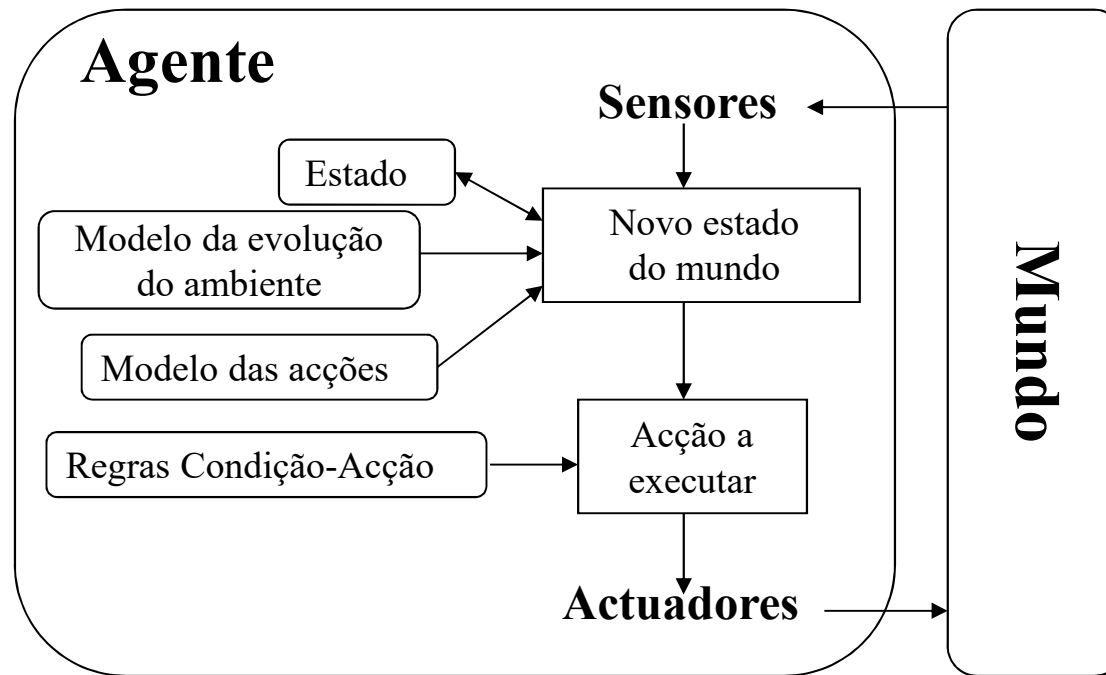


- O conceito de “regra de condição-acção” é também conhecido como “regra de situação-acção” ou “regra de produção”.
- Os agentes ou sistemas reactivos simples são também conhecidos como “sistemas de estímulo-resposta” ou “sistemas de produção”

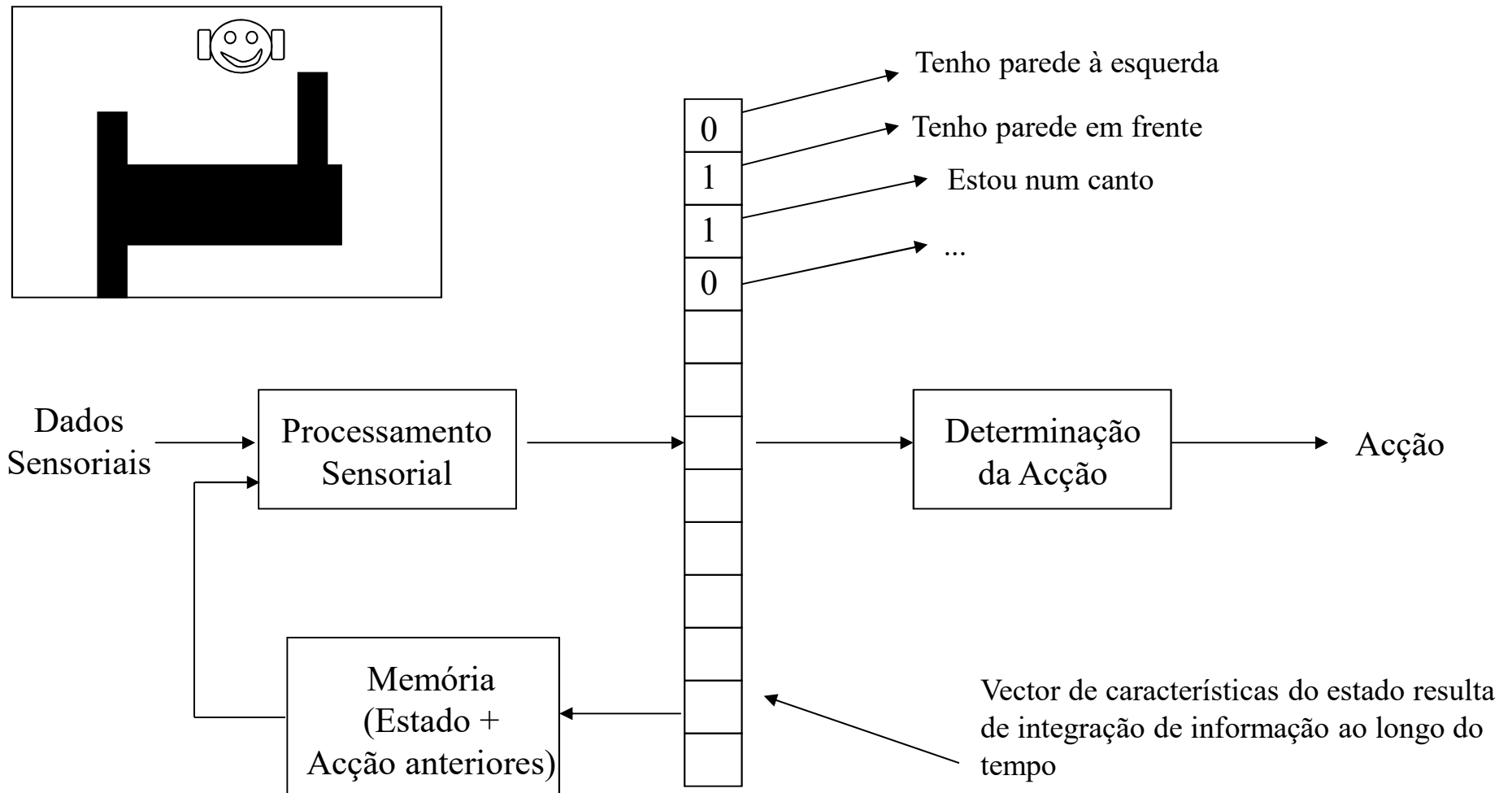
# Representando a percepção através de um vector de características



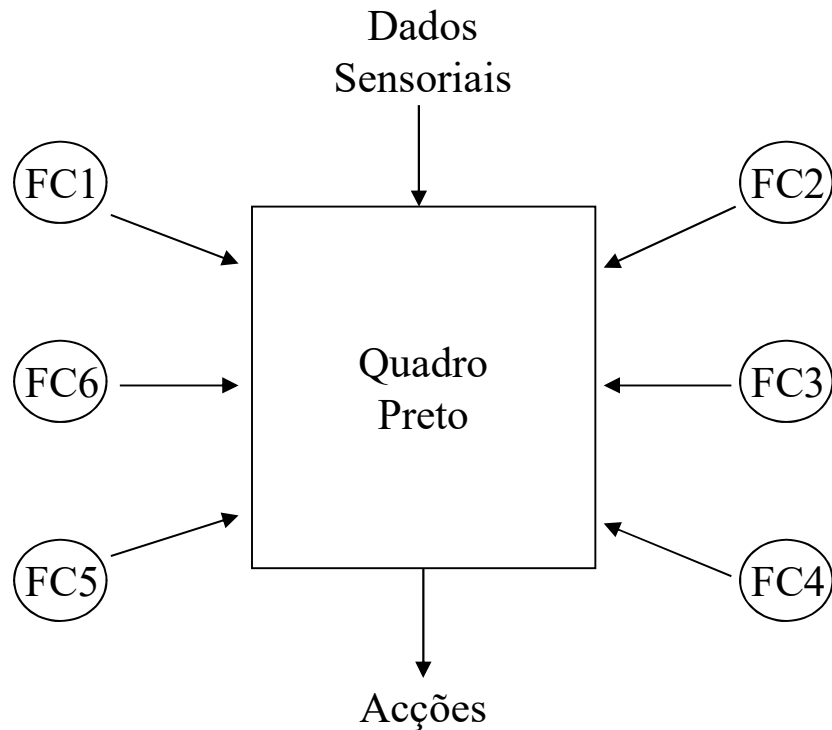
# Agente reactivo: com estado interno



# Representando a percepção através de um vector de características: agente com estado

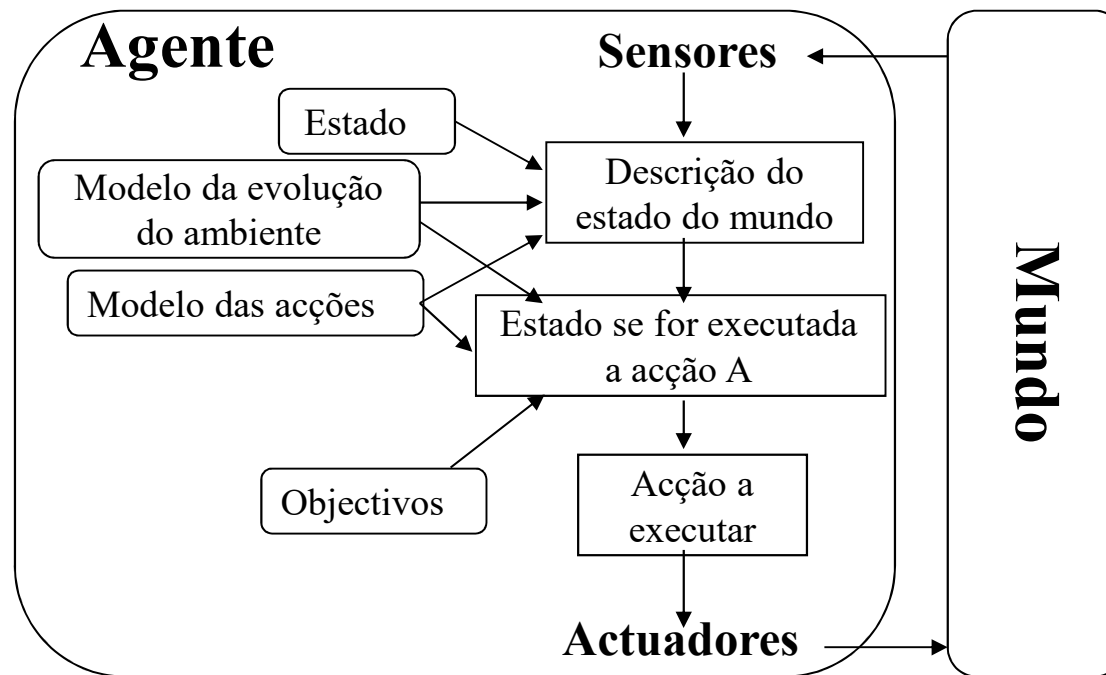


# Sistemas de Quadro Preto

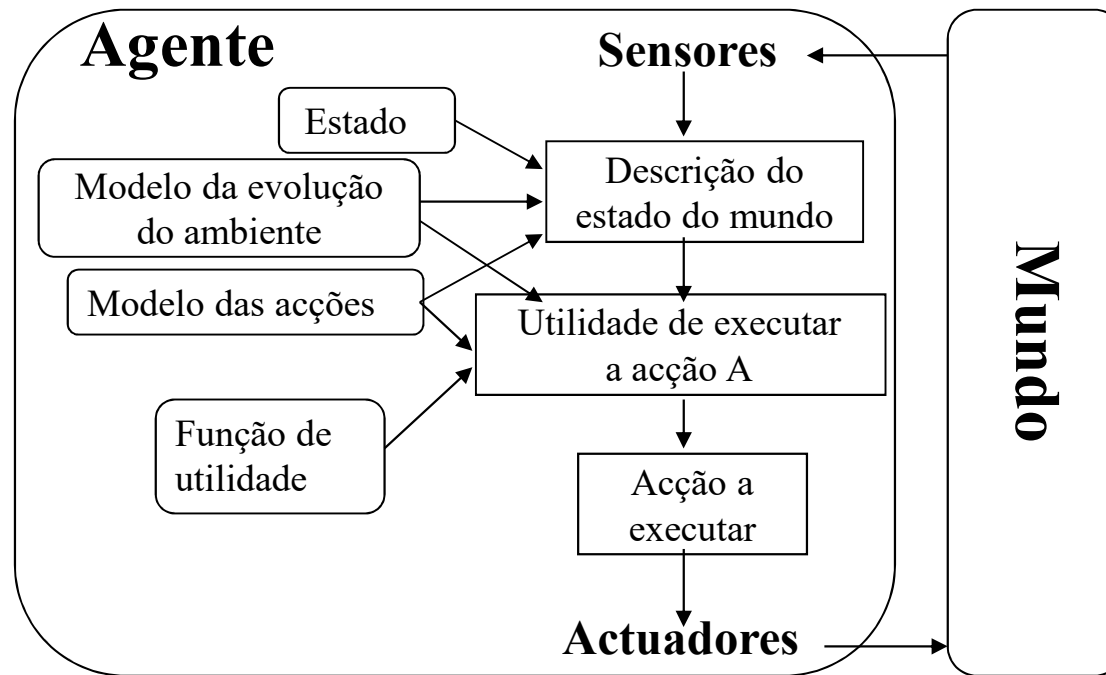


- Podem ser vistos como uma elaboração dos sistemas reactivos com estado interno.
- Uma “fonte de conhecimento” (FC) é um programa que vai fazendo alterações no Quadro Preto.
- Uma FC pode ser vista como um especialista num dado domínio.
- Tipicamente, cada FC rege-se por um conjunto de regras de situação-acção.

# Agente deliberativo: orientado por objectivos



# Agente deliberativo: orientado por função de utilidade





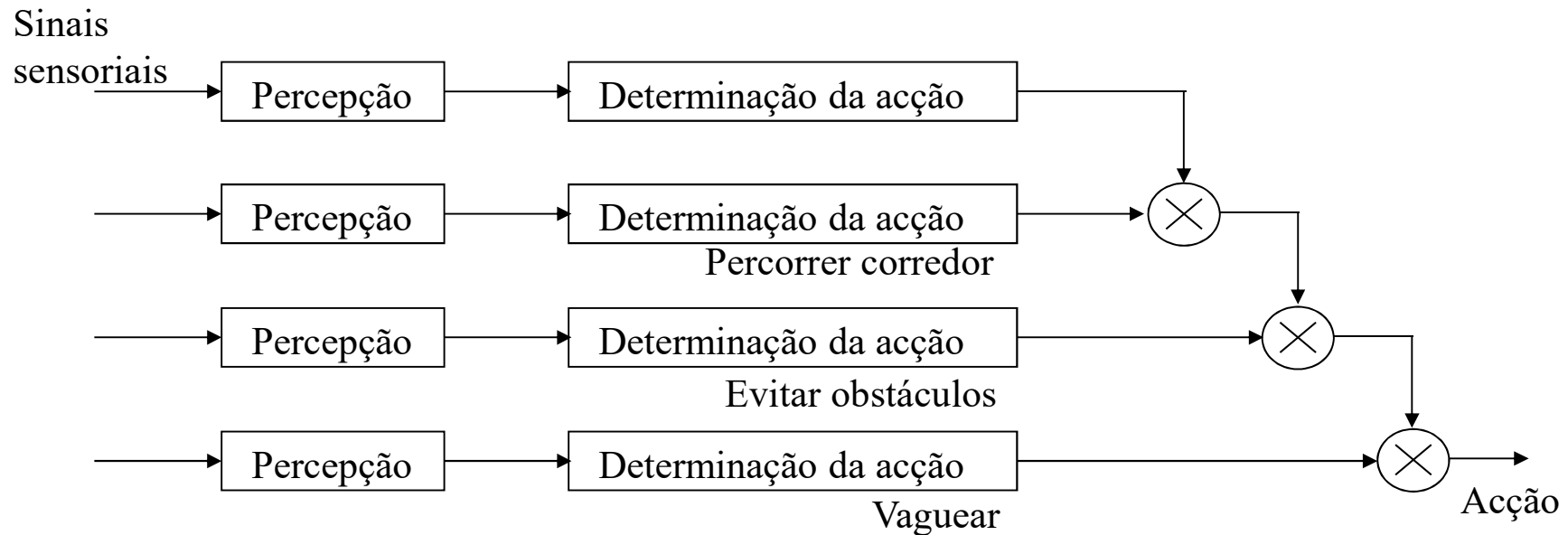
# Propriedades do mundo de um agente

- Acessibilidade – o mundo é “acessível” se os sensores do agente permitem obter uma descrição completa do estado do mundo; o mundo será “efectivamente acessível” se é possível obter toda a informação relevante ao processo de escolha das acções.
- Determinismo – o mundo é “determinístico” se o estado resultante da execução de uma acção é totalmente determinado pelo estado actual e pelos efeitos esperados da acção.
- Mundo episódico – no caso em que cada episódio de percepção-acção é totalmente independente dos outros.
- Dinamismo – o mundo é “dinâmico” se o seu estado pode mudar enquanto o agente delibera; caso contrário, o mundo diz-se “estático”.
- Continuidade – o mundo é “contínuo” quando a evolução do estado do mundo é um processo contínuo ou sem saltos; caso contrário o mundo diz-se “discreto”.

# Mundo de um agente: Exemplos

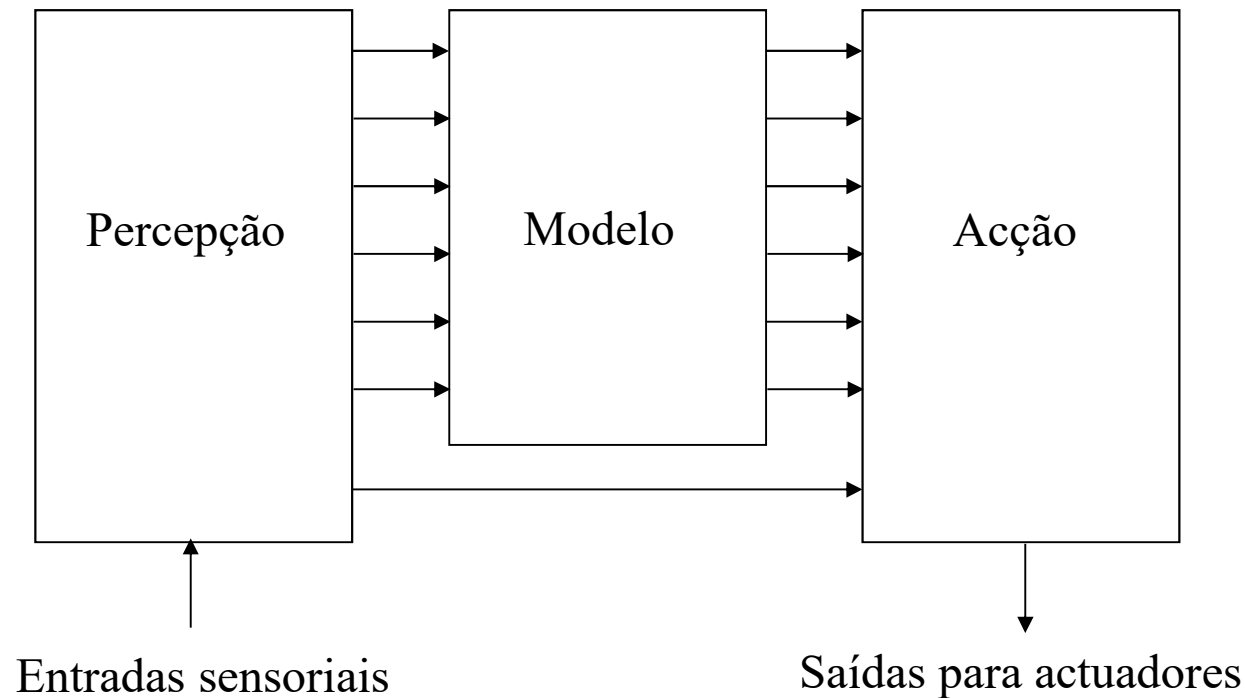
| Mundo                        | Acessível | Determinístico | Episódico | Dinâmico | Contínuo |
|------------------------------|-----------|----------------|-----------|----------|----------|
| Xadrês s/ relógio            | Sim       | Sim            | Não       | Não      | Não      |
| Xadrês c/ relógio            | Sim       | Sim            | Não       | Semi     | Não      |
| Poker                        | Não       | Não            | Não       | Não      | Não      |
| Condução de carro            | Não       | Não            | Não       | Sim      | Sim      |
| Diagnóstico médico           | Não       | Não            | Não       | Não      | Sim      |
| Sistema de análise de imagem | Sim       | Sim            | Sim       | Semi     | Sim      |
| Manipulação robótica         | Não       | Não            | Sim       | Sim      | Sim      |
| Controlo de refinaria        | Não       | Não            | Não       | Sim      | Sim      |
| Tutor de Inglês interactivo  | Não       | Não            | Não       | Sim      | Não      |

# Arquitecturas de agentes: Subsunção

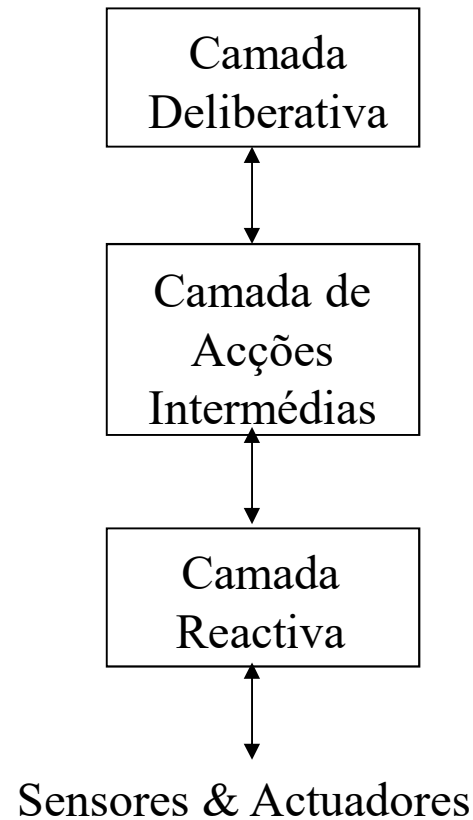


- A arquitectura de subsunção [Brooks,1986; Connell,1990] procura estabelecer a ligação entre percepção e acção a vários níveis – daqui resulta uma organização em camadas.
- A camada mais baixa é a mais reactiva.
- O peso da componente deliberativa aumenta à medida que se sobe na estrutura de camadas.

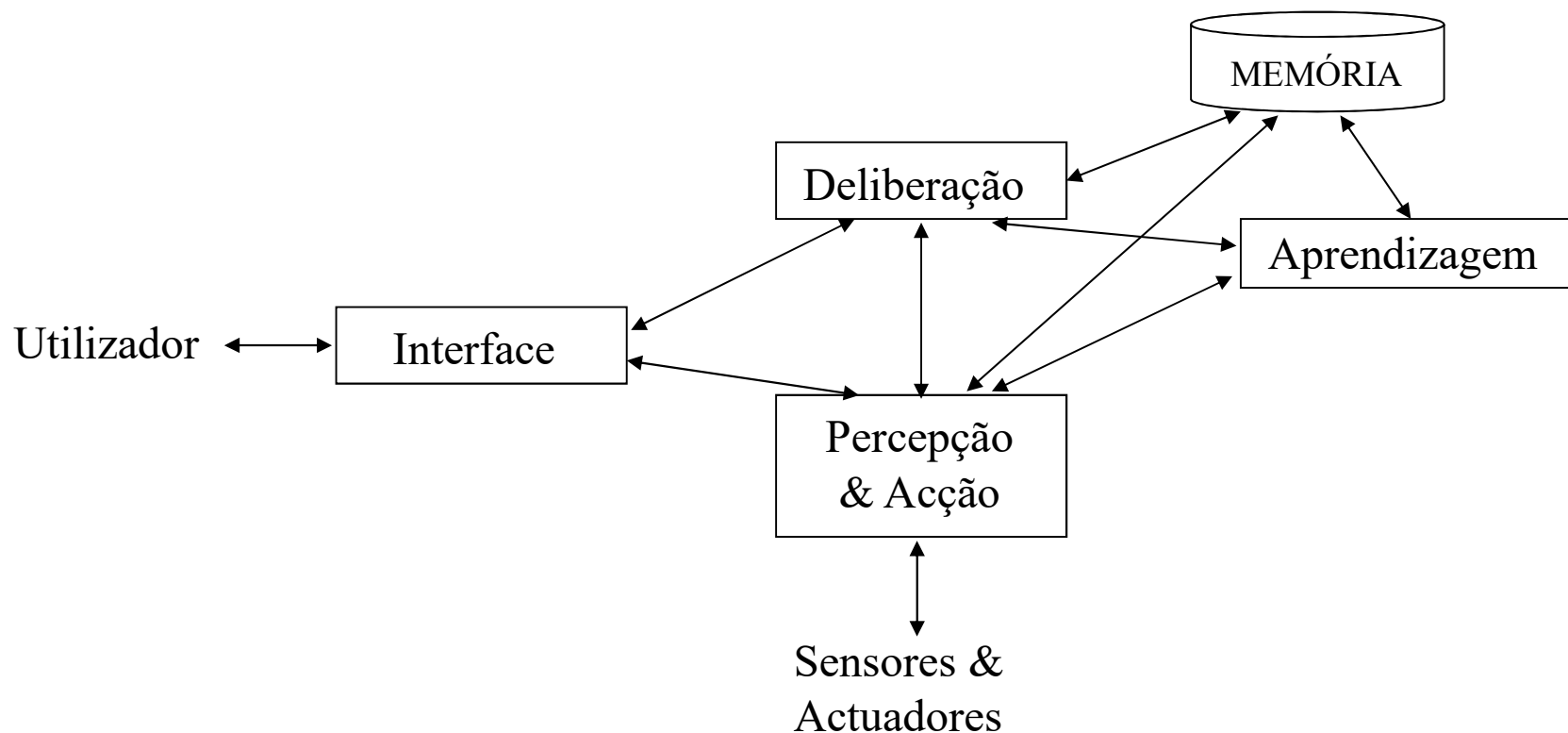
# Arquitecturas de Agentes: Três Torres



# Arquitecturas de Agentes: Três Camadas



# Arquitecturas de Agentes: CARL



# Tópicos de Inteligência Artificial

- Agentes
  - Noção de agente
  - Objectivo da Inteligência Artificial
  - Agentes reactivos e deliberativos
  - Propriedades do mundo de um agente
  - Arquitecturas de agentes
- Representação do conhecimento
- Técnicas de resolução de problemas

# Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistemas de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Exemplo para aulas práticas
- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes



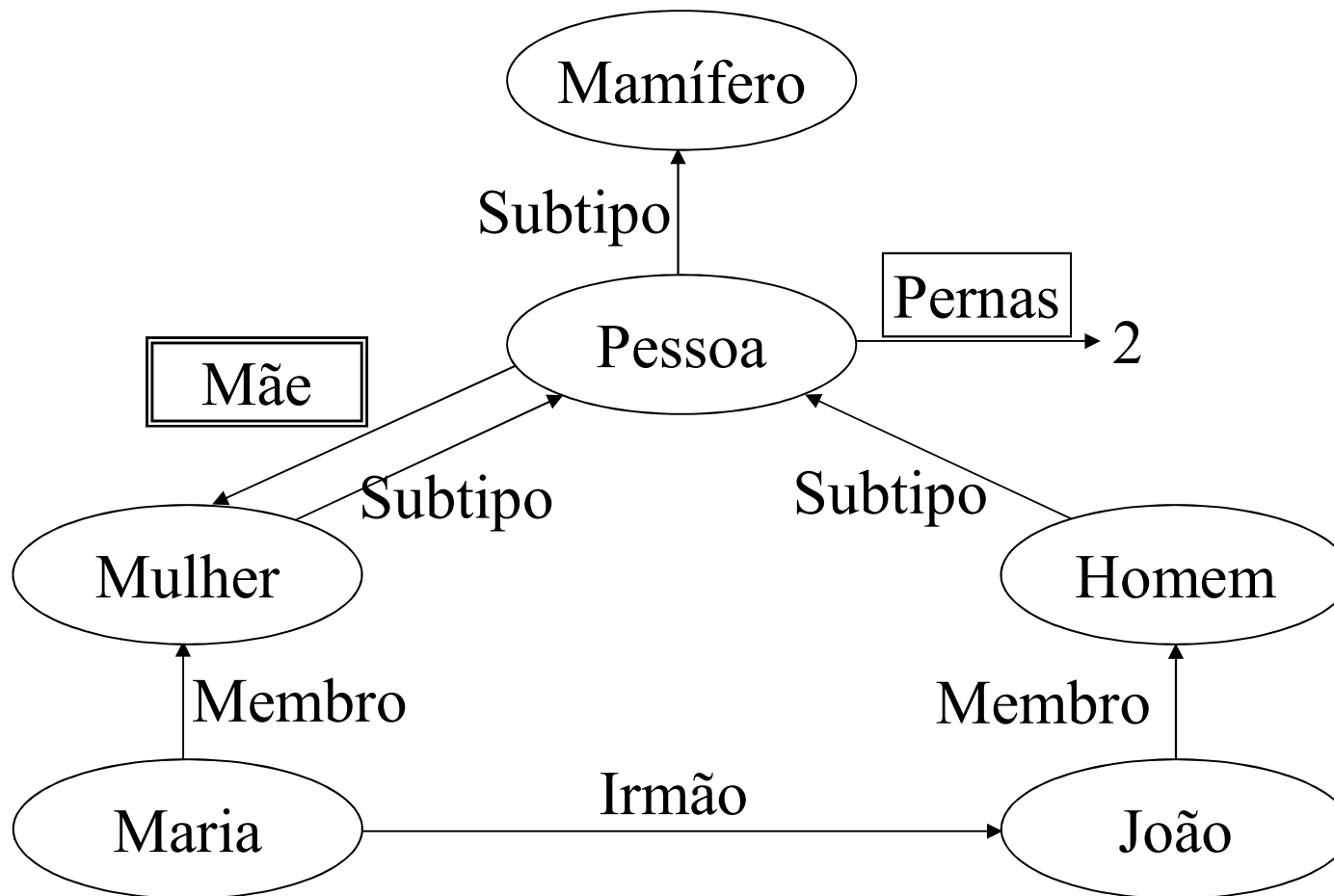
# Redes Semânticas

- Redes semânticas são representações gráficas do conhecimento
- Têm a vantagem da legibilidade
- As redes semânticas podem ser tão expressivas quanto a lógica de primeira ordem

# Redes Semânticas – tipos de relações

- *Sub-tipo* (ou *sub-conjunto* ou ainda *sub-classe*):
  - $A \subset B$
- *Membro* (ou *instância*):
  - $A \in B$
- Relação objecto-objecto:
  - $R(A,B)$
- Relação conjunto-objecto:
  - $\forall x \ x \in A \Rightarrow R(x,B)$
- Relação conjunto-conjunto:
  - $\forall x \ x \in A \Rightarrow \exists y \ (y \in B \wedge R(x,y))$

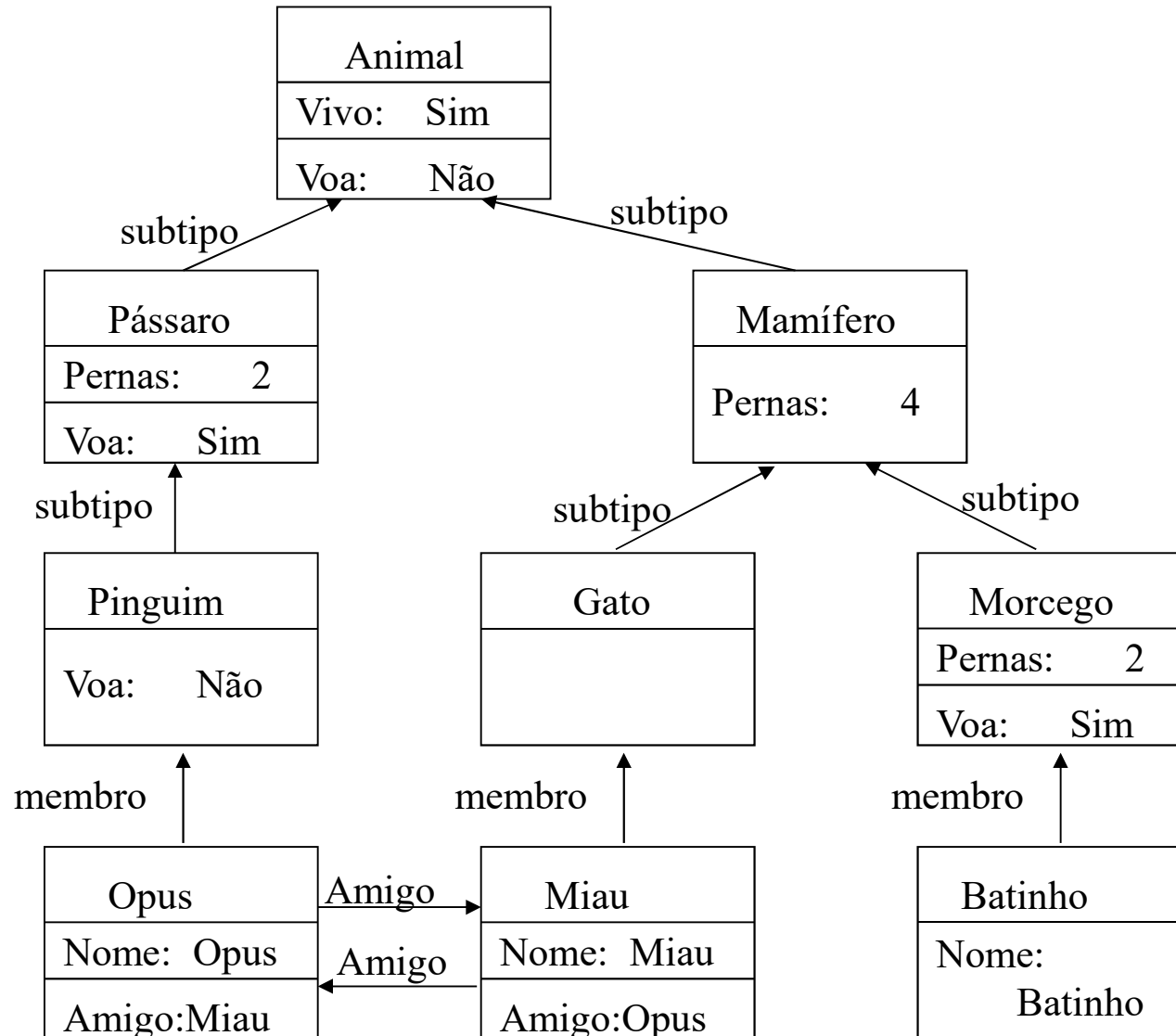
# Redes semânticas – exemplo



# Redes Semânticas - herança

- As relações de *sub-tipo* e *membro* permitem a herança de propriedades:
  - O sub-tipo herda todas as propriedades dos tipos mais abstractos dos quais descende
  - A instância herda todas as propriedades do tipo a que pertence
- A inferência pode ser vista como o seguimento das ligações entre entidades com vista à herança de propriedades
- Pode implementar-se raciocínio não monotónico através do estabelecimento de valores por defeito e o correspondente cancelamento da herança. (ver exemplo)

# Redes Semânticas - exemplo



# Redes Semânticas – Métodos e Demónios

- Normalmente, por razões computacionais, usam-se redes semânticas bastante menos expressivas do que a lógica de primeira ordem
- Deixa-se de lado:
  - Negação
  - Disjunção
  - Quantificação
- Em contra-partida, nomeadamente nos chamados sistemas de *frames*, usam-se métodos e demónios:
  - *Métodos* têm uma semântica similar à da programação orientada por objectos.
  - *Demónios* são procedimentos cuja execução é disparada automaticamente quando certas operações de leitura ou escrita são efectuadas.

# GOLOG – Um gestor de objectos em Prolog

- O GOLOG é um gestor de objectos cuja semântica é próxima das *frames* (Seabra Lopes, 1995)
- Algumas primitivas:
  - new\_frame(Frame)
  - new\_slot(Slot)
  - new\_value(Frame,Slot,Value)
  - new\_relation(Rel,Trans,Restrictions,Inv)
    - Trans ::= transitive | intransitive
    - Restrictions ::= all | none | inclusion(Slots) | exclusion(Slots)
  - call\_method(Frame,Method,ParamList)
  - new\_demon(Frame,Slot,Proc,Access,When,Effect).
    - Access ::= if\_read | if\_write | if\_delete | if\_execute
    - When ::= before | after
    - Effect ::= alter\_value | side\_effect

# UML / Diagramas de Classes - 1

- Na linguagem gráfica UML (*Unified Modelling Language*), os diagramas de classes definem as relações existentes entre as diferentes classes de objectos num dado domínio
  - **Classe** – descrição de um conjunto de objectos que partilham os mesmos atributos, operações, relações e semântica; estes objectos podem ser:
    - Objectos físicos
    - Conceitos que não tenham uma existência palpável
  - **Atributo** – uma propriedade de uma classe
  - **Operação (ou método)** – é a implementação de um serviço que pode ser executado por qualquer objecto da classe
  - **Instância** de uma classe – um objecto que pertence a essa classe, ou seja, constitui uma concretização dessa classe
    - As instâncias de uma classe diferenciam-se pelos valores dos atributos
    - As instâncias “herdam” todos os atributos e métodos da sua classe

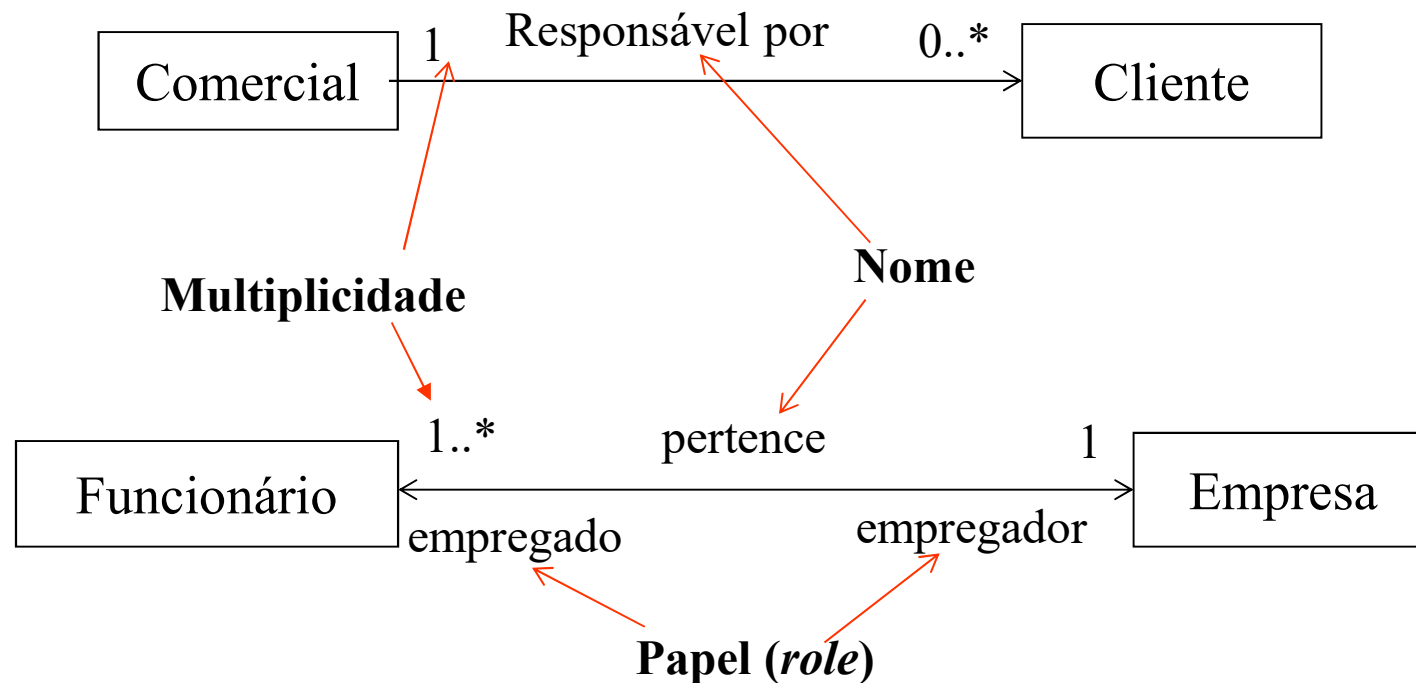


# UML / Diagramas de Classes – 2:

## Relações

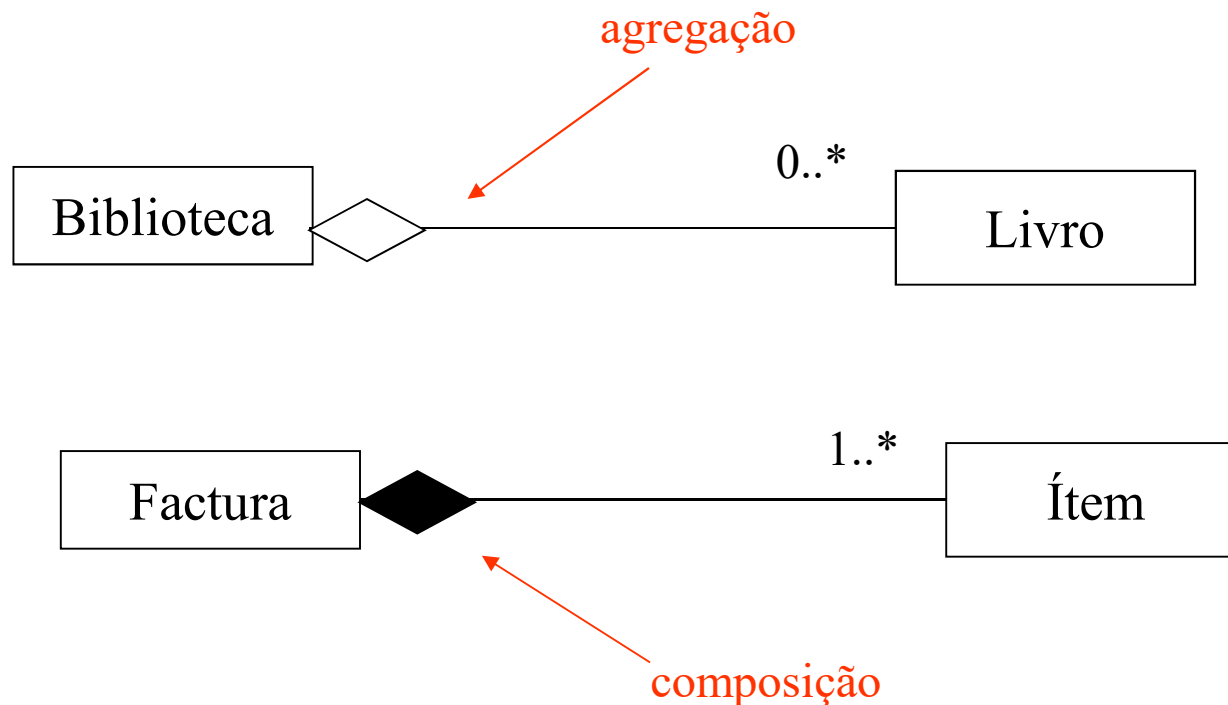
- Um aluno lê um livro
  - **Associação** : classe A “usa a”/ “interage com” classe B
- Um recibo tem vários itens
  - **Composição**: relação todo-parte
- Uma biblioteca tem vários livros
  - **Agregação**: representa relação estrutural entre instâncias de duas classes, em que a classe agregada existe independentemente da outra
- Um aluno é uma pessoa
  - **Generalização**: classe A generaliza classe B e B especializa A (super-classe/sub-classe)

# UML / Diagramas de Classes – 3: Associação

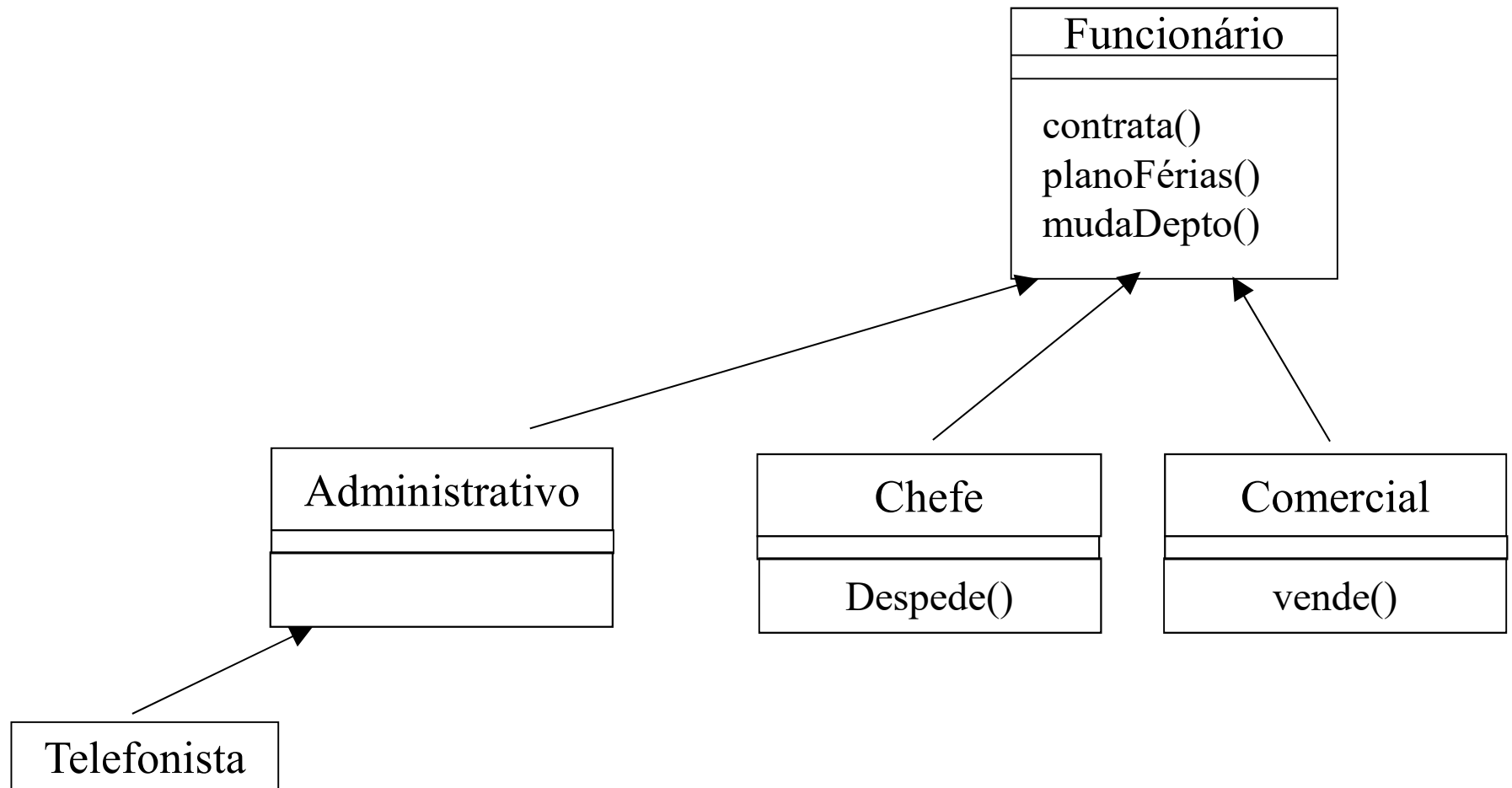


# UML / Diagramas de Classes – 4:

## Agregação e Composição



# UML / Diagramas de Classes – 5: Generalização



# Redes semânticas *versus* UML

| <u>Redes semânticas</u> | <u>UML</u>  |
|-------------------------|---|
| subtipo(SubTipo,Tipo)   | Generalização em diagramas de classes                       |
| membro(Obj,Tipo)        | Diagramas de objectos                                       |
| Relação Objecto/Objecto | Associação, agregação e composição em diagramas de objectos |
| Relação Objecto/Tipo    | não tem   |
| Relação Tipo/Tipo       | Associação, agregação e composição em diagramas de classes  |

# Indução versus Dedução

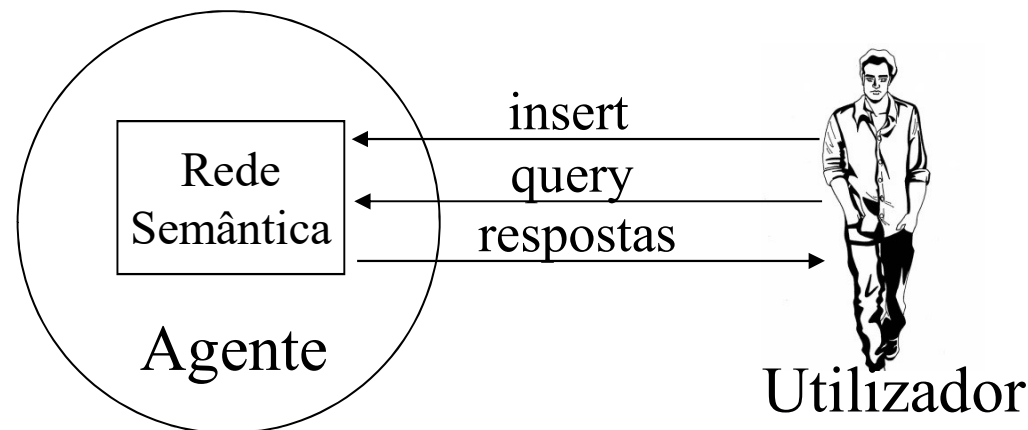
- Dedução – permite inferir casos particulares a partir de regras gerais
  - Preserva a verdade
  - As regras de inferência apresentadas anteriormente são regras dedutivas
- Indução – é o oposto da dedução; permite inferir regras gerais a partir de casos particulares
  - É a base principal da aprendizagem

# Indução

- Exemplo:
  - Casos conhecidos
    - O gato Tareco gosta de leite
    - O gato Pirata gosta de leite
  - Regra inferida
    - Os gatos (normalmente) gostam de leite
  - Nas redes semânticas, a indução pode ser vista como uma “herança de baixo para cima”

# Redes Semânticas em Python

- Vamos criar uma rede semântica, definida como um conjunto de declarações
- Cada declaração associa uma relação semântica ao indivíduo que a declarou
  - Declaration(user,relation)

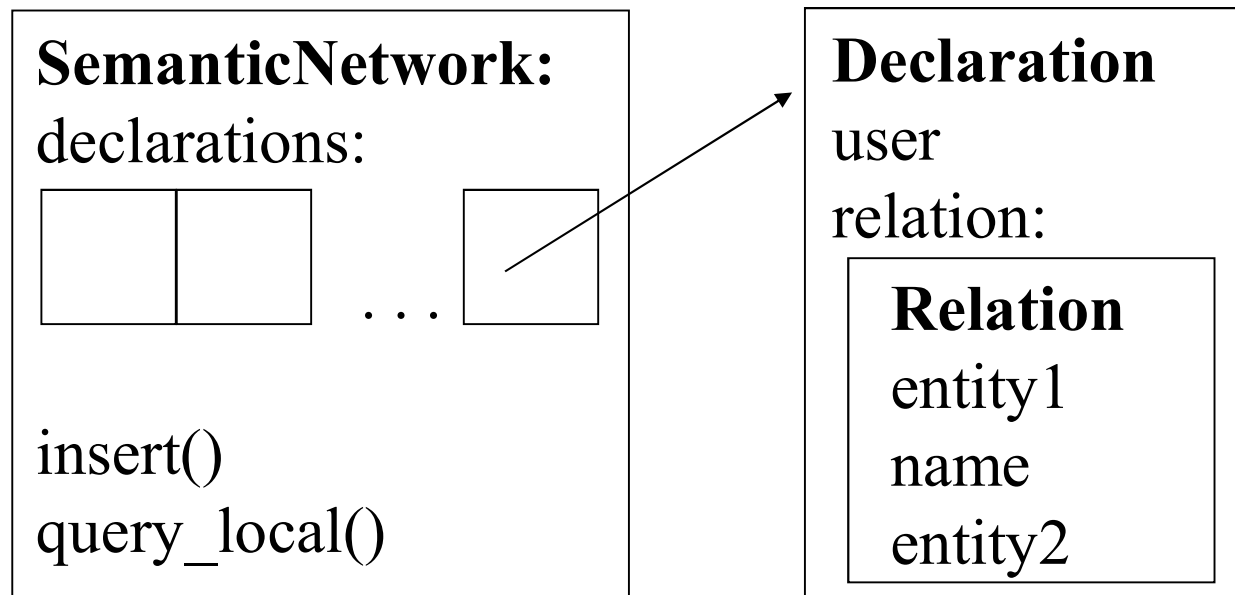




# Redes Semânticas em Python

- Uma relação pode ser dos três tipos seguintes:
  - `Member(obj, type)` – um objecto é membro de um tipo
  - `Subtype(subtype, supertype)` – um tipo é subtipo de outro
  - `Association(entity1, name, entity2)` – uma entidade (objecto ou tipo) está associada a outra
- Operações principais:
  - `insert` – introduzir uma nova declaração
  - `query_local` – questionar a rede semântica sobre as declarações existentes
- Através da introdução incremental de declarações por diferentes interlocutores, emulamos de forma simplificada um processo de aprendizagem, em que o conhecimento é adquirido através da interacção com outros agentes

# Redes Semânticas em Python



- Nota: ver módulo usado nas aulas práticas

# Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistemas de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Implementação em Python
- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes

# Lógicas

- Uma lógica tem:
  - Síntaxe - descreve o conjunto de frases ou fórmulas que é possível escrever.
    - Nota: Estas são as fórmulas bem formadas ou WFF (do inglês *Well Formed Formula*)
  - Semântica - estabelece a relação entre as frases escritas nessa linguagem e os factos que representam.
    - Exemplo: a semântica da lógica proposicional é definida através de tabelas de verdade.
  - Regras de inferência - permitem manipular as frases, gerando umas a partir das outras; as regras de inferência são a base do processo de raciocínio.

# Lógica Proposicional

- Baseada em proposições
  - *Proposição* = frase declarativa elementar que pode ser verdadeira ou falsa
  - Exemplos:
    - “A neve é branca”
    - “O açúcar é um hidrocarbono”
  - Variável proposicional = uma variável que toma o valor de verdade de uma dada proposição
- Uma fórmula em lógica proposicional é composta por uma ou mais variáveis proposicionais ligadas por conectivas lógicas
  - Uma frase proposicional elementar é um frase composta por uma única variável proposicional

# Lógica de Primeira Ordem

- Componentes:
  - *Objectos* ou *entidades*
    - Exemplos: 1215, DDinis, Aveiro
  - *Expressões funcionais*
    - Exemplos: Potencia(4,3), Pai-de(Paulo)
    - Nota 1: Os objectos podem ser considerados como expressões funcionais cuja aridade é zero
    - Nota 2: A noção de *termo* engloba quer os objectos quer as expressões funcionais
  - *Predicados* ou *relações*
    - Exemplos: Pai(Rui, Paulo), Irmão(Paulo,Rosa)
    - Nota: Por definição, os argumentos de um predicado são termos.
- Aqui, as frases elementares são os predicados

# Conectivas Lógicas

- Servem para combinar frases lógicas elementares por forma a obter frases mais complexas
- As conectivas lógicas mais comuns são as seguintes:
  - $\wedge$  (conjunção)
  - $\vee$  (disjunção)
  - $\Rightarrow$  (implicação)
  - $\neg$  (negação)

# Variáveis, Quantificadores

- Na lógica de primeira ordem, os argumentos dos predicatos podem ser variáveis, usadas para representar termos não especificados
  - Exemplos:  $x$ ,  $y$ ,  $pos$ ,  $soma$ ,  $pai$ , ...
- Quantificação universal
  - $\forall x A \equiv$  ‘Qualquer que seja  $x$ , a fórmula  $A$  é verdadeira’
  - Se  $A$  é uma fórmula bem formada, então  $\forall x A$  também é uma fórmula bem formada.
- Quantificação existencial
  - $\exists x A \equiv$  ‘Existe um  $x$ , para o qual a fórmula  $A$  é verdade’
  - Se  $A$  é uma fórmula bem formada, então  $\exists x A$  também é uma fórmula bem formada.



# Lógica de Primeira Ordem - Gramática

*Fórmula*  $\rightarrow$  *FórmulaAtômica*

| *Fórmula Conectiva* *Formula*

| *Quantificador Variável, ... Fórmula*

| '¬' *Fórmula*

| '(' *Fórmula* ')'

*FórmulaAtômica*  $\rightarrow$  *Predicado* '(' *Termo* ',' ... ') | *Termo* '=' *Termo*

*Termo*  $\rightarrow$  *Função* '(' *Termo* ',' ... ') | *Constante* | *Variável*

*Conectiva*  $\rightarrow$  '⇒' | '∧' | '∨' | '⇔'

*Quantificador*  $\rightarrow$  '∃' | '∀'

*Constante*  $\rightarrow$  A | X1 | Paula | ...

*Variável*  $\rightarrow$  a | x | s | ...

*Predicado*  $\rightarrow$  Portista | Cor | ...

*Função*  $\rightarrow$  Registo | Mãe | ...

# Exemplos

- “Todos em Oxford são espertos”:
  - $\forall x \text{ Estuda}(x, \text{Oxford}) \Rightarrow \text{Esperto}(x)$
  - Erro comum: Usar  $\wedge$  em vez de  $\Rightarrow$   
 $\forall x \text{ Estuda}(x, \text{Oxford}) \wedge \text{Esperto}(x)$   
Significa “Todos estão em Oxford e todos são espertos”
- “Alguém em Oxford é esperto”:
  - $\exists x \text{ Estuda}(x, \text{Oxford}) \wedge \text{Esperto}(x)$
  - Erro comum: Usar  $\Rightarrow$  em vez de  $\wedge$   
 $\exists x \text{ Estuda}(x, \text{Oxford}) \Rightarrow \text{Esperto}(x)$   
qualquer estudante de outra universidade forneceria uma interpretação verdadeira.
- “Existe uma pessoa que gosta de toda a gente”
  - $\exists x \forall y \text{ Gosta}(x, y)$

# Interpretações em Lógica Proposicional

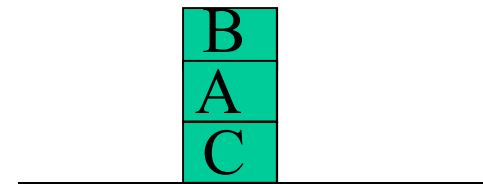
- Na lógica proposicional, uma interpretação de uma fórmula é uma atribuição de valores de verdade ou falsidade às várias proposições que nela ocorrem
  - Exemplo: a fórmula  $A \wedge B$  tem quatro interpretações possíveis.
- Satisfatibilidade - Uma interpretação satisfaz uma fórmula se a fórmula toma o valor ‘verdadeiro’ para essa interpretação.
- Modelo de uma fórmula - uma interpretação que satisfaz essa fórmula.
- Tautologia - uma fórmula cujo valor é ‘verdadeiro’ em qualquer interpretação.

# Interpretações em Lógica de Primeira Ordem

- Uma interpretação de uma fórmula em lógica de primeira ordem é o estabelecimento de uma correspondência entre as várias constantes que ocorrem na fórmula e os objectos do mundo, funções e relações que essas constantes representam.

– Exemplo:

- Objectos: A, B, C, Chão
- Funções: nenhuma
- Relações:
  - Em\_cima\_de: { <B,A>, <A,C>, <C,Chão> }
  - Livre: { <B> }
- Assumindo o estado dado pela figura, esta interpretação constitui um modelo



# Lógica - Regras de Substituição - I

- São válidas quer na lógica proposicional quer na lógica de primeira ordem
- Leis de DeMorgan
$$\neg (A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg (A \vee B) \equiv \neg A \wedge \neg B$$
- Dupla negação:
$$\neg \neg A \equiv A$$
- Definição da implicação:
$$A \Rightarrow B \equiv \neg A \vee B$$
- Transposição:
$$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$$

# Lógica - Regras de Substituição - II

- Comutação

$$A \wedge B \equiv B \wedge A$$

$$A \vee B \equiv B \vee A$$

- Associação:

$$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$$

$$(A \vee B) \vee C \equiv A \vee (B \vee C)$$

- Distribuição:

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

# Lógica - Regras de Substituição - III

- Leis de DeMorgan generalizadas (estas são específicas da lógica de primeira ordem):

$$\neg(\forall x P(x)) \equiv \exists x \neg P(x)$$

$$\neg(\exists x P(x)) \equiv \forall x \neg P(x)$$

# Exercícios

- Representar as seguintes frases em lógica de primeira ordem:
  - Só um aluno chumbou a História
  - Nem todos os estudantes se inscreveram simultaneamente a *Introdução à Inteligência Artificial e Sistemas Inteligentes*
  - A melhor nota a História foi mais elevada do que a melhor nota a Biologia
  - Todos os Portistas gostam de Pinto da Costa
  - Existe um Sportinguista que gosta de todos os Benfiquistas que não são espertos
  - Existe um Barbeiro que barbeia toda a gente menos ele próprio



# CNF e Forma Clausal

- Uma fórmula na *forma normal conjuntiva* (abreviado *CNF*, de *Conjunctive Normal Form*) é uma fórmula que consiste de uma conjunção de cláusulas.
- Uma *cláusula* é uma fórmula que consiste de uma disjunção de literais.
- Um *literal* é uma fórmula atômica (literal positivo) ou a negação de uma fórmula atômica (literal negativo).
  - Nota: na lógica proposicional uma fórmula atômica é uma proposição.
- *Forma clausal* é a representação de uma fórmula CNF através do conjunto das respectivas cláusulas

# Conversão de uma Fórmula Proposicional para CNF e forma clausal

- Através dos seguintes passos:
  - Remover implicações
  - Reduzir o âmbito de aplicação das negações
  - Associar e distribuir até obter a forma CNF
- Exemplo:
  - Fórmula original:  $A \Rightarrow (B \wedge C)$
  - Após remoção de implicações:  $\neg A \vee (B \wedge C)$
  - Forma CNF:  $(\neg A \vee B) \wedge (\neg A \vee C)$
  - Forma clausal:  $\{ \neg A \vee B, \neg A \vee C \}$

# Conversão para forma clausal em Lógica de Primeira Ordem - I

- Renomear variáveis, de forma a que cada quantificador tenha uma variável diferentes
- Remover as implicações
- Reduzir o âmbito das negações, ou seja, aplicar a negação
- Para estas transformações, aplicar as regras de substituição já apresentadas

## Exemplo

Fórmula original:

$$\forall x \forall y \neg( p(x,y) \Rightarrow \forall y q(y,y) )$$

Variáveis renomeadas:

$$\forall a \forall b \neg( p(a,b) \Rightarrow \forall c q(c,c) )$$

Implicações removidas:

$$\forall a \forall b \neg(\neg p(a,b) \vee \forall c q(c,c) )$$

Negações aplicadas:

$$\forall a \forall b ( p(a,b) \wedge \exists c \neg q(c,c) )$$

# Conversão para forma clausal em Lógica de Primeira Ordem - II

- Skolemização
  - Nome dado à eliminação dos quantificadores existenciais
  - Substituir todas as ocorrências de cada variável quantificada existencialmente por uma função cujos argumentos são as variáveis dos quantificadores universais exteriores
- Remover quantificadores universais

Exemplo (cont.)

Skolemizada aplicada:

$$\forall a \forall b ( p(a,b) \wedge \neg q(f(a,b), f(a,b)) )$$

Quantificadores removidos:

$$p(a,b) \wedge \neg q(f(a,b), f(a,b))$$

# Conversão para forma clausal em Lógica de Primeira Ordem - III

- Converter para CNF
  - Usar as regras de substituição relativas à comutação, associação e distribuição
- Converter para a forma clausal, ou seja, eliminar conjunções
- Renomear variáveis de forma a que uma variável não apareça em mais do que uma fórmula

Exemplo (cont.)

Convertida para a forma clausal:  
 $\{ p(a,b) , \neg q(f(a,b), f(a,b)) \}$

Variáveis renomeadas:  
 $\{ p(a_1,b_1) , \neg q(f(a_2,b_2), f(a_2,b_2)) \}$

# Lógica - Regras de Inferência

- Modus Ponens:  $\{ A, A \Rightarrow B \} \vdash B$
- Modus Tolens:  $\{ \neg B, A \Rightarrow B \} \vdash \neg A$
- Silogismo hipotético:  $\{ A \Rightarrow B, B \Rightarrow C \} \vdash A \Rightarrow C$
- Conjunção:  $\{ A, B \} \vdash A \wedge B$
- Eliminação da conjunção:  $\{ A \wedge B \} \vdash A$
- Disjunção:  $\{ A, B \} \vdash A \vee B$
- Silogismo disjuntivo (ou resolução unitária):  
 $\{ A \vee B, \neg B \} \vdash A$
- Resolução:  $\{ A \vee B, \neg B \vee C \} \vdash A \vee C$
- Dilema construtivo:  
 $\{ (A \Rightarrow B) \wedge (C \Rightarrow D), A \vee C \} \vdash B \vee D$
- Dilema destrutivo:  
 $\{ (A \Rightarrow B) \wedge (C \Rightarrow D), \neg B \vee \neg D \} \vdash \neg A \vee \neg C$

# Lógica de Primeira Ordem

## - Regras de Inferência específicas

- *Instanciação universal:*

$$\{\forall x P(x)\} \vdash P(A)$$

- *Generalização existencial*

$$\{P(A)\} \vdash \exists x P(x)$$

# Consequências Lógicas, Provas

- Consequência lógica

- Diz-se que  $A$  é consequência lógica do conjunto de fórmulas em  $\Delta$ , e escreve-se

$$\Delta \models A,$$

se  $A$  toma o valor ‘verdadeiro’ em todas as interpretações para as quais cada uma das fórmulas em  $\Delta$  toma também o valor verdadeiro.

- Definição de Prova

- Uma sequência de fórmulas  $\{ A_1, A_2, \dots, A_n \}$  é uma prova (ou dedução) de  $A_n$  a partir de um conjunto de fórmulas  $\Delta$  sse cada uma das fórmulas  $A_i$  está em  $\Delta$  ou pode ser inferida a partir das fórmulas  $A_1 \dots A_{i-1}$ .
- Neste caso escreve-se:  $\Delta \vdash A_n$



# Correcção, Completude

- Correcção - Diz-se que um conjunto de regras de inferência é correcto se todas as fórmulas que gera são consequências lógicas
- Completude - Diz-se que um conjunto de regras de inferência é completo se permite gerar todas as consequências lógicas.
- Um sistema de inferência correcto e completo permite tirar consequências lógicas sem ter que analisar caso a caso as várias interpretações.

# Metateoremas

- Teorema da dedução:
  - Se  $\{ A_1, A_2, \dots, A_n \} \models B$ , então  $A_1 \wedge A_2 \wedge \dots \wedge A_n \Rightarrow B$ , e vice-versa.
- Redução ao absurdo:
  - Se o conjunto de fórmulas  $\Delta$  é satisfatível (logo tem pelo menos um modelo) e  $\Delta \cup \{\neg A\}$  não é satisfatível, então  $\Delta \models A$ .

# Resolução não é Completa

- A resolução é uma regra de inferência correcta (gera fórmulas necessariamente verdadeiras)

$$\{ A \vee B, \neg B \vee C \} \vdash A \vee C$$

- A resolução não é completa.
  - Exemplo - A resolução não consegue derivar a seguinte consequência lógica:

$$\{ A \wedge B \} \models A \vee B$$

# Refutação por Resolução

- A refutação por resolução é um mecanismo de inferência completo
  - Neste caso, usa-se a resolução para provar que a negação da consequência lógica é inconsistente com a premissa (*meta-teorema da redução ao absurdo*).
  - No exemplo dado, prova-se que
$$(A \wedge B) \wedge \neg(A \vee B)$$
é inconsistente (basta mostrar que é possível derivar a fórmula ‘Falso’).
- Passos da refutação por resolução:
  - Converter a premissa e a negação da consequência lógica para um conjunto de cláusulas.
  - Aplicar a resolução até obter a cláusula vazia.

# Substituições, Unificação

- A aplicação da *substituição*  $s = \{ t_1/x_1, \dots, t_n/x_n \}$  a uma fórmula  $W$  denota-se  $SUBST(W,s)$  ou  $Ws$ ; Significa que todas as ocorrências das variáveis  $x_1, \dots, x_n$  em  $W$  são substituídas pelos termos  $t_1, \dots, t_n$
- Duas fórmulas  $A$  e  $B$  são unificáveis se existe uma substituição  $s$  tal que  $As = Bs$ . Nesse caso, diz-se que  $s$  é uma *substituição unificadora*.
- A *substituição unificadora mais geral* (ou *minimal*) é a mais simples (menos extensa) que permite a unificação.

# Resolução e Refutação na Lógica de Primeira Ordem

- Resolução:  
 $\{ A \vee B, \neg C \vee D \} \vdash \text{SUBST}(A \vee D, g)$   
em que B e C são unificáveis sendo g a sua substituição unificadora mais geral
- A regra da resolução é correcta
- A regra da resolução não é completa
- Tal como na lógica proposicional, também aqui a refutação por resolução é completa

# Resolução com Cláusulas de Horn

- O mecanismo de prova baseado na refutação por resolução é completo e correcto mas não é eficiente (na verdade é NP-completo)
- Uma cláusula de Horn é uma cláusula que tem no máximo um literal positivo
  - Exemplos:

|                             |                      |
|-----------------------------|----------------------|
| $A$                         | $\neg A \vee B$      |
| $\neg A \vee B \vee \neg C$ | $\neg A \vee \neg B$ |
- Existem algoritmos de dedução baseados em cláusulas de Horn cuja complexidade temporal é linear
  - As linguagens Prolog e Mercury baseiam-se em cláusulas de Horn

# Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistemas de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Implementação em Python
- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes



# KIF (= Knowledge Interchange Format)

- Esta é uma linguagem desenhada para representar o conhecimento trocado entre agentes.
  - A motivação para a criação do KIF é similar à que deu origem a outros formatos de representação, como o PostScript.
- Pode ser usada também para representar os modelos internos de cada agente.
- Características principais:
  - Semântica puramente declarativa (o Prolog é também uma linguagem declarativa, mas a semântica depende em parte do modelo de inferência)
  - Pode ser tão ou mais expressiva quanto a lógica de primeira ordem.
  - Permite a representação de meta-conhecimento (ou seja, conhecimento sobre o conhecimento)

# KIF – características gerais

- O mundo é conceptualizado em termos de objectos e relações entre objectos
- Uma relação é um conjunto arbitrário de listas de objectos.
  - Exemplo: a relação  $<$  é o conjunto de todos os pares  $(x,y)$  em que  $x < y$ .
- O universo de discurso é o conjunto de todos os objectos cuja existência é conhecida, presumida ou suposta.
  - Os objectos podem ser *concretos* ou *abstratos*
  - Os objectos podem ser *primitivos* (não decomponíveis) ou *compostos*

# KIF - Componentes da linguagem

- Caracteres
- Lexemas
  - Lexemas especiais (aqueles que têm um papel pré-definido na própria linguagem)
  - Palavras
  - Códigos de caracteres
  - Blocos de códigos de caracteres
  - Cadeias de caracteres
- Expressões
  - Termos - objectos primitivos ou compostos
  - Frases - expressões com valor lógico
  - Definições - frases verdadeiras por definição

# KIF - termos

- Constante
- Variável individual
- Expressão funcional
  - $(functor\ arg1\ ..\ argn)$
  - $(functor\ arg1\ ..\ argn\ seqvar)$
- Lista
  - $(listof\ t1\ \dots\ tn)$
- Termo lógico
  - $(if\ c1\ t1\ ..\ cn\ tn\ default)$
- Código de caracter, bloco de códigos de caracteres e cadeia de caracteres
- Citação (quotation)
  - $(quote\ lista)$  ou  $'lista$

# KIF - frases

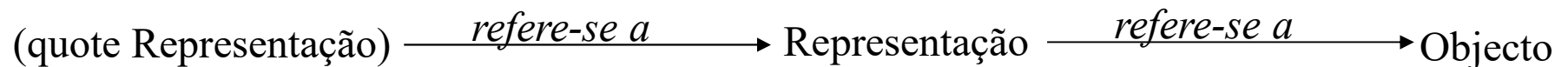
- Constante: true, false
- Equação  
(= *termo1 termo2*)
- Inequação  
(/= *termo1 termo2*)
- Frase relacional  
(*relação t1 .. tn*)
- Frase lógica: construída com as conectivas lógicas ('not', 'and', 'or', '=>', '<=>', '<=>')
- Frase quantificada  
(forall *var1 ... varn frase*)  
(exists *var1 ... varn frase*)

# KIF - definições

- Definição de objectos
  - Igualdade: (defobject  $s := t$ )  
Exemplo: (defobject nil := (listof))
  - Conjunção: (defobject  $s p1 .. pn$ )
  - etc.
- Definição de funções
  - (deffunction  $f(v1 .. vn) := t$ )
  - Exemplo:
    - (deffunction head (?l) := (if (= (listof ?x @items) ?l) ?x))
- Definição de relações (=predicados)
  - (defrelation  $r(v1 .. vn) := p$ )
  - etc.
  - Exemplos:
    - (defrelation null (?l) := (= ?l (listof)))
    - (defrelation list (?x) :=  
(exists (@l) (= ?x (listof (@l)))))

# KIF - meta-conhecimento

- Pode formalizar-se conhecimento sobre o conhecimento
- O mecanismo da citação (quotation) permite tratar expressões como objectos
- Por exemplo a ocorrência da palavra `joão` numa expressão designará uma pessoa; entretanto a expressão `(quote joão)` ou `'joão` designa a própria palavra `joão` e não o objecto ou pessoa a que ela se refere.
- Outros exemplos:
  - `(acredita joão '(material lua queijo))`
  - `(=> (acredita joão ?p) (acredita ana ?p))`
- Graficamente, podemos ilustrar da forma seguinte:



# KIF - dimensões de conformação

- KIF é uma linguagem altamente expressiva
- No entanto, KIF tende a sobrecarregar os sistemas de geração e de inferência
- Por isso, foram definidas várias dimensões de conformação
- Um perfil de conformação é uma selecção de níveis de conformação para cada uma das dimensões referidas



# KIF - perfis de conformação

- Foram definidos os seguintes perfis de conformação:
  - Lógica - atômica, conjuntiva, positiva, lógica, baseada em regras (de Horn ou não, recursivas ou não)
  - Complexidade dos termos - termos simples (constantes e variáveis), termos complexos
  - Ordem - *proposicional*, *primeira ordem* (contem variáveis, mas os funtores e as relações são constantes), *ordem superior* (os funtores e relações podem ser variáveis)
  - Quantificação - conforme se usa ou não
  - Meta-conhecimento - conforme se usa ou não

# Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistemas de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Implementação em Python
- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes

# Engenharia do Conhecimento

- Uma *base de conhecimento* (BC) é um conjunto de representações de *factos* e *regras* de funcionamento do mundo; factos e regras recebem a designação genérica de *frases*.
- Engenharia do conhecimento é o processo ou actividade de construir bases de conhecimento. Isto envolve:
  - Estudar o domínio de aplicação – frequentemente através de entrevistas com peritos (processo de *adquisição de conhecimento*)
  - Determinar os objectos, conceitos e relações que será necessário representar
  - Escolher um vocabulário para entidades, funções e relações (por vezes chamado *ontologia*)
  - Codificar conhecimento genérico sobre o domínio (um conjunto de *axiomas*)
  - Codificar descrições para problemas concretos, interrogar o sistema e obter respostas.
  - Por vezes o domínio é tão complexo que não é praticável codificar à mão todo o conhecimento necessário. Neste caso usa-se *aprendizagem automática*.

# Identificação de objectos, conceitos e relações - 1

- Na modelação em análise de sistemas e engenharia de software coloca-se o mesmo problema
  - Assim, para um problema complexo de representação do conhecimento, não é descabido seguir uma metodologia de análise em boa parte similar às que se usam nos sistemas de informação
- Algumas das palavras que usamos para descrever um domínio em linguagem natural dão naturalmente origem a nomes de objectos, conceitos e relações
  - Substantivos comuns → *conceitos* (também chamados *classes* ou *tipos*)
  - Substantivos próprios → *objectos* (também chamados *instâncias*)
  - Verbo “ser” → pode indicar uma relação de *instanciação* (entre objecto e tipo) ou de *generalização* (entre subtipo e tipo)
  - Verbos “ter” e “conter” → podem indicar uma relação de composição
  - Outros verbos → podem sugerir outras relações relevantes

# Identificação de objectos, conceitos e relações - 2

- Convem avaliar a importância para o problema das palavras utilizadas bem como dos objectos, conceitos e relações subjacentes
  - Não considerar substantivos que identifiquem objectos, conceitos ou relações irrelevantes para o problema
  - Quando vários substantivos aparecem a referir-se ao mesmo conceito, escolher o mais representativo ou adequado
- Um conceito mais abstracto pode ser criado atribuindo-lhe o que é comum a outros dois ou mais conceitos previamente identificados

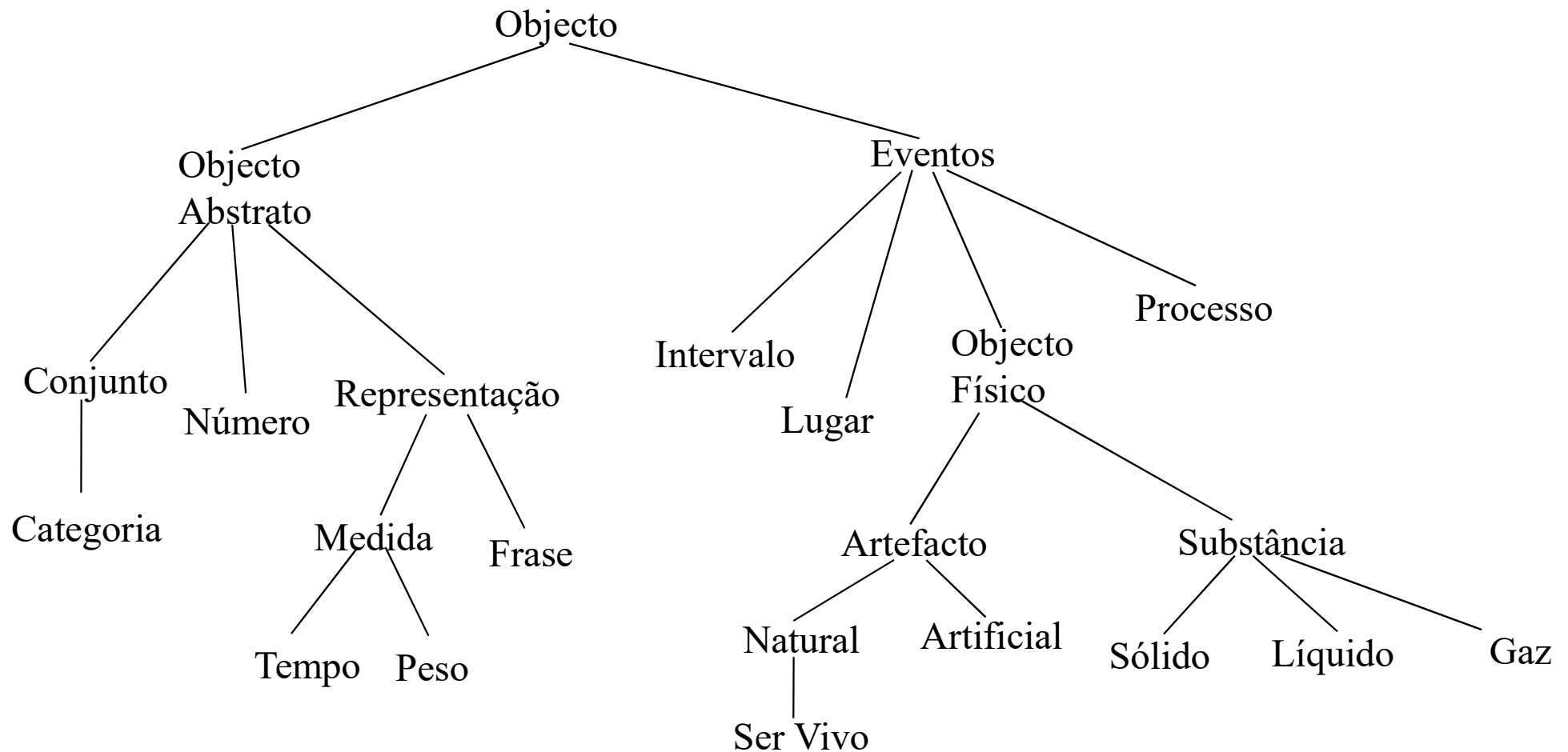
# Ontologias

- Uma ontologia é um vocabulário sobre um domínio conjugado com relações hierárquicas como *membro* e *subtipo* e eventualmente outras.
- O objectivo de uma ontologia é captar a essência da organização do conhecimento num domínio.

# Ontologia Geral

- Uma ontologia geral, aplicável a uma grande variedade de domínios de aplicação, envolve as seguintes noções:
  - Categorias, tipos ou classes
  - Medidas numéricas
  - Objectos compostos
  - Tempo, espaço e mudanças
  - Eventos e processos (eventos contínuos)
  - Objectos físicos
  - Substâncias
  - Objectos abstractos e crenças

# Uma possível ontologia geral





# Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistemas de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Implementação em Python
- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes

# Redes de crença bayesianas

- Também conhecidas simplesmente como “redes de Bayes”
- Permitem representar conhecimento impreciso em termos de um conjunto de variáveis aleatórias e respectivas dependências
  - As dependências são expressas através de probabilidades condicionadas
  - A rede é um grafo dirigido acíclico

# Axiomas das probabilidades

- Para uma qualquer proposição  $a$ , a sua probabilidade é um valor entre 0 e 1:

$$0 \leq P(a) \leq 1$$

- Proposições necessariamente verdadeiras têm probabilidade 1

$$P(\text{true}) = 1$$

- Proposições necessariamente falsas têm probabilidade 0

$$P(\text{false}) = 0$$

- A probabilidade da disjunção é a soma das probabilidades subtraída da probabilidade da intercepção:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

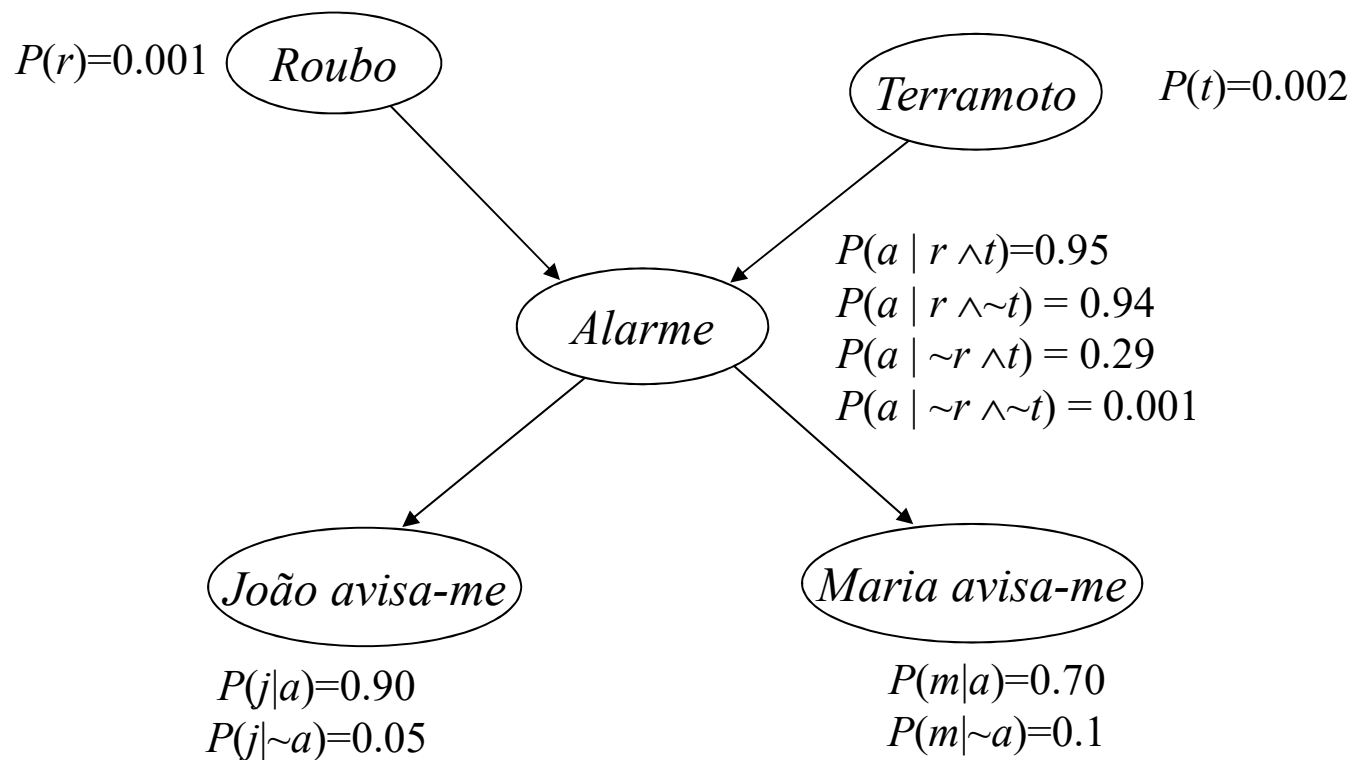
# Probabilidades condicionadas

- Uma probabilidade condicionada  $P(a|b)$  identifica a probabilidade de ser verdadeira a proposição  $a$  na condição de (isto é, sabendo nós que) a proposição  $b$  é verdadeira
- Pode calcular-se da seguinte forma:

$$P(a | b) = \frac{P(a \wedge b)}{P(b)}$$

# Redes de crença bayesianas – exemplo

- Por simplicidade, focamos em variáveis aleatórias booleanas:



# Redes de crença bayesianas – probabilidade conjunta

- A probabilidade conjunta identifica a probabilidade de ocorrer uma dada combinação de valores de todas as variáveis da rede:

$$P(x_1 \wedge \dots \wedge x_n) = \prod_{i=1}^n P(x_i \mid \text{pais}(x_i))$$

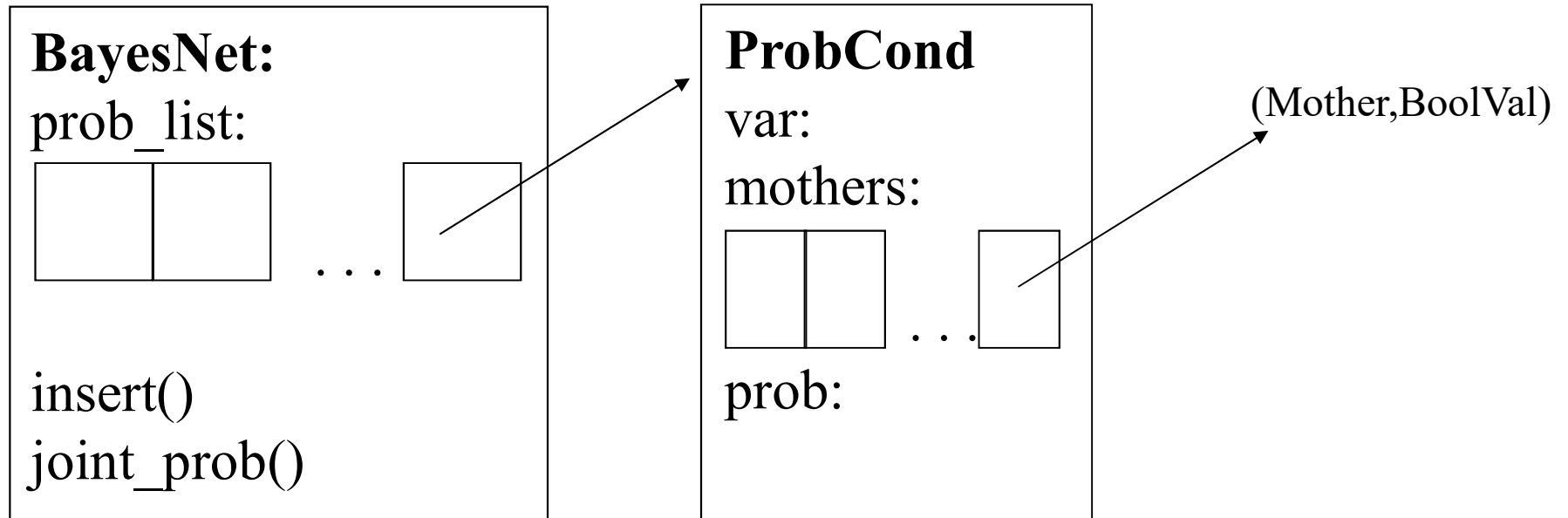
- Assim, no exemplo anterior, a probabilidade de o alarme tocar e o João e a Maria ambos avisarem num cenário em que não há roubo nem terramoto, é dada por:

$$\begin{aligned} &P(j \wedge m \wedge a \wedge \sim t \wedge \sim r) \\ &= P(j \mid a) \times P(m \mid a) \times P(a \mid \sim r \wedge \sim t) \times P(\sim r) \times P(\sim t) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 \\ &= 0.000628 \end{aligned}$$

# Redes Bayesianas em Python

- Vamos criar uma rede de crença bayesiana, representada com base numa lista de probabilidades condicionadas
  - Classe `BayesNet()`
- A probabilidade condicionada de uma dada variável ser verdadeira, dados os valores (`True` ou `False`) das variáveis mães, é representado pela seguinte classe:
  - Classe `ProbCond(var,mother_vals,prob)`
  - Exemplo: `ProbCond("a", [ ("r",True), ("t",True) ], 0.95)`
- Operações principais:
  - `insert` – introduzir uma nova probabilidade condicionada na rede
  - `joint_prob` – obter a probabilidade conjunta para uma dada conjunção de valores de todas as variáveis da rede

# Redes de crença em Python



- Nota: ver módulo usado nas aulas práticas



# Redes de crença bayesianas – probabilidade individual

- A probabilidade individual é a probabilidade de um valor específico (*verdadeiro* ou *falso*) de uma variável
- Calcula-se somando as probabilidades conjuntas das situações em que essa variável tem esse valor específico
- O cálculo das probabilidades conjuntas pode restringir-se à variável considerada e às outras variáveis das quais depende (ascendentes na rede bayesiana)
  - Exemplo: o conjunto dos ascendentes de “João avisa” é { “alarme”, “roubo” e “terramoto” }

# Redes de crença bayesianas – probabilidade individual

$$P(x_i = v_i) = \sum_{\substack{a_j \in \{v, f\} \\ j=1, \dots, k}} P(x_i \wedge a_1 \wedge \dots \wedge a_k)$$

- Seja:
  - $C = \{x_1, \dots, x_n\}$  – conjunto de variáveis da rede
  - $x_i \in C$  – uma qualquer variável da rede
  - $v_i \in \{v, f\}$  – valor de  $x_i$  cuja probabilidade se pretende calcular
  - $\{a_1, \dots, a_k\} \subset C$  – conjunto das variáveis da rede que são ascendentes de  $x_i$

# Tópicos de Inteligência Artificial

- Agentes
- Representação do conhecimento
- Técnicas de resolução de problemas
  - Técnicas de pesquisa em árvore
  - Técnicas de pesquisa em grafo
  - Técnicas de pesquisa por melhorias sucessivas
  - Técnicas de pesquisa com propagação de restrições
  - Técnicas de planeamento

# Resolução de problemas em IA

- Um *problema* é algo (um objectivo) cuja solução não é imediata
- Por isso, a resolução de um problema requer a *pesquisa de uma solução*

# Resolução de problemas em IA

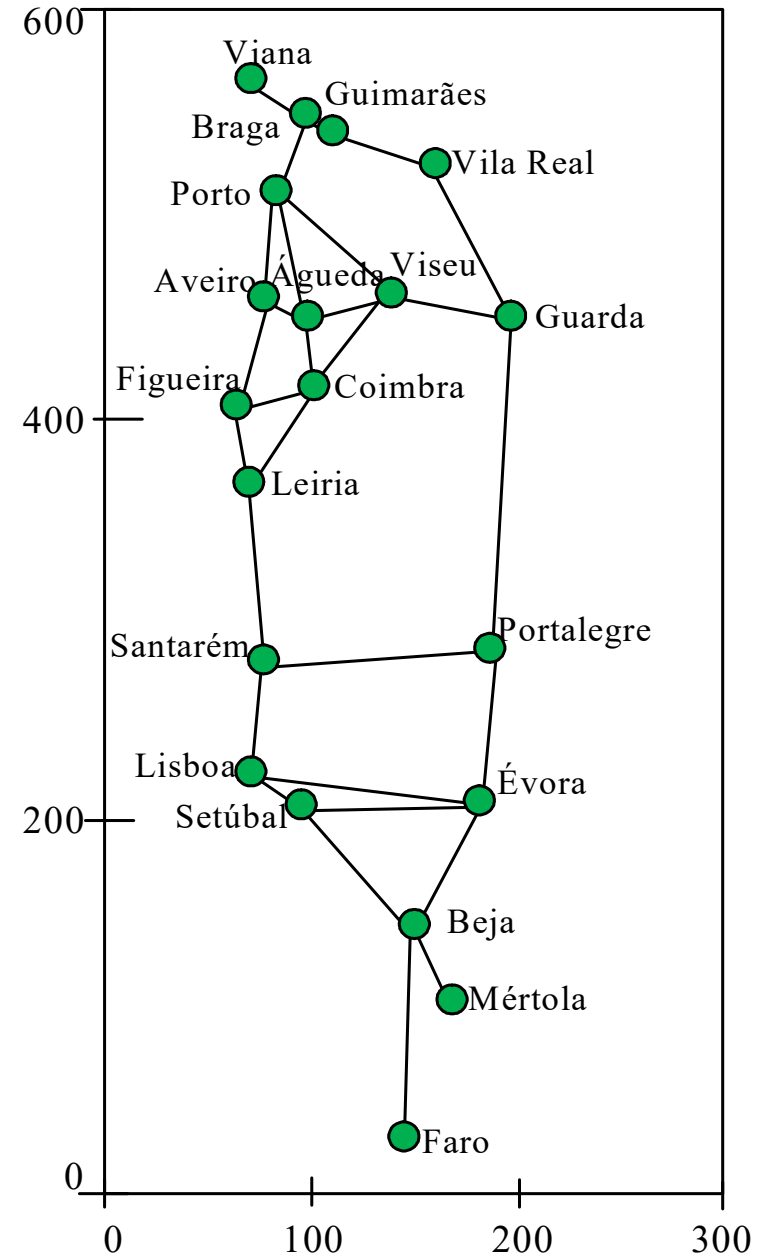
- Um *problema* é algo cuja solução não é imediata
- Exemplos de problemas:
  - Dado um conjunto de axiomas, demonstrar um novo teorema
  - Dado um mapa, determinar o melhor caminho entre dois pontos.
  - Dada uma situação num jogo de xadrez, determinar uma boa jogada.
  - Determinar a melhor distribuição das portas lógicas no circuito VLSI
  - Dada as peças de um produto a montar, determinar a melhor sequência de montagem.

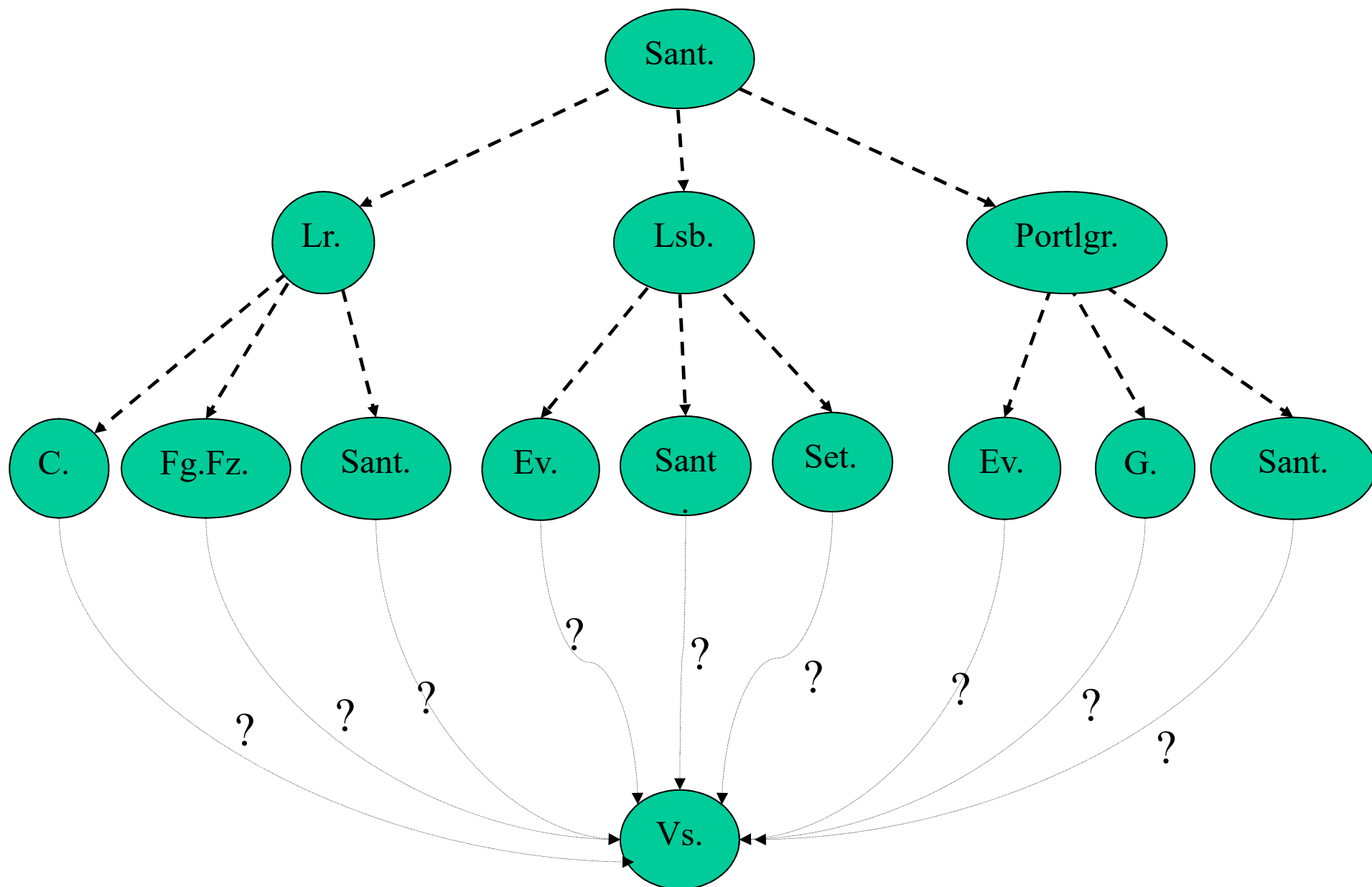
# Formulação de problemas e pesquisa de soluções

- A formulação de um problema inclui:
  - Descrição do ponto de partida – o estado inicial
  - Exemplos
    - A situação no jogo de xadrez
    - A descrição de um mapa e a localização inicial do viajante
  - Um conjunto de transições de estados
  - Um função que diz se um dado estado satisfaz o objectivo
  - Por vezes também uma função que avalia o custo de uma solução
- A pesquisa de uma solução é um processo que, de forma recursiva ou iterativa, vai executando transições de estados até que um estado gerado satisfaça o objectivo.

# Aplicação: determinar um percurso num mapa topológico

- Dados:
  - Distâncias por estrada entre cidades vizinhas
- Exemplo:
  - Determinar um caminho de Santarém para a Viseu







# Estratégias de pesquisa

- Pesquisa em árvore
  - Estratégias de pesquisa cega (não informada):
    - Em largura
    - Em profundidade
    - Em profundidade com limite
    - Em profundidade com limite crescente
  - Estratégias de pesquisa informada
    - Pesquisa A\* e suas variantes (custo uniforme, gulosa)
  - Advanced techniques (graph-search, IDA\*, RBFS, SMA\*)
- Pesquisa com propagação de restrições
- Pesquisa por melhorias sucessivas
- Planeamento

# Pesquisa em árvore – algoritmo genérico

pesquisa(Problema,Estratégia) **retorna** a Solução, ou ‘falhou’

Árvore  $\leftarrow$  árvore de pesquisa inicializada com o estado inicial do Problema

Ciclo:

**se** não há candidatos para expansão, **retornar** ‘falhou’

Folha  $\leftarrow$  uma folha escolhida de acordo com Estratégia

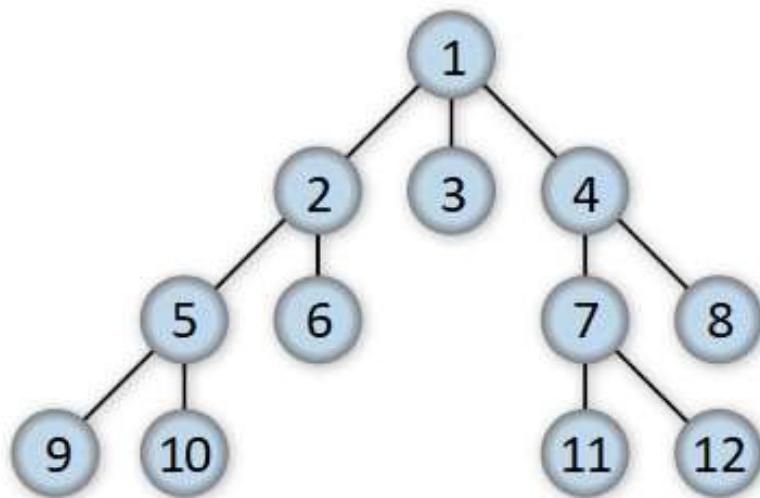
**se** Folha contem um estado que satisfaz o objectivo

**então retornar** a Solução correspondente

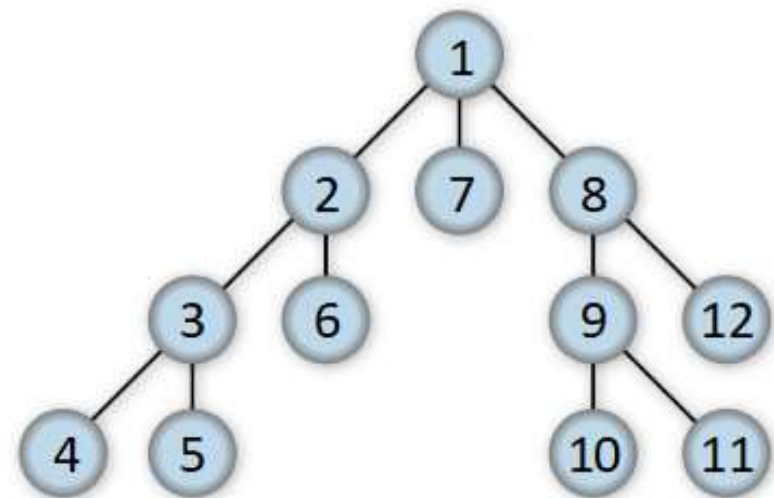
**senão** expandir Folha e adicionar os nós resultantes à Árvore

Fim do ciclo;

# Percursos na árvore de pesquisa



Pesquisa em largura



Pesquisa em profundidade

(crédito das figuras: Alexander Drichel / Wikipedia)

# Pesquisa em árvore – implementação baseada numa fila

pesquisa\_em\_arvore(Problema,AdicionarFila) **retorna** a Solução, ou ‘falhou’

Fila  $\leftarrow$  [ fazer\_nó(estado inicial do Problema) ]

Ciclo

se Fila está vazia, **retornar** ‘falhou’

Nó  $\leftarrow$  remover\_cabeça(Fila)

se estado(Nó) satisfaz o objectivo

**então retornar** a solução(Nó)

**senão** Fila  $\leftarrow$  AdicionarFila(Fila, expansão(Nó))

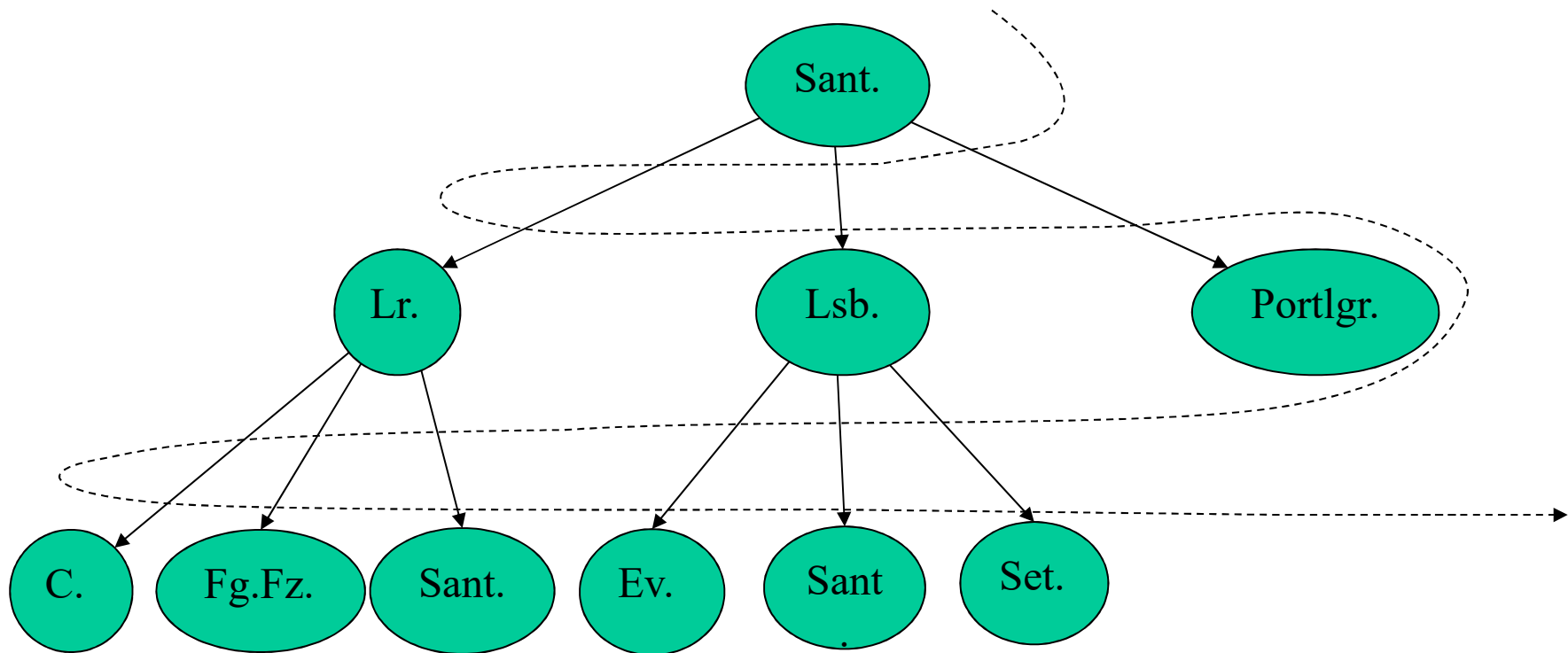
pesquisa\_em\_largura(Problema) **retorna** a Solução, ou ‘falhou’

retornar pesquisa\_em\_arvore(Problema,juntar\_no\_fim)

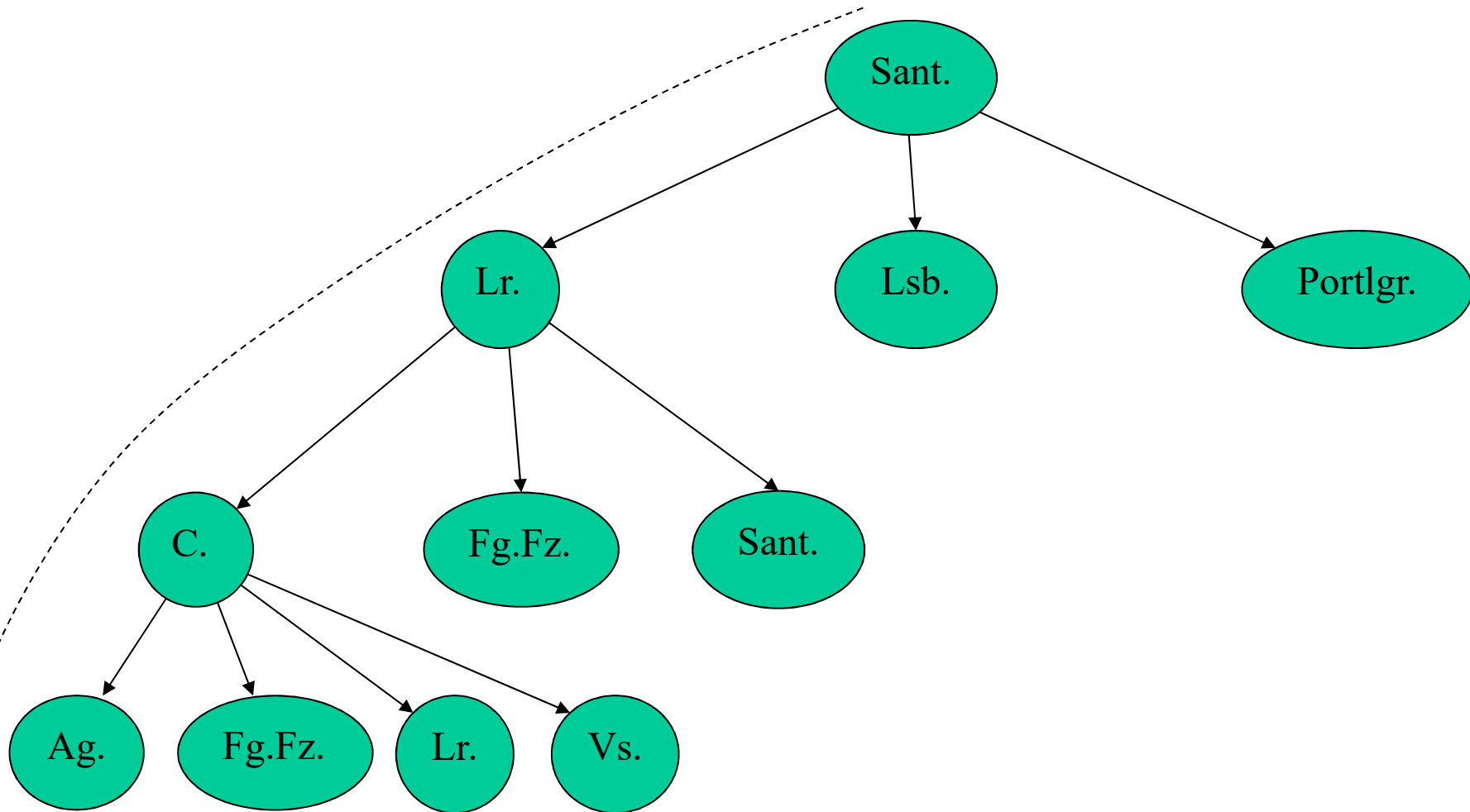
pesquisa\_em\_profundidade(Problema) **retorna** a Solução, ou ‘falhou’

retornar pesquisa\_em\_arvore(Problema,juntar\_à\_cabeça)

# Pesquisa em largura



# Pesquisa em profundidade



# Pesquisa em Árvore em Python

- Vamos criar um conjunto de classes para suporte à resolução de problemas por pesquisa em árvore
  - Classe `SearchDomain()` – classe abstracta que formata a estrutura de um domínio de aplicação
  - Classe `SearchProblem(domain,initial,goal)` – classe para especificação de problemas concretos a resolver
  - Classe `SearchNode(state,parent)` – classe dos nós da árvore de pesquisa
  - Classe `SearchTree(problem)` – classe das árvores de pesquisa, contendo métodos para a geração de uma árvore para um dado problema

# Pesquisa em Árvore em Python

## **SearchTree:**

problem:

### **SearchProblem**

domain:

#### **SearchDomain**

*actions()*

*result()*

*cost()*

*heuristic()*

initial:

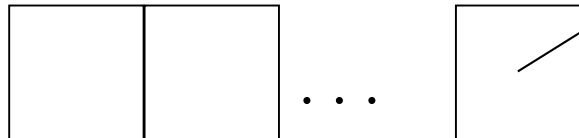
goal:

*test\_goal()*

strategy:

*search()*

open\_nodes:



### **SearchNode**

state:

parent:

- Nota: Ver módulo usado nas aulas práticas



# Pesquisa em profundidade - variantes

- Pesquisa em profundidade *sem repetição de estados* – para evitar ciclos infinitos, convém garantir que estados já visitados no caminho que liga o nó actual à raiz da árvore de pesquisa não são novamente gerados.
- Pesquisa em profundidade *com limite* – não são considerados para expansão os nós da árvore de pesquisa cuja profundidade excede um dado limite.
- Pesquisa em profundidade *com limite crescente* – consiste no seguinte procedimento:
  - 1) Tenta-se resolver o problema por pesquisa em profundidade com um dado limite  $N$
  - 2) Se foi encontrada uma solução, retornar.
  - 3) Incrementar  $N$ .
  - 4) Voltar ao passo 1.

# Pesquisa informada (“melhor primeiro”)

`pesquisa_informada(Problema,FuncAval)` **retorna** a Solução, ou ‘falhou’  
Estratégia  $\leftarrow$  estratégia de gestão de fila de acordo com FuncAval  
`pesquisa_em_arvore(Problema,Estratégia)`

# Avaliação das estratégias de pesquisa

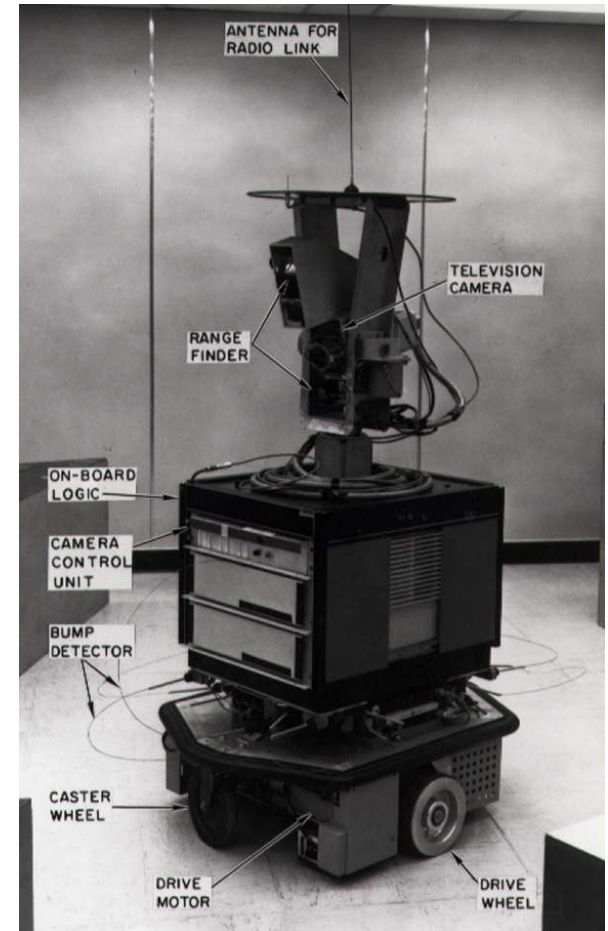
- Compleitude – uma estratégia é completa se é capaz de encontrar uma solução quando existe uma solução
- Complexidade temporal – quanto tempo demora a encontrar a solução
- Complexidade espacial – quanto espaço de memória é necessário para encontrar uma solução
- Optimalidade – a estratégia de pesquisa consegue encontrar melhor solução.

# Pesquisa A\*

- Escolhe-se o nó em que a função de custo total  $f(n)=g(n)+h(n)$  tem o menor valor
  - $g(n)$  = custo desde o nó inicial até ao nó  $n$
  - $h(n)$  = custo estimado desde o nó  $n$  até à solução [heurística]
- A função heurística  $h(n)$  diz-se *admissível* se nunca sobrestima o custo real de chegar a uma solução a partir de  $n$ .
- Se for possível garantir que  $h(n)$  é admissível, então a pesquisa A\* encontra sempre (um)a solução óptima.
- A pesquisa A\* é também completa.

# Shakey the Robot

- A pesquisa A\* foi inventada em 1968 para otimizar o planeamento de caminhos deste robô



# Pesquisa A\* - variantes

- *Pesquisa de custo uniforme*
  - $h(n) = 0$
  - $f(n) = g(n)$
  - É um caso particular da pesquisa A\*
  - Também conhecido como algoritmo de Dijkstra
  - Tem um comportamento parecido com o da pesquisa em largura
  - Caso exista solução, a primeira solução encontrada é ótima
- *Pesquisa gulosa*
  - Ignora custo acumulado  $g(n)$
  - $f(n) = h(n)$
  - Dado que o custo acumulado é ignorado, não é verdadeiramente um caso particular da pesquisa A\*
  - Tem um comportamento que se aproxima da pesquisa em profundidade
  - Ao ignorar o custo acumulado, facilmente deixa escapar a solução ótima

# Pesquisa num grafo de estados - motivação

- Em inglês: “*graph search*”
- Frequentemente, o espaço de estados é um grafo.
- Ou seja, transições a partir de diferentes estados podem levar ao mesmo estado.
- Isto leva a que a pesquisa fique menos eficiente.
- Portanto, o que se deve fazer é memorizar os estados já visitados por forma a evitar tratá-los novamente.
- Memoriza-se apenas o melhor caminho até cada estado

# Pesquisa num grafo de estados

- Tal como no algoritmo anterior, trabalha-se com uma fila de nós
  - Chama-se fila de nós ABERTOS (nós ainda não expandidos, ou folhas)
  - Em cada iteração, o primeiro nó em ABERTOS é seleccionado para expansão
- Adicionalmente, usa-se também uma lista de nós FECHADOS (os já expandidos)
  - Necessário para evitar repetições de estados



# Pesquisa num grafo de estados - algoritmo

- 1. Inicialização
  - $N0 \leftarrow$  nó do estado inicial;  $ABERTOS \leftarrow \{ N0 \}$
  - $FECHADOS \leftarrow \{ \}$
- 2. Se  $ABERTOS = \{ \}$ , então acaba sem sucesso.
- 3. Seja  $N$  o primeiro nó de  $ABERTOS$ .
  - Retirar  $N$  de  $ABERTOS$ .
  - Colocar  $N$  em  $FECHADOS$ .
- 4. Se  $N$  satisfaz o objectivo, então retornar a solução encontrada.
- 5. Expandir  $N$  :
  - $CV \leftarrow$  conjunto dos vizinhos sucessores de  $N$
  - Para cada  $X \in CV - (ABERTOS \cup FECHADOS)$ , ligá-lo ao antecessor directo,  $N$
  - Para cada  $X \in CV \cap (ABERTOS \cup FECHADOS)$ , ligá-lo a  $N$  caso o melhor caminho passe por  $N$
  - Adicionar os novos nós a  $ABERTOS$
  - Reordenar  $ABERTOS$
- 6. Voltar ao passo 2.

# Pesquisa num grafo de estados

- Tal como a pesquisa em árvore, a “pesquisa em grafo” ou “graph search” utiliza uma árvore de pesquisa
- No entanto, a pesquisa em árvore normal ignora a possibilidade de o espaço de estados ser um grafo
  - Mesmo que o espaço de estados seja um grafo, a pesquisa em árvore trata-o como se fosse uma árvore
- Pelo contrário, a pesquisa em grafo leva em conta que o espaço de estados é normalmente um grafo e garante que a árvore de pesquisa não tem mais do que um caminho para cada estado

# Avaliação da pesquisa em árvore

## - factores de ramificação

- Seja:
  - $N$  – número de nós da árvore de pesquisa no momento em que se encontra a solução
  - $X$  – Número de nós expandidos (não terminais)
  - $d$  – comprimento do caminho na árvore correspondente à solução

- *Ramificação média* – número médio de filhos por nó expandido:

$$RM = \frac{N - 1}{X}$$

Nota: a ramificação média é um indicador da dificuldade do problema.

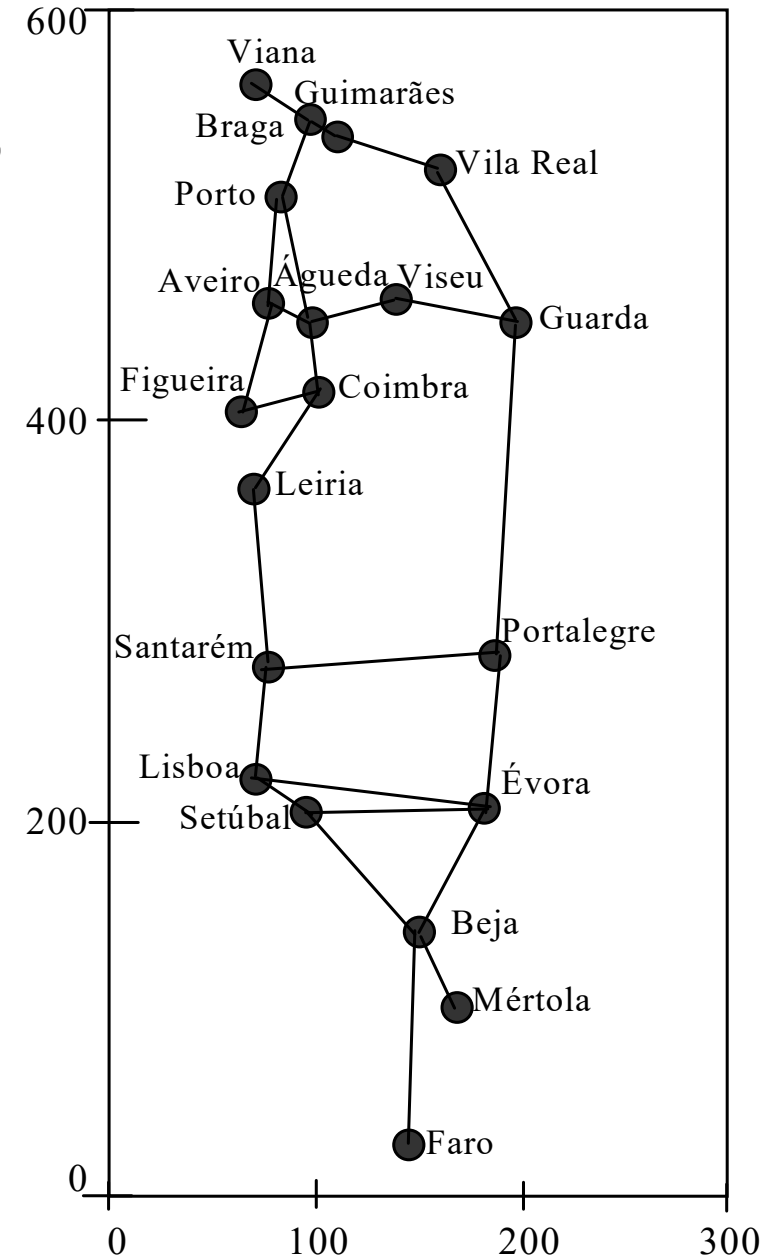
- *Factor de ramificação efectivo* – número de filhos por nó,  $B$ , numa árvore com ramificação constante e com profundidade constante  $d$ . Portanto:

$$1 + B + B^2 + \dots + B^d = N \quad \text{ou seja:} \quad \frac{B^{d+1} - 1}{B - 1} = N \quad (\text{resolve-se por métodos numéricos}).$$

- O factor de ramificação efectiva é um indicador da eficiência da técnica de pesquisa utilizada.

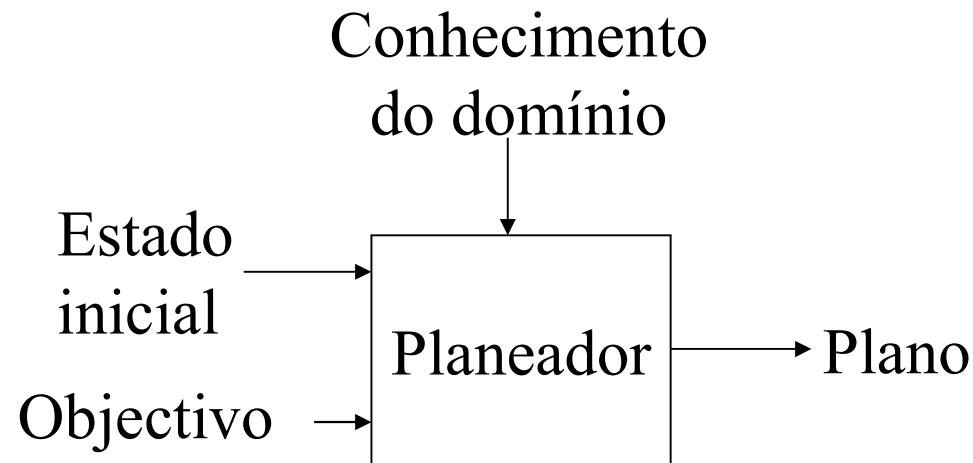
# Aplicação: planejar um passeio turístico

- Dados:
  - Coordenadas entre cidades
  - Distâncias por estrada entre cidades vizinhas
- Calcular:
  - O melhor caminho entre duas cidades.
- Usando:
  - Pesquisa em largura
  - Pesquisa A\*



# Aplicação: planeamento de sequências de acções

- O problema consiste em determinar uma sequência de acções a desempenhar por um agente por forma a que, partindo de um dado *estado inicial*, se atinja um dado *objectivo*.
- O *conhecimento do domínio* inclui uma descrição das *condições de aplicabilidade* e *efeitos* das acções possíveis.

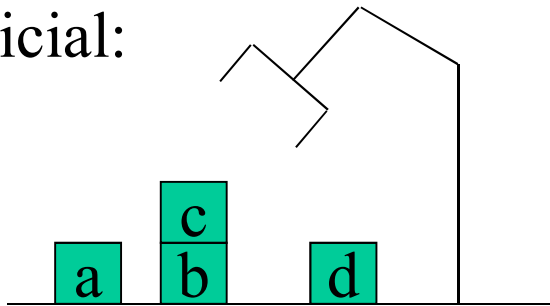


# Representação de acções em problemas de planeamento

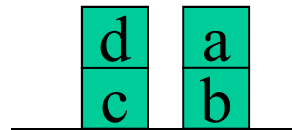
- STRIPS – planeador desenvolvido por volta de 1970, por Fikes, Hart e Nilsson
- A funcionalidade de um dado tipo de operação é definida, no formalismo STRIPS, através de uma estrutura chamada *operador*, que inclui a seguinte informação:
  - *Pré-condições* - um conjunto de fórmulas atómicas que representam as condições de aplicabilidade deste tipo de operação.
  - *Efeitos negativos (delete list)* – um conjunto de fórmulas atómicas que representam propriedades do mundo que deixam de ser verdade ao executar-se a operação.
  - *Efeitos positivos (add list)* – um conjunto de fórmulas atómicas que representam propriedades do mundo que passam a ser verdade ao executar-se a operação.

# Exemplo: planeamento no mundo dos blocos

Estado inicial:



Objectivo:



Plano:

[ desempilhar(c,b),  
poisar(c),  
levantar(d),  
empilhar(d,c),  
levantar(a),  
empilhar(a,b) ]

Especificação de acções. Exemplo: empilhar(X,Y)

- ⊞ Pré-condições: [ no\_robot(X), livre(Y) ]
- ⊞ Efeitos negativos: [ no\_robot(X), livre(Y) ]
- ⊞ Efeitos positivos: [ em\_cima(X,Y), robot\_livre ]

# Pesquisa A\* - heurísticas

- Uma heurística é tanto melhor quanto mais se aproximar do custo real
  - A qualidade de uma heurística pode ser medida através do factor de ramificação efectiva
  - Quanto melhor a heurística, mais baixo será esse factor
- Em alguns domínios, há funções de estimação de custos que naturalmente constituem heurísticas admissíveis
  - Exemplo: Distância em linha recta no domínio dos caminhos entre cidades
- Em muitos outros domínios práticos, não há uma heurística admissível que seja óbvia
  - Exemplo: Planeamento no mundo dos blocos



# Pesquisa A\* - cálculo de heurísticas admissíveis em problemas simplificados

- Um problema simplificado (*relaxed problem*) é um problema com menos restrições do que o problema original
  - É possível gerar automaticamente formulações simplificadas de problemas a partir da formulação original
  - A resolução do problema simplificado será feita usando pouca ou nenhuma pesquisa
  - Pode-se assim “inventar” heurísticas, escolhendo a melhor, ou combinando-as numa nova heurística
- **IMPORTANTE:** O custo de uma solução óptima para um problema simplificado constitui uma heurística admissível para o problema original

# Pesquisa A\* - combinação de heurísticas

- Se tivermos várias heurísticas admissíveis ( $h_1, \dots, h_n$ ), podemos combiná-las numa nova heurística:
  - $H(n) = \max(\{h_1(n), \dots, h_n(n)\})$
- Esta nova heurística tem as seguintes propriedades:
  - Admissível
  - Dado que é uma melhor aproximação ao custo real, vai ser uma heurística melhor do que qualquer das outras

# Pesquisa A\* em aplicações práticas

- Principais vantagens
  - Completa
  - Óptima
- Principais desvantagens
  - Na maior parte das aplicações, o consumo de memória e tempo de computação têm um comportamento exponencial em função do tamanho da solução
  - Em problemas mais complexos, poderá ser preciso utilizar algoritmos mais eficientes, ainda que sacrificando a optimalidade
  - Ou então, usar heurísticas com uma melhor aproximação média ao custo real, ainda que não sendo estritamente admissíveis, e não garantindo portando a optimalidade da pesquisa

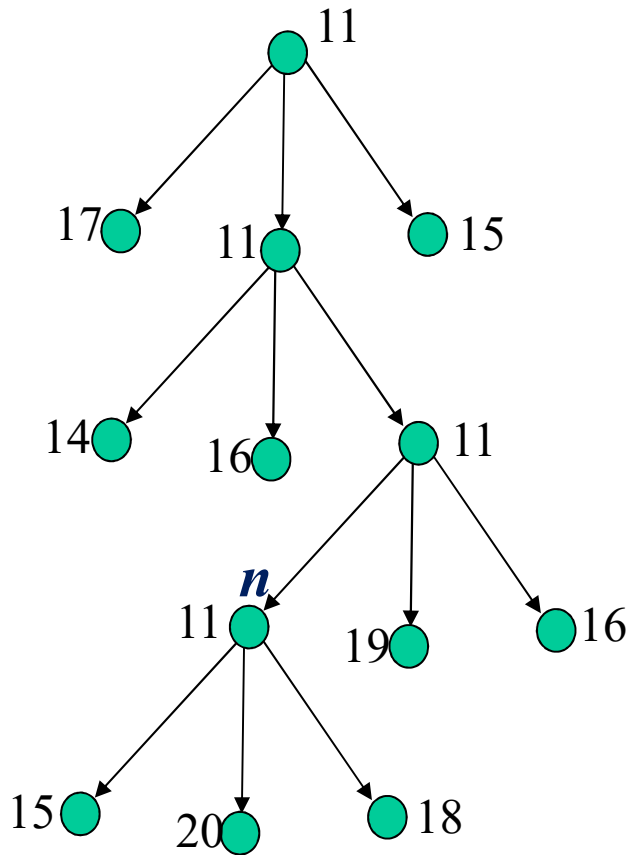
# IDA\*

- Semelhante à pesquisa em profundidade com aprofundamento iterativo
- A limitação à profundidade é estabelecida indirectamente através de um limite na função de avaliação  $f(n) = g(n) + h(n) \leq f_{max}$ 
  - Ou seja: Qualquer nó  $n$  com  $f(n) > f_{max}$  não será expandido
- Passos do algoritmo:
  1.  $f_{max} = f(\text{raiz})$
  2. Executar pesquisa em profundidade com limite  $f_{max}$
  3. Se encontrou solução, retornar solução encontrada
  4.  $f_{max} \leftarrow$  menor  $f(n)$  que tenha sido superior a  $f_{max}$  na última execução do A\*
  5. Voltar ao passo 2

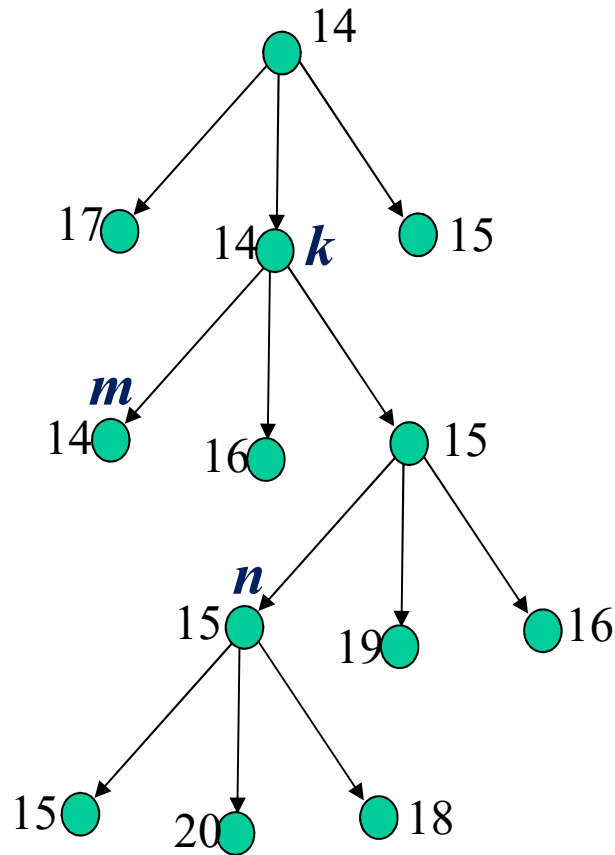
# RBFS

- Pesquisa recursiva melhor-primeiro (*Recursive Best-First Search*)
- Para cada nó  $n$ , o algoritmo não guarda o valor da função de avaliação  $f(n)$ , mas sim o menor valor  $f(x)$ , sendo  $x$  uma folha descendente do nó  $n$ 
  - Sempre que um nó é expandido, os custos armazenados nos ascendentes são actualizados
- Funciona como pesquisa em profundidade com retrocesso
  - Quando a folha  $m$  com menor custo  $f(m)$  não é filha do último nó expandido  $n$ , então o algoritmo retrocede até ao ascendente comum de  $m$  e  $n$

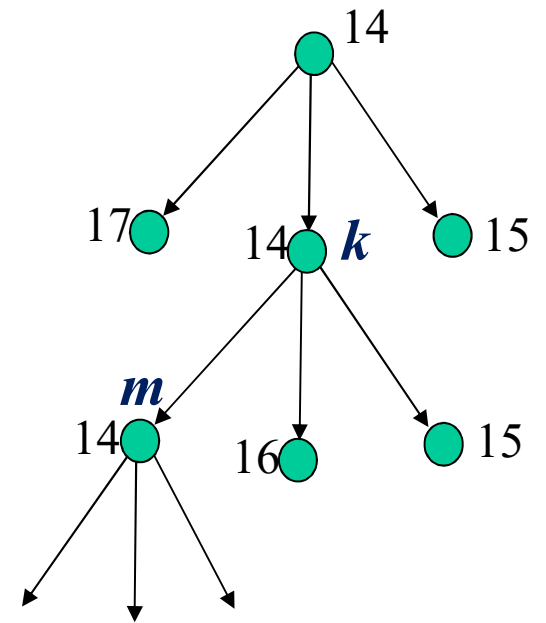
# RBFS – exemplo



Nó *n* acaba de ser expandido



Custos foram actualizados

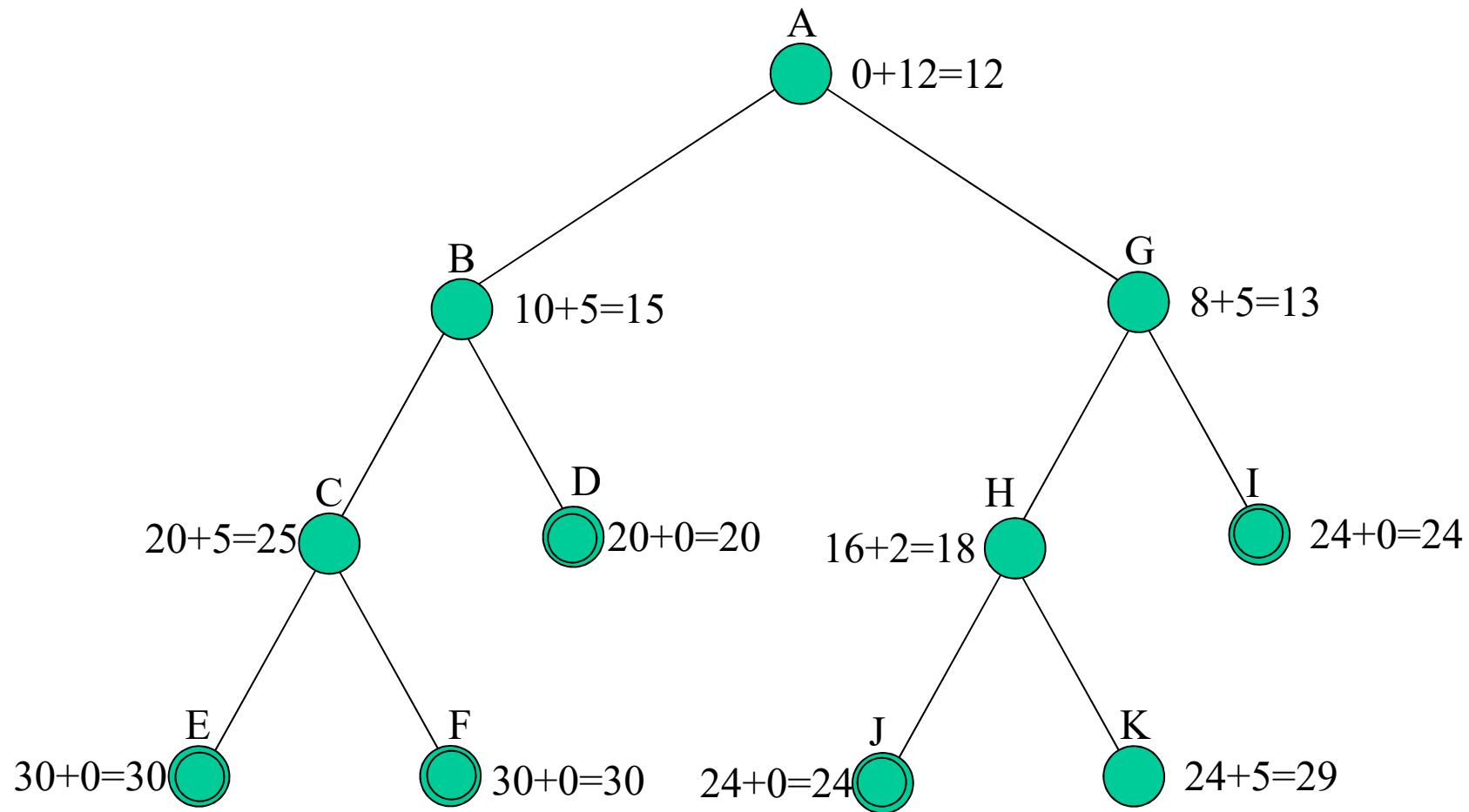


Algoritmo retrocedeu até ao nó *k*;  
Expansão segue pelo nó *m*

# SMA\*

- A\* com memória limitada simplificado (*simplified memory-bounded A\**)
- Usa a memória disponível
  - Contraste com IDA\* e RBFS: estes foram desenhados para poupar memória, independentemente de ela existir de sobra ou não
- Quando a memória chega ao limite, esquece (remove) o nó  $n$  com maior custo  $f(n)=g(n)+h(n)$ , actualizando em cada um dos nós ascendentes o “custo do melhor nó esquecido”
- Só volta a gerar o nó  $n$  quando o custo do melhor nó esquecido registado no antecessor de  $n$  for inferior aos custos dos restantes nós
- Em cada iteração, é gerado apenas um nó sucessor
  - Existindo já um ou mais filhos de um nó, apenas se gera ainda outro se o custo do nó pai for menor do que qualquer dos custos dos filhos
  - Quando se gerou todos os filhos de um nó, o custo do nó pai é actualizado como no RBFS

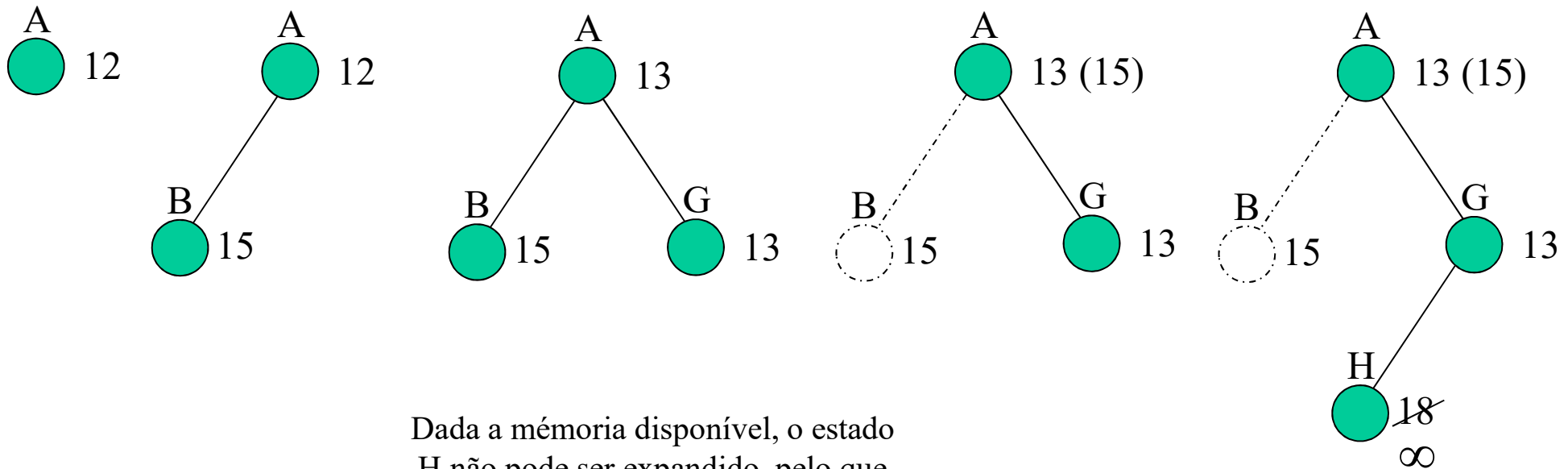
# SMA\* - exemplo – espaço de estados





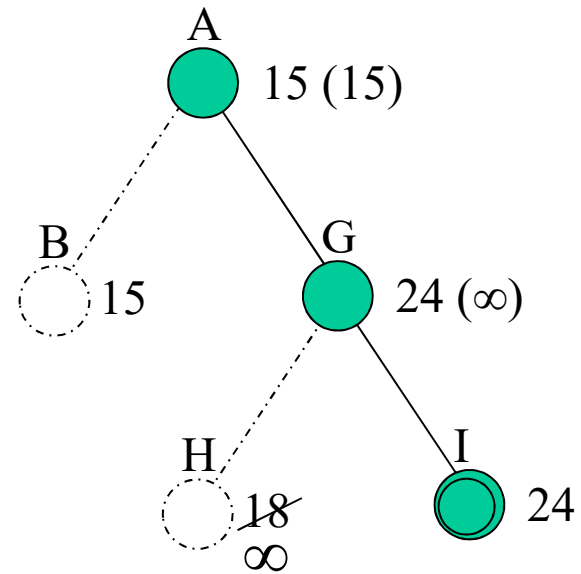
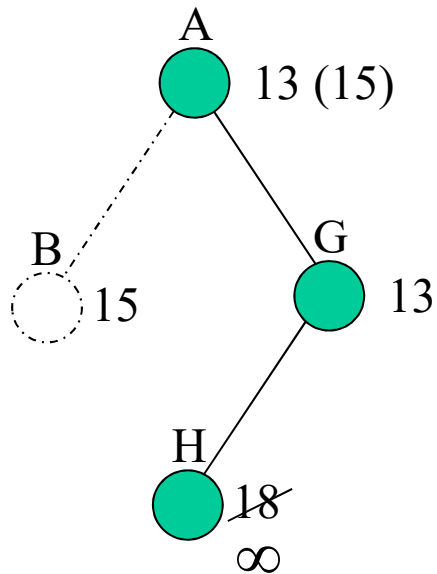
# SMA\* - exemplo

- Neste exemplo: memória = 3 nós
  - Melhores custos de nós esquecidos anotados entre parêntesis



Dada a memória disponível, o estado  
H não pode ser expandido, pelo que  
o seu custo é infinito

# SMA\* - exemplo (cont.)



- Chegámos a uma solução (estado I)
- Se quisermos continuar: Das restantes folhas já exploradas, a que tinha o estado B era a melhor, por isso a pesquisa retrocede e continua expandindo esse folha

# Estratégias de pesquisa

- Pesquisa em árvore
- Pesquisa com propagação de restrições
- Pesquisa por melhorias sucessivas
- Planeamento

# Pesquisa para problemas de atribuição

- Nos problemas de atribuição pretende-se atribuir valores a um conjunto de variáveis, respeitando um conjunto de restrições.
- Exemplos:
  - Problema das 8 rainhas - distribuir 8 rainhas num tabuleiro de xadrez de forma a que haja uma e uma só rainha em cada linha e em cada coluna e não haja mais do que uma rainha em cada diagonal.
  - Invenção de palavras cruzadas – dada uma matriz de palavras cruzadas vazia, preencher os espaços brancos com letras, de forma a que a matriz possa ser usada como passatempo de palavras cruzadas.
- Técnicas de resolução de problemas de atribuição:
  - Método construtivo – usando técnicas de pesquisa em árvore
    - Em cada passo da pesquisa atribui-se um valor a uma variável
  - Método construtivo combinado com propagação de restrições
  - Resolução por melhorias sucessivas

# Pesquisa com propagação de restrições em problemas de atribuição

- Construir um grafo de restrições:
  - Em cada nó do grafo está uma variável
  - Um arco dirigido liga um nó  $i$  a um nó  $j$  se o valor da variável de  $j$  impõe restrições ao valor da variável de  $i$ .
  - Um arco  $(i,j)$  é *consistente* se, para cada valor da variável  $i$ , existe um valor da variável  $j$  que não viola as restrições.
- Tipicamente, usa-se uma estratégia de pesquisa em profundidade; em cada iteração da pesquisa, faz-se o seguinte:
  - 1) Seleciona-se arbitrariamente um dos valores possíveis para uma das variáveis (descartam-se os restantes)
  - 2) Restringem-se os conjuntos de valores possíveis das restantes variáveis por forma a que os arcos do grafo de restrições continuem consistentes.
- Nota: Neste caso, cada estado da pesquisa não representa uma situação ou configuração possível do mundo, como acontece no problema dos blocos; o estado é constituído pelos conjuntos de valores possíveis para as variáveis.

# Pesquisa com propagação de restrições em problemas de atribuição - algoritmo

1. Inicialização: o nó inicial da árvore de pesquisa é composto por todas as variáveis e todos os valores possíveis para cada uma delas
2. Se pelo menos uma variável tem um conjunto de valores vazio, falha e retrocede; se não puder retroceder, a pesquisa falha
3. Se todas as variáveis têm exactamente um valor possível, tem-se uma solução; retornar com sucesso
4. Expansão: Escolher arbitrariamente uma variável  $V_k$  e, de entre os valores possíveis, um dado valor  $X_{kl}$  – descartar os restantes valores possíveis dessa variável
5. Propagação de restrições: para cada arco  $(i,j)$  no grafo de restrições, remover os valores na variável  $V_i$  por forma a que o arco fique consistente
6. Caso tenha sido preciso remover valores na origem de algum arco, voltar a repetir o passo 5.
7. Voltar ao passo 2.

# Propagação de restrições - algoritmo

- Os passos 5 e 6 do algoritmo anterior executam a propagação de restrições
- Esta parte do processo é suportada por uma fila de arestas do grafo de restrições
  - Inicialmente, a fila contém as arestas que apontam para a variável cujo valor foi fixado

```
propagarRestricoes(grafoRestricoes, FilaArestas) retorna o grafo de restrições com  
domínios possivelmente mais limitados  
enquanto FilaArestas não vazia fazer {  
     $(X_j, X_i) \leftarrow$  remover cabeça de FilaArestas  
    remover valores inconsistentes em  $X_j$   
    se removeu valores, então  
        para cada vizinho  $X_k$ , acrescentar  $(X_k, X_j)$  a FilaArestas  
}
```

# Tipos de restrições

- Restrições unárias – envolvem apenas uma variável
  - Podem ser satisfeitas através de pré-processamento do domínio de valores da variável – aproveitam-se apenas os valores que satisfazem a restrição
- Restrições binárias – envolvem duas variáveis
  - Uma restrição binária é directamente representada por uma aresta no grafo de restrições
- Restrições de ordem superior – envolvem três ou mais variáveis
  - Através da introdução de variáveis auxiliares, uma restrição de ordem superior pode ser transformada num conjunto de restrições binárias e/ou unárias



# Exemplo: quebra-cabeças critpoaritmético

$$\begin{array}{r} \text{T W O} \\ + \text{T W O} \\ \hline \text{F O U R} \end{array}$$

- Variáveis principais: F, O, R, T, U, W ( $\in \{ 0 .. 9 \}$ )
- Variáveis internas:  $X_1$  (transporte das unidades para as dezenas) e  $X_2$  (transporte das dezenas para as centenas) ( $\in \{ 0, 1 \}$ )
- Restrições:
  - Todas as variáveis são diferentes [ restrição sobre 6 variáveis ]
  - $2 \cdot O = R + 10 \cdot X_1$  [ restrição sobre 3 variáveis ]
  - $2 \cdot W + X_1 = U + 10 \cdot X_2$  [ restrição sobre 4 variáveis ]
  - $2 \cdot T + X_2 = O + 10 \cdot F$  [ restrição sobre 4 variáveis ]

# Restrições de ordem superior – conversão para restrições binárias

- No exemplo anterior, a restrição ternária  $2 \cdot O = R + 10 \cdot X_I$  pode ser transformada no seguinte conjunto de restrições:
  - Restrições binárias:
    - $O = \text{primeiro}(Aux)$
    - $R = \text{segundo}(Aux)$
    - $X_I = \text{terceiro}(Aux)$
  - Restrição unária:
    - $2 \cdot \text{primeiro}(Aux) = \text{segundo}(Aux) + 10 \cdot \text{terceiro}(Aux)$
- $Aux$  é uma variável auxiliar cujo domínio é o produto cartesiano dos domínios de  $O$ ,  $R$  e  $X_I$ .
  - Ou seja:  $Aux \in \{0 \dots 9\} \times \{0 \dots 9\} \times \{0, 1\}$
  - A restrição unária sobre  $Aux$  pode ser satisfeita através de pré-processamento

# Estratégias de pesquisa

- Pesquisa em árvore
- Pesquisa com propagação de restrições
- Pesquisa por melhorias sucessivas
  - Montanhismo (*hill-climbing*)
  - Recozimento simulado (*Simulated annealing*)
  - Algoritmos genéticos
- Planejamento

# Pesquisa por melhorias sucessivas

- Também conhecida como **pesquisa local**
  - A partir de uma dada configuração inicial, fazem-se refinamentos sucessivos até obter uma configuração satisfatória
  - A solução inicial pode ser aleatória
- Técnicas mais comuns:
  - Reparação heurística
    - É a versão mais básica deste tipo de pesquisa: reparações à solução inicial vão sendo aplicadas de acordo com uma heurística local.
    - No caso de problemas de satisfação de restrições, a heurística pode ser:
      - Fazer a reparação que, naquele momento, mais contribui para reduzir os conflitos entre variáveis, dadas as restrições.
  - Montanhismo
  - Recozimento simulado
  - Algoritmos genéticos

# Pesquisa por melhorias sucessivas:

## montanhismo

- A pesquisa é vista como um problema de otimizar uma função
- O espaço de soluções é visto como uma paisagem de vales (zonas de soluções menos satisfatórias) e colinas (zonas de soluções melhores).
- Tem semelhanças com a pesquisa em profundidade e com a pesquisa gulosa, diferenciando-se pelo seguinte:
  - Escolhe-se sempre o sucessor com melhor valor da função de avaliação
  - Não há retrocesso (*backtracking*)
  - Quando o valor da função no nó actual é superior ao valor da função em qualquer dos seus sucessores, a pesquisa pára. ( atingiu-se um máximo local )
- Problemas:
  - Máximos locais, planaltos, ravinas

# Montanhismo: variantes

- **Montanhismo estocástico** – escolhe aleatoriamente entre os sucessores que melhoram a função de avaliação
- **Montanhismo de primeira escolha** – escolhe sucessores aleatoriamente até encontrar um com melhor função de avaliação que o estado actual
- **Montanhismo com reinício aleatório** – executar o montanhismo várias vezes, partindo de estados iniciais aleatórios, e escolhe a melhor solução
- **Recozimento simulado** (página seguinte)

# Pesquisa por melhorias sucessivas:

## recozimento simulado

- *Recozimento simulado (Simulated Annealing)* é uma variante da pesquisa por montanhismo na qual podem ser aceites refinamentos que, localmente, piorem a solução.
  - O nome inspira-se no processo industrial chamado *recozimento*.
    - Recozer = “deixar esfriar lentamente (um produto de cerâmica ou de vidro) num forno especial, logo após o seu fabrico”.
- Começou a ser usado circa 1980 para resolver problemas de configuração de circuitos VLSI
- Particularidades:
  - O sucessor é seleccionado aleatoriamente
  - Quando o valor da função no nó actual é superior ao valor da função no sucessor, o sucessor é aceite com uma probabilidade que diminui exponencialmente em função da perda na função de avaliação.
  - Pesquisa termina quando um indicador designado “temperatura” chega a zero.

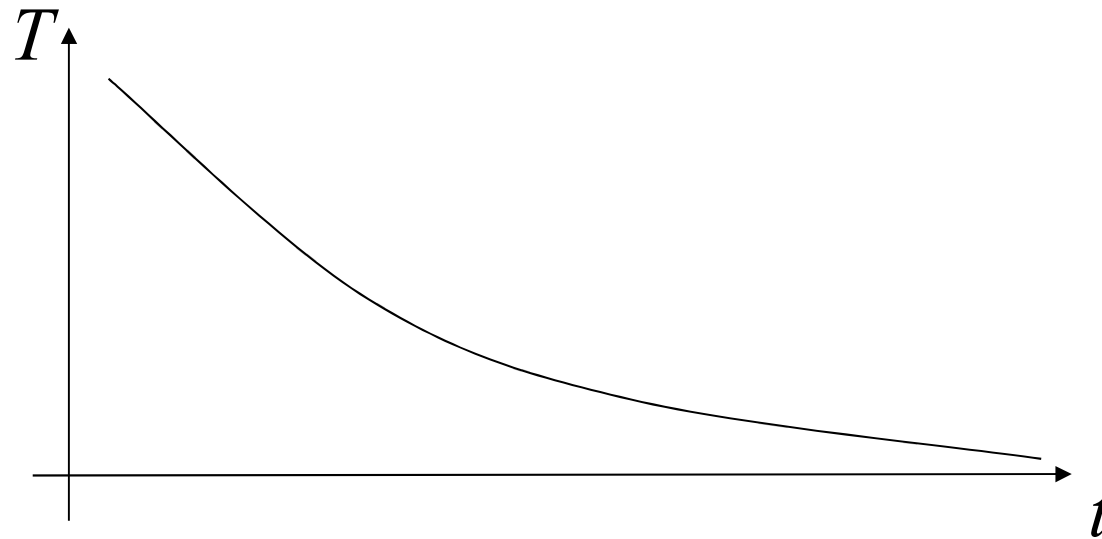
# Recozimento simulado: algoritmo

```
recozimento_simulado(Problema, Regime_termico, Aval)  
(* A função Regime_termico dá a temperatura em função do tempo. *)  
Nó ← fazer_nó(estado inicial do Problema)  
repetir para  $t=0 \dots \infty$ : {  
     $T \leftarrow \text{Regime\_termico}(t)$   
    se  $T=0$ , retornar a solução de Nó  
    Prox ← um sucessor de Nó gerado aleatoriamente  
     $\text{Ganho} \leftarrow \text{Aval}(\text{Prox}) - \text{Aval}(\text{Nó})$   
    se  $\text{Ganho} > 0$ , Nó ← Prox  
    senão, com probabilidade  $\exp(\text{Ganho}/T)$ , fazer: Nó ← Prox  
}
```

- ⌘ Nota: Se a temperatura  $T$  diminuir de forma suficientemente lenta, o recozimento simulado encontra um máximo global. (solução ótima)



# Recozimento simulado: regime térmico



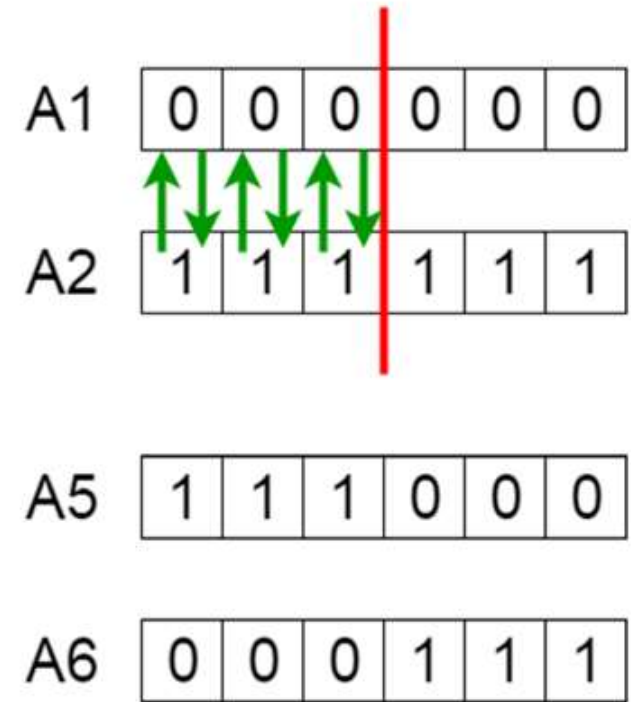
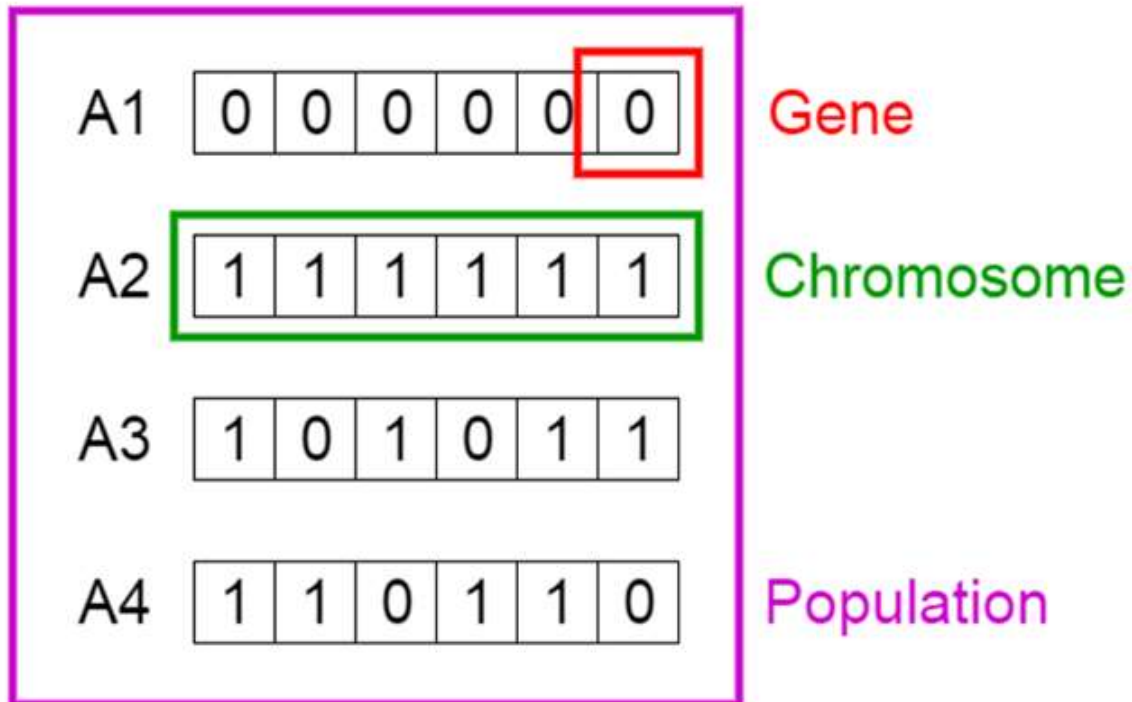
- $t \rightarrow \infty$
- $T \rightarrow 0$
- $Ganho/T \rightarrow -\infty$  (dado que o Ganho é negativo)
- Probabilidade:  $\exp(Ganho/T) \rightarrow 0$
- Ou seja: À medida que o tempo passa, a pesquisa arrisca cada vez menos quanto a aceitar alterações com ganho negativo

# Pesquisa local alargada

## *(local beam search)*

- **Pesquisa local alargada** – semelhante ao montanhismo mas, em cada iteração, são mantidos  $k$  estados, e os melhores  $k$  sucessores são passados para a iteração seguinte
- **Pesquisa alargada estocástica** – semelhante à pesquisa local alargada, mas os  $k$  sucessores são seleccionados aleatoriamente
- **Algoritmos genéticos** – variante da pesquisa alargada estocástica em que os sucessores são gerados por combinação de dois estados, em vez de modificar um único estado

# Algoritmos Genéticos



# Estratégias de pesquisa

- Pesquisa em árvore
- Pesquisa com propagação de restrições
- Pesquisa por melhorias sucessivas
- Planeamento

# Planeamento: STRIPS, o primeiro planeador

STRIPS(*EI*,*Objectivos*)    % *EI* é argumento de entrada/saída

*Plano*  $\leftarrow$  [ ]

**repetir** {

*C*  $\leftarrow$  uma condição em *Objectivos* não satisfeita em *EI*

*OP*  $\leftarrow$  um operador que pode ter *C* como efeito positivo

*A*  $\leftarrow$  acção, dada por uma completa instanciação de *OP*

*PC*  $\leftarrow$  pré-condições de *A*

*SubPlano*  $\leftarrow$  STRIPS(*EI*,*PC*)

*Plano*  $\leftarrow$  concatenar(*Plano*,concatenar(*SubPlano*,[*A*]))

*EI*  $\leftarrow$  novo estado, resultante da aplicação de *A* em *EI*

**se** *Objectivos* satisfeitos em *EI*,

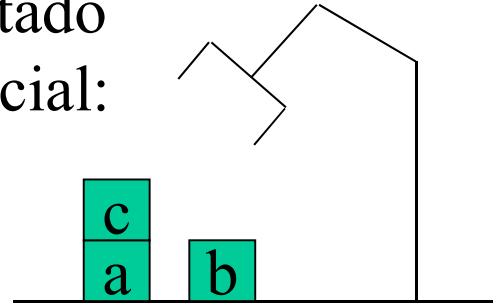
**retornar** *Plano*

}

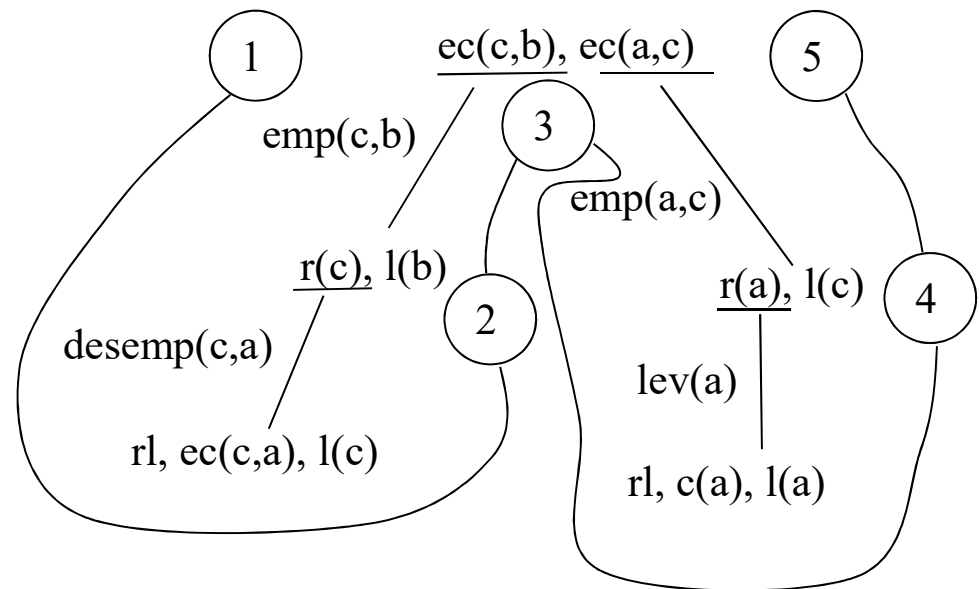
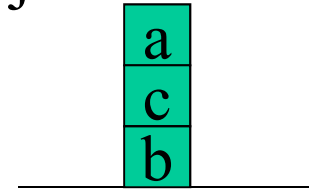
# STRIPS: exemplo

- Abreviaturas de condições:
  - Bloco em cima de bloco:  $ec(A,B)$
  - Bloco no chão:  $c(B)$
  - Bloco no robô:  $r(X)$
  - Robô livre:  $rl$
  - Bloco livre:  $l(X)$
- Abreviaturas de operadores:
  - Empilhar:  $emp(A,B)$
  - Desempilhar:  $desemp(A,B)$
  - Levantar:  $lev(X)$
  - Poisar:  $p(X)$
- Plano:
  - $desemp(c,a), emp(c,b), lev(a), emp(a,c)$
- Sucessão de estados:
  - 1:  $ec(c,a), c(a), l(c), c(b), l(b), rl$
  - 2:  $r(c), l(a), c(a), c(b), l(b)$
  - 3:  $ec(c,b), l(c), l(a), c(b), c(a), rl$
  - 4:  $r(a), ec(c,b), c(b), l(c)$
  - 5:  $ec(a,c), ec(c,b), c(b), rl$

Estado inicial:

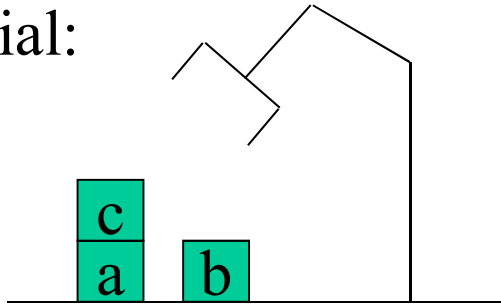


Objectivo:

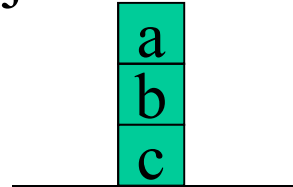


# A Anomalia de Sussman

Estado  
inicial:



Objectivo:



- Dependendo da ordem pela qual o STRIPS trata os objectivos, os seguintes planos poderão ser gerados:
  - [ desempilhar(c,a), poisar(c), levantar(a), empilhar(a,b), desempilhar(a,b), poisar(a), levantar(b), empilhar(b,c), levantar(a), empilhar(a,b) ]
  - [ levantar(b), empilhar(b,c), desempilhar(b,c), poisar(b), desempilhar(c,a), poisar(c), levantar(a), empilhar(a,b), desempilhar(a,b), poisar(a), levantar(b), empilhar(b,c), levantar(a), empilhar(a,b) ]
- Nenhum deles é óptimo
  - Na verdade, o algoritmo STRIPS não consegue gerar um plano óptimo para este problema

# Planeamento no espaço de soluções

- Em todas as aproximações ao planeamento anteriormente apresentadas, cada nó da pesquisa corresponde a um estado do mundo → *planeamento no espaço de estados*.
- Uma técnica alternativa consiste em partir de um plano vazio e adicionar sucessivamente operações e restrições de sequenciamento → *planeamento no espaço de soluções*.
- Neste caso, cada nó da pesquisa corresponde a uma solução parcial para o problema.
- Operações de transformação da solução:
  - Adicionar um operador
  - Re-ordenar operadores
  - Instanciar um operador



# Planeamento Hierárquico

## – a técnica ABSTRIPS

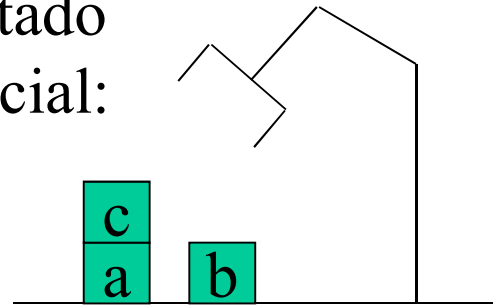
- O planeamento é realizado numa hierarquia de níveis de abstração.
- Um valor de “criticalidade” é atribuído a cada uma das condições que podem aparecer na descrição do estado do mundo.
- Algoritmo:
  1.  $CM \leftarrow$  valor inicial para o nível de criticalidade mínima.
  2. Gerar um plano que satisfaça todas as condições com nível de criticalidade  $\geq CM$ .
  3.  $CM \leftarrow CM-1$
  4. Usando o plano anterior como guia, gerar um plano que satisfaça todas as condições com criticalidade  $\geq CM$ .
  5. **se** todas as condições estão satisfeitas, **retornar** a solução.
  6. **voltar** ao passo 3.

# ABSTRIPS: exemplo

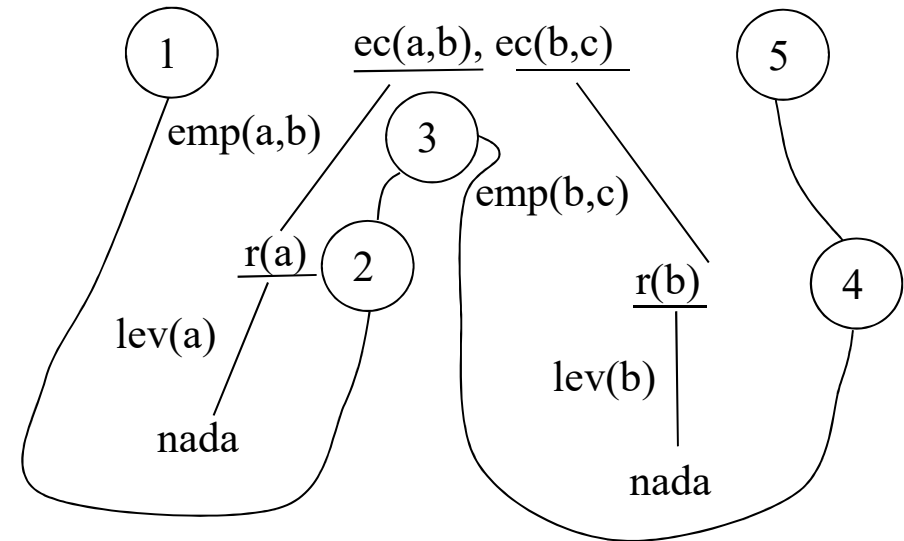
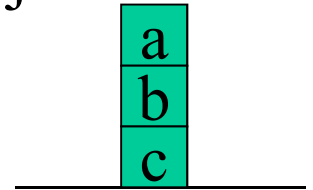
## – planeamento inicial para CM=2

- Dois níveis de criticalidade:
  - $ec(A,B) - 2$
  - $c(B) - 1$
  - $r(X) - 2$
  - $rl - 1$
  - $l(X) - 1$
- Plano inicial:
  - $lev(a), emp(a,b), lev(b), emp(b,c)$
- Sucessão de estados:
  - 1:  $ec(c,a)$
  - 2:  $ec(c,a), r(a)$
  - 3:  $ec(c,a), ec(a,b)$
  - 4:  $ec(c,a), ec(a,b), r(b)$
  - 5:  $ec(c,a), ec(a,b), ec(b,c)$
- Os estados não são consistentes!

Estado inicial:



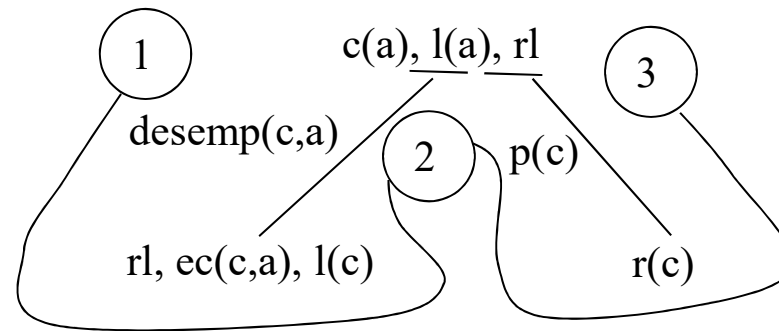
Objectivo:



# ABSTRIPS: exemplo

## – Planeamento para CM=1

- As precondições de criticalidade 1 da primeira acção,  $lev(a)$ , não estão reunidas, pelo que é preciso determinar um plano para as atingir



- Plano:
  - $desemp(c,a), p(c)$
- Sucessão de estados:
  - 1:  $ec(c,a), c(a), l(c), c(b), l(b), rl$
  - 2:  $r(c), l(a), c(a), c(b), l(b)$
  - 3:  $rl, c(c), l(c), l(a), c(a), c(b), l(b)$

# Operadores com fórmulas não atômicas e condicionais

- Literal – uma formula atômica (literal positivo) ou negação de uma fórmula atômica (literal negativo)
- Fórmula de aplicabilidade do operador pode ser:
  - Fórmula atômica
  - Negação de uma fórmula
  - Conjunção de fórmulas
  - Disjunção de fórmulas
  - Fórmula quantificada existencialmente
  - Fórmula quantificada universalmente
- Fórmula de efeitos do operador pode ser:
  - Literal
  - Conjunção de literais
  - Efeitos condicionais: when <fórmula de aplicabilidade> <fórmula de efeitos>
  - Fórmula de efeitos quantificada universalmente
  - Conjunção de fórmulas de efeitos
- Ver “PDDL - Planning Domain Definition Language”.

# PDDL - exemplo

(:action stop

:parameters (?f - floor)

:precondition (lift-at ?f)

:effect (and

(forall (?p - passenger)

(when (and (boarded ?p) (destin ?p ?f) )

(and (not (boarded ?p)) (served ?p))))

(forall (?p - passenger)

(when (and (origin ?p ?f) (not (served ?p)))

(boarded ?p))))

# PDDL - exemplo

```
(:action drive-truck
:parameters (?truck – truck
              ?loc-from ?loc-to - location
              ?city - city)
:precondition (and (at ?truck ?loc-from) (in-city ?loc-from ?city)
                  (in-city ?loc-to ?city))
:effect (and (at ?truck ?loc-to)
             (not (at ?truck ?loc-from))
             (forall (?x - obj)
              (when (and (in ?x ?truck)
                        (and (not (at ?x ?loc-from))
                           (at ?x ?loc-to)))))))
```