Introdução à Gestão de Conhecimento

- Bayesian Networks

Objectives

Review on Probability Theory
Bayesian Networks

Ways to deal with Uncertainty

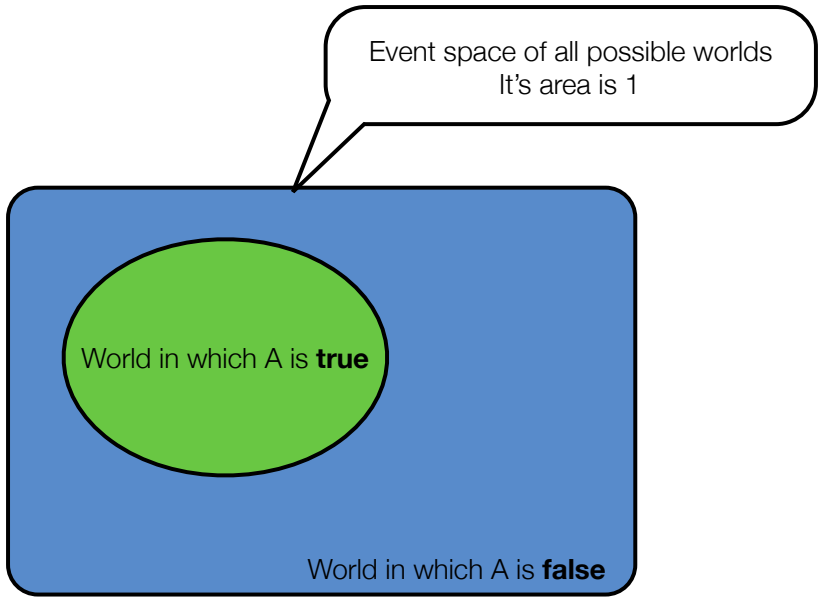
- Three-valued logic: True / False / Maybe
- Fuzzy logic (truth values between 0 and 1)
- Non-monotonic reasoning
- Dempster-Shafer theory (and an extension known as quasi-Bayesian theory)
- Possibilistic Logic
- Probability

Discrete Random Variables

- A is a Boolean-valued random variable if A denotes an event, and there is some degree of uncertainty as to whether A occurs.
- Examples
  - A = The US president in 2023 will be male
  - A = You wake up tomorrow with a headache
  - A = You have Ebola

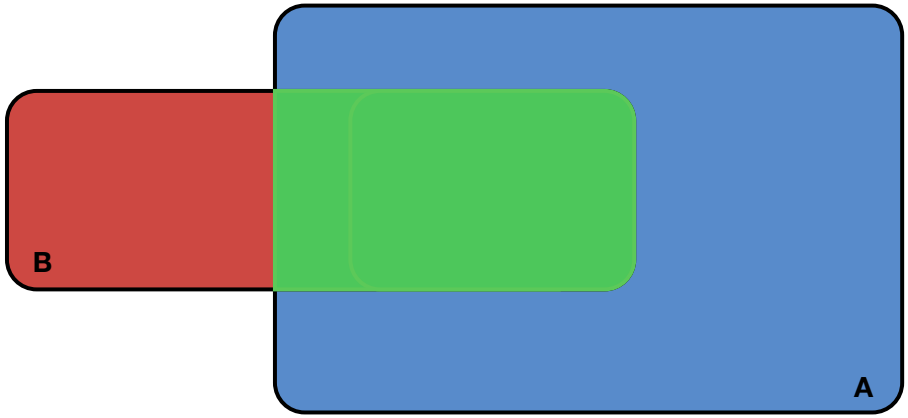
Probabilities

- We write P(A) as “the fraction of possible worlds in which A is true”
- P(A) = Area of the green oval



Interpreting the axioms

- Axioms:
  - 0<=P(A)<=1
  - P(True) = 1
  - P(False) = 0
  - P(A or B)=P(A) + P(B) - P(A and B)



Theorems from the Axioms

- 0<=P(A)<=1, P(True)=1, P(False)=0
- P(A or B) = P(A) + P(B) - P(A and B)
- From these we can prove:
  - **P(not A) = P(~A) = 1-P(A)**

- 0<= P(A) <=1, P(True)=1, P(False)=0
- P(A or B) =P(A) + P(B) - P(A and B)
- From these we can prove:
  - **P(A) = P(A ^ B) + P(A ^ ~B)**

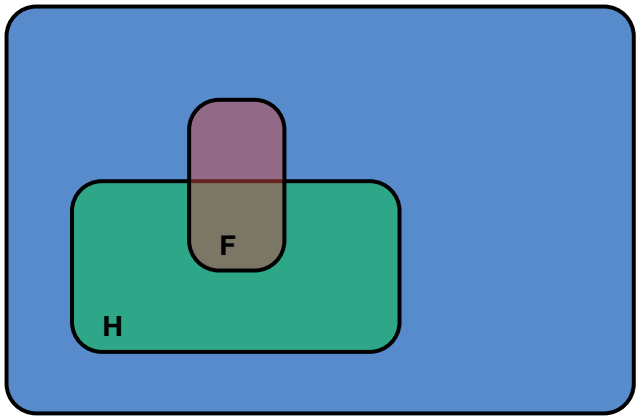
Conditional Probability

- P(A|B) = Fraction of worlds in which B is true that also have A true

  H = “Have a headache”
  F = “Coming down with Flu”

  P(H) = 1/10
  P(F) = 1/40
  P(H|F) = 1/2

- “Headaches are rare and flu is rarer, but if you’re coming down with ‘flu there’s a 50-50 chance you’ll have a headache.”



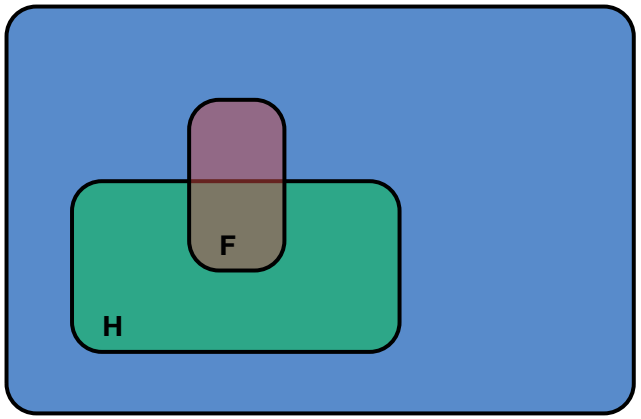
Conditional Probability

P(H|F) = Fraction of flu-inflicted worlds in which you have a headache

$$= \frac{\#\text{worlds with flu and headache}}{\#\text{worlds with flu}}$$

$$= \frac{\text{Area of “H and F” region}}{\text{Area of “F” region}}$$

$$= \frac{P(H \wedge F)}{P(F)}$$



Conditional Probability

- Definition of Conditional Probability

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

- Corollary: The Chain Rule

$$P(A \wedge B) = P(A|B)\,P(B)$$

- **The Bayes Rule**:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)} = \frac{P(A|B)\,P(B)}{P(A)}$$

Conditional Probability

- Suppose A can take on more than 2 values
- A is a random variable with arity k if it can take on exactly one value out of $\{v_1, v_2, .. v_k\}$
- Thus...

  $P(A=v_i \wedge A=v_j)=0$ if $i \neq j$
  $P(A=v_1 \vee A=v_2 \vee A=v_k) =1$

- From the previous axioms:

  $$P(B \wedge [A=v_1 \vee A=v_2 \vee A=v_i]) = \sum P(B \wedge A=v_j)$$

- And thus we can prove

  $$P(B) = \sum P(B \wedge A = v_j)$$

More General Forms of Bayes Rule

- $$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A)+P(B|\sim A)P(\sim A)}$$

- $$P(A|B \wedge X) = \frac{P(B|A \wedge X)P(A \wedge X)}{P(B \wedge X)}$$

- $$P(A=v_i|B) = \frac{P(B|A=v_i)P(A=v_i)}{\sum P(B|A=v_k)P(A=v_k)}$$

Bayes Rule - Example

- Marie is getting married tomorrow, at an outdoor ceremony in the desert. In recent years, it has rained only 5 days each year. Unfortunately, the weatherman has predicted rain for tomorrow. When it actually rains, the weatherman correctly forecasts rain 90% of the time. When it doesn't rain, he incorrectly forecasts rain 10% of the time. What is the probability that it will rain on the day of Marie's wedding?
- **Solution**: The sample space is defined by two mutually-exclusive events - it rains or it does not rain. Additionally, a third event occurs when the weatherman predicts rain. Notation for these events appears below.
- Event **A1**. It rains on Marie's wedding.
- Event **A2**. It does not rain on Marie's wedding
- Event **B**. The weatherman predicts rain.
- In terms of probabilities, we know the following:

  P(A1) = 5/365 =0.0136985 [It rains 5 days out of the year.]

  P(A2) = 360/365 = 0.9863014 [It does not rain 360 days out of the year.]

  P(B | A1) = 0.9 [When it rains, the weatherman predicts rain 90% of the time.]

  P(B | A2) = 0.1 [When it does not rain, the weatherman predicts rain 10% of the time.]

- We want to know P(A1 | B), the probability it will rain on the day of Marie's wedding, given a forecast for rain by the weatherman. The answer can be determined from Bayes' theorem, as shown below.

  $$P(A1 \mid B) = \frac{P(B \mid A1) * P(A1)}{P(A1)\,P(B \mid A1) + P(A2)\,P(B \mid A2)}.$$

  P(A1 | B) = (0.014)(0.9) / [(0.014)(0.9) + (0.986)(0.1)]

  P(A1 | B) = 0.111

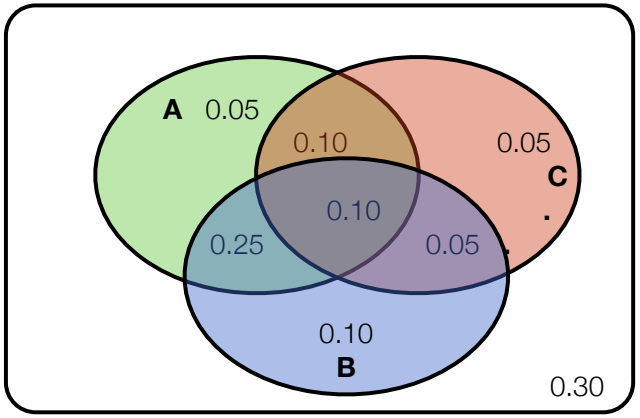
- Note the somewhat unintuitive result. Even when the weatherman predicts rain, it only rains only about 11% of the time. Despite the weatherman's gloomy prediction, there is a good chance that Marie will not get rained on at her wedding.

The Joint Distribution

- Recipe for making a joint distribution of M variables:

1. Make a truth table listing all combinations of values of your variables (if there are M Boolean variables then the table will have 2^M rows).
2. For each combination of values, say how probable it is.
3. If you subscribe to the axioms of probability, those numbers must sum to 1.

A	B	C	Prob
0	0	0	0.30
0	0	1	0.05
0	1	0	0.10
0	1	1	0.05
1	0	0	0.05
1	0	1	0.10
1	1	0	0.25
1	1	1	0.10



Using the Joint Distribution

- Once you have the JD you can ask for the probability of any logical expression involving your attribute

- $P(E) = \sum_{\text{rows matching E}} P(\text{row})$

- P(poor male) = 0.46

- P(poor) = 0.75

gender	hours/ worked	wealth	Prob
Female	<40	poor	0.25
		rich	0.03
	>40	poor	0.04
		rich	0.01
Male	<40	poor	0.33
		rich	0.10
	>40	poor	0.13
		rich	0.11

Inference with the Joint Distribution

- $P(E_1|E_2) = \frac{P(E_1 \wedge E_2)}{P(E_2)}$

$$= \frac{\sum_{\text{rows matching } E_1 \text{ and } E_2} P(\text{row})}{\sum_{\text{rows matching } E_2} P(\text{row})}$$

- P(Male | Poor) = 0.46 / 0.75 = 0.61

gender	hours/ worked	wealth	Prob
Female	<40	poor	0.25
		rich	0.03
	>40	poor	0.04
		rich	0.01
Male	<40	poor	0.33
		rich	0.10
	>40	poor	0.13
		rich	0.11

Joint distributions

- Good news

  Once you have a joint distribution, you can ask important questions about stuff that involves a lot of uncertainty

- Bad news

  Impossible to create for more than about ten attributes because there are so many numbers needed when you build it.

Independence

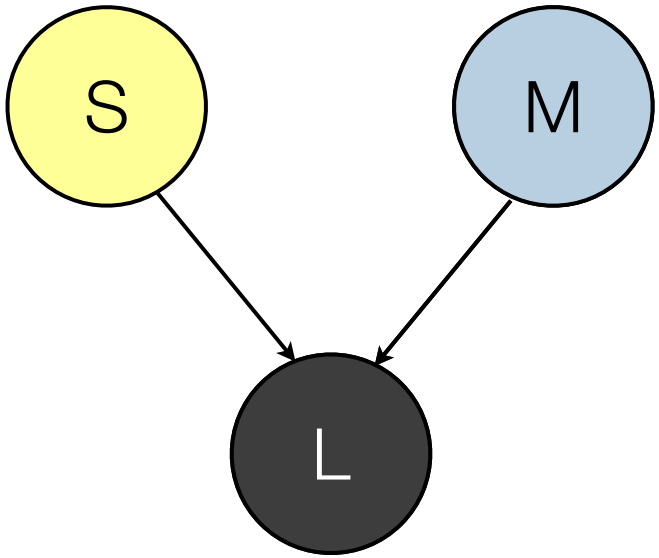
- Suppose there are two events:
  - M: class subject is Maths
  - S: It is Sunny
- The joint probability distribution function (p.d.f.) for these events contain 4 entries
- If we want to build the joint p.d.f. we'll have to invent those four numbers. Do we?
  - We don't have to specify with bottom level conjunctive events such as P(~M^S) IF...
  - ...instead it may sometimes be more convenient for us to specify things like: P(M), P(S).
- But just P(M) and P(S) don't derive the joint distribution. So you can't answer all questions.

Independence

- “The sunshine levels do not depend on and do not influence the subject”
- This can be specified very simply:
  - P(S | M) = P(S) **This is a powerful statement!**
- It required extra domain knowledge. A different kind of knowledge than numerical probabilities. It needed an understanding of causation.
- From P(S | M) = P(S), the rules of probability imply:
- P(~S | M) = P(~S)
- P(M | S)=P(M)
- P(M^S)=P(M).P(S)
- P(~M ^ S) = P(~M) P(S), P(M^~S) = P(M)P(~S), P(~M^~S) = P(~M)P(~S)

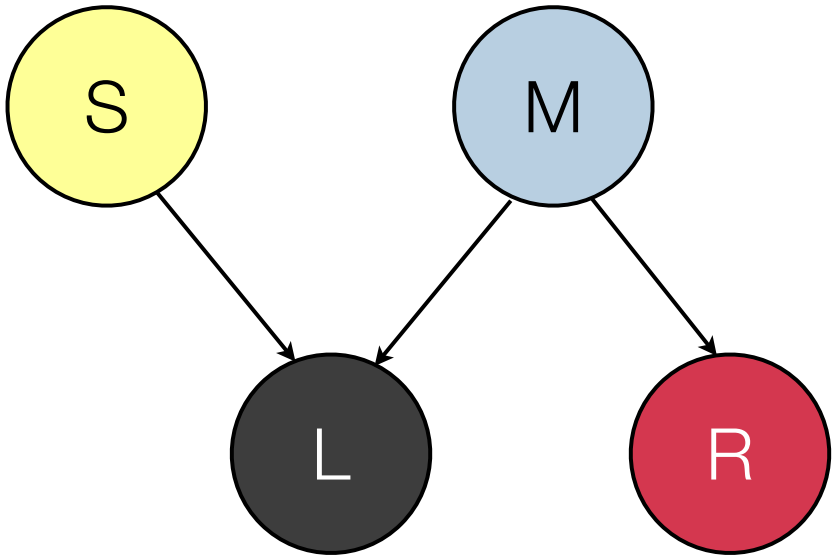
A bit of notation

- Assume a new event
  - L : The lecturer arrives slightly late.
- P(S | M) = P(S)
  P(S) = 0.3
  P(M) = 0.6
- P(L | M ^ S) = 0.05
  P(L | M ^ ~S) = 0.1
  P(L | ~M ^ S) = 0.1
  P(L | ~M ^ ~S) = 0.2



Conditional Independence

- Assume a new event
  - R: we will do Revisions in class
- P(R | M,L) = P(R | M)
  P(R | ~M,L) = P(R | ~M)
- “R and L are conditionally independent given M”
- Given knowledge of M and S, knowing anything else in the diagram won’t help us with L, etc.



Conditional Independence formalized

- R and L are conditionally independent given M if for all x,y,z in {True,False}:
  - P(R=x | M=y ^ L=z) = P(R=x | M=y)
- More generally:
  - Let S1 and S2 and S3 be sets of variables.
  - Set-of-variables S1 and set-of-variables S2 are conditionally independent given S3 if for all assignments of values to the variables in the sets,

    P(S1's assignments | S2's assignments & S3's assignments)
    = P(S1's assignments | S3's assignments)
  - **BUT**

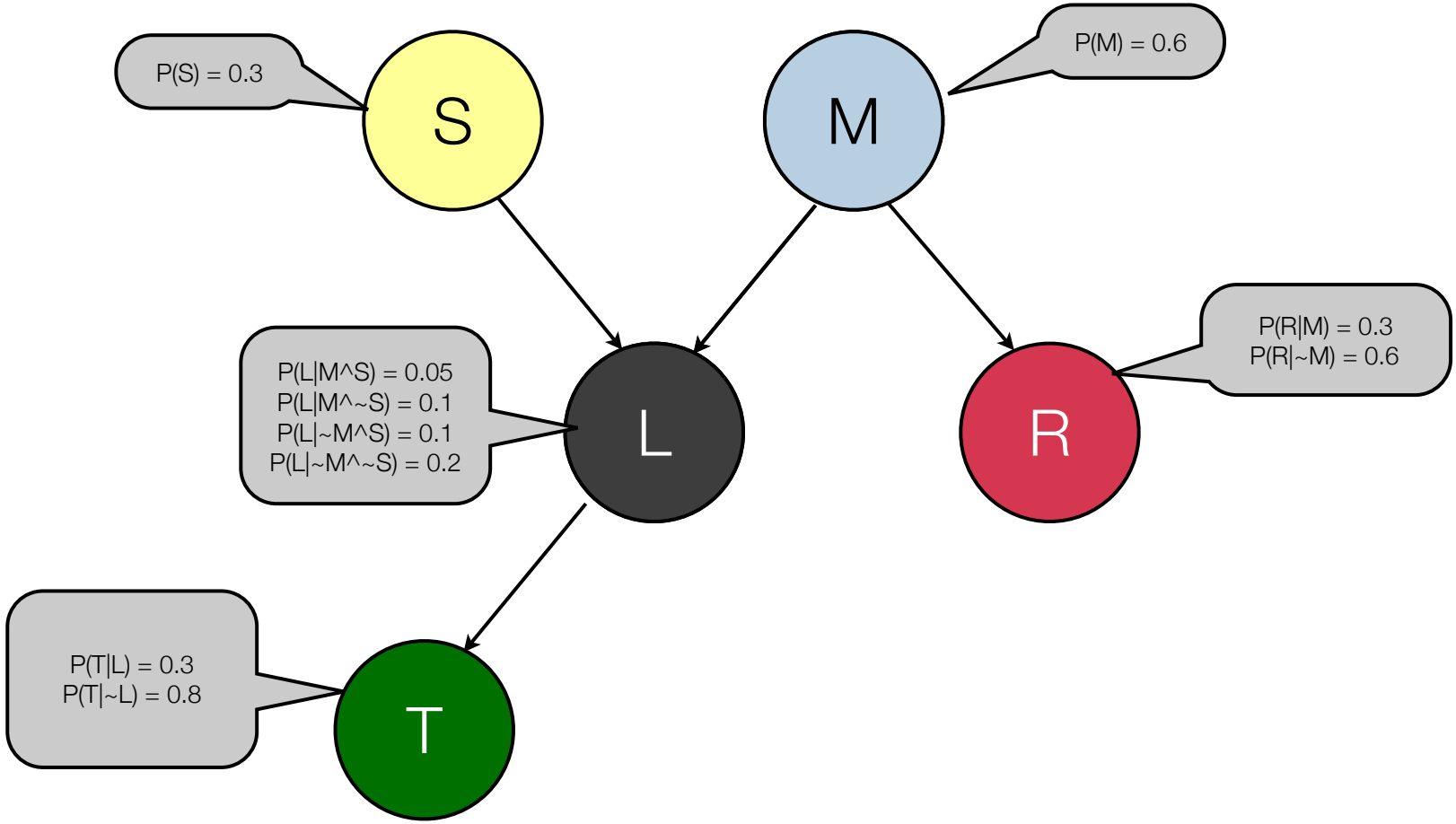
    P(S1's assignments | S3's assignments)
    ≠ P(S1's assignments | S2's assignments & S3's assignments)

Bayes Nets Formalized

- A Bayes net (also called a belief network) is an augmented directed acyclic graph, represented by the pair V , E where:
  - V is a set of vertices.
  - E is a set of directed edges joining vertices. No loops of any length are allowed.
- Each vertex in V contains the following information:
  - The name of a random variable
  - A probability distribution table indicating how the probability of this variable's values depends on all possible combinations of parental values.

Computing a Joint Entry

- Assume a new event T (class starts at 9h15)
- How to compute an entry in a joint distribution?
  - What is P(S ^ ~M ^ L ^ ~R ^ T)?



Computing a Joint Entry

- P(T ^ ~R ^ L ^ ~M ^ S) =

  P(T | ~R ^ L ^ ~M ^ S) * P(~R ^ L ^ ~M ^ S) =

  P(T | L) * P(~R^L^~M^S) =

  P(T | L) * P(~R | L^~M^S)*P(L^~M^S) =

  P(T | L) * P(~R | ~M) * P(L^~M^S) =

  P(T | L) * P(~R | ~M) * P(L | ~M^S) * P(~M^S) =

  P(T | L) * P(~R | ~M) * P(L | ~M^S) * P(~M | S)*P(S) =

  P(T | L) * P(~R | ~M) * P(L | ~M^S) * P(~M)*P(S)

The general case

- $P(X_1=x_1 \wedge X_2=x_2 \wedge ....X_{n-1}=x_{n-1} \wedge X_n=x_n) =$
  $P(X_n=x_n \wedge X_{n-1}=x_{n-1} \wedge ....X_2=x_2 \wedge X_1=x_1) =$
  $P(X_n=x_n \mid X_{n-1}=x_{n-1} \wedge ....X_2=x_2 \wedge X_1=x_1) * P(X_{n-1}=x_{n-1} \wedge .... X_2=x_2 \wedge X_1=x_1) =$
  $P(X_n=x_n \mid X_{n-1}=x_{n-1} \wedge ....X_2=x_2 \wedge X_1=x_1) * P(X_{n-1}=x_{n-1} \mid .... X_2=x_2 \wedge X_1=x_1) *$
  $P(X_{n-2}=x_{n-2} \wedge .... X_2=x_2 \wedge X_1=x_1) =$

  ...

  $= \prod P((X_i = x_i) \mid ((X_{i-1}=x_{i-1}) \wedge (X_1 = x_1)))$

  $= \prod P((X_i = x_i) \mid \text{Assignments of Parents}(X_i))$

- So any entry in joint pdf table can be computed. And so **any conditional probability** can be computed.

Case Study's

- Pathfinder system. (Heckerman 1991, Probabilistic Similarity Networks, MIT Press, Cambridge MA).
  - Diagnostic system for lymph-node diseases.
  - 60 diseases and 100 symptoms and test-results.
  - 14,000 probabilities
  - Expert consulted to make net.
    - 8 hours to determine variables.
    - 35 hours for net topology.
    - 40 hours for probability table values.
  - Apparently, the experts found it quite easy to invent the causal links and probabilities.
  - Pathfinder is now outperforming the world experts in diagnosis. Being extended to several dozen other medical domains.

What you should know

- The meanings and importance of independence and conditional independence.
- The definition of a Bayes net.
- Computing probabilities of assignments of variables (i.e. members of the joint p.d.f.) with a Bayes net.

Exercise

- For example, suppose that we are interested in diagnosing cancer in patients who visit a chest clinic.
    - Let A represent the event "Person has cancer"
    - Let B represent the event "Person is a smoker"
- Suppose we know the probability of the prior event A is 0.1 on the basis of past data (10% of patients entering the clinic turn out to have cancer). Thus:
    - $P(A)=0.1$
- We want to compute the probability of the posterior event $P(A|B)$.
- It is difficult to find this out directly. However, we are likely to know $P(B)$ by considering the percentage of patients who smoke suppose $P(B)=0.5$. We are also likely to know $P(B|A)$ by checking from our records the proportion of smokers among those diagnosed with cancer. Suppose $P(B|A)=0.8$.
- Use Bayes' rule to compute $P(A|B)$

Exercise

- Doing it using Erlang:
- First we represent the basic probability facts as Erlang functions:

```
p({patient, cancer}) -> 0.1;
p({patient, smoker}) -> 0.5.
```

- We then represent the conditional probabilities

```
cp({patient, smoker}, {patient, cancer}) -> 0.8;
cp(_, _) -> none.
```

- Next we write query rules that can be used to find various probabilities. There are three rules, covering the case where the probability is known, the conditional probability is known, or we can compute the conditional probability using Bayes theorem.

```
getp({A, B}) when is_tuple(A), is_tuple(B) ->
     case cp(A, B) of
        none ->
            Pba = cp(B, A),
            Pa = getp(A),
            Pb = getp(B),
            Pba * Pa / Pb;
        X ->
            X
     end;
getp(A) -> p(A).
```

- We now have a simple system for representing and querying knowledge expressed as probabilities.

```
1> bayes:getp({{patient, cancer}, {patient, smoker}});
```

Exercise 2

- Consider the next probabilities:

  p(bonus) = 0.6

  p(not_bonus) = 0.4

  cp(money ,bonus) = 0.8

  cp(money, not_bonus) = 0.3

  cp(hawaii, money) = 0.7

  cp(Hawaii, not_money) = 0.1

  cp(san_francisco, money) = 0.2

  cp(san_francisco, not_money) = 0.5

  cp(wierd_people, san_francisco) = 0.95

  cp(wierd_people, not_san_francisco) = 0.6

  cp(surfing, Hawaii) = 0.75

  cp(surfing, not_Hawaii) = 0.2

  P(surfing) = ?

Universidade de Aveiro

Introdução à Inteligência Artificial – MIECT
Inteligência Artificial – LEI

Programação ao Estilo Funcional em Python

Ano lectivo 2020/2021

Regente: Luís Seabra Lopes

Python

- Características principais da linguagem de programação Python:
  - Interpretada
  - Interactiva
  - Portável
  - Funcional
  - Orientada a objectos
  - Implementação aberta

Python (cont.)

- Objectivos da linguagem
  - Simplicitade sem prejuizo da utilidade
  - Programação modular
  - Legibilidade
  - Desenvolvimento rápido
  - Facilidade de integração, nomeadamente com outras linguagens

Python é multi-paradigma

Programação funcional

Expressões lambda
Funções de ordem superior
Listas com sintaxe simplificada
Listas de compreensão
Iteradores

Programação OO

Classes
Objectos
Métodos
Herança

Programação imperativa / modular

Instrução de atribuição
Sequências de instruções
Análise condicional (if-elif-else)
Ciclos for, while
Sistema de módulos

Python - história

- Criada em 1989-1991 por Guido Van Rossum
  - Tem como predecessora directa a linguagem imperativa/estruturada ABC, tendo também sido influenciada pela linguagem Modula-3
  - O nome da linguagem tem origem no "*Monty Python's Flying Circus*"
- Inicialmente desenhada como uma linguagem de *scripting* no sistema operativo Amoeba

Python versus Java

- Código mais conciso
  - Os espaços brancos são sintaticamente relevantes
- Verificação de tipos dinâmica
- Desenvolvimento mais rápido
- Não compila para código nativo

- Mas, programas mais lentos …

Python – áreas de aplicação

- Interligação de sistemas
- Aplicações gráficas
- Aplicações para bases de dados
- Multimédia
- Internet protocol / Web
- Robótica & inteligência artificial

Python - aplicações

- Google - fortemente baseado em Python, segundo o lema inicial:
  - "Python where we can, C++ where we must"
- Zope – servidor de aplicações para a Web, de código aberto, totalmente escrito em Python
- ROS (Robot Operating System) – Python é uma das linguagens suportadas, juntamente com C++ e Lisp

Python – aplicações (cont.)

- Inteligência Artificial – Python é uma das linguagens com popularidade crescente nesta área
  - Os exemplos do livro "*Artificial Intelligence: a Modern Approach*" estão implementados em Python, Java e Lisp
  - CWM – Máquina de inferência de propósito geral para a Web Semântica, totalmente desenvolvido em Python por Tim Berners-Lee
  - Google (algoritmo de ordenação de páginas, etc.) também é um exemplo aqui!

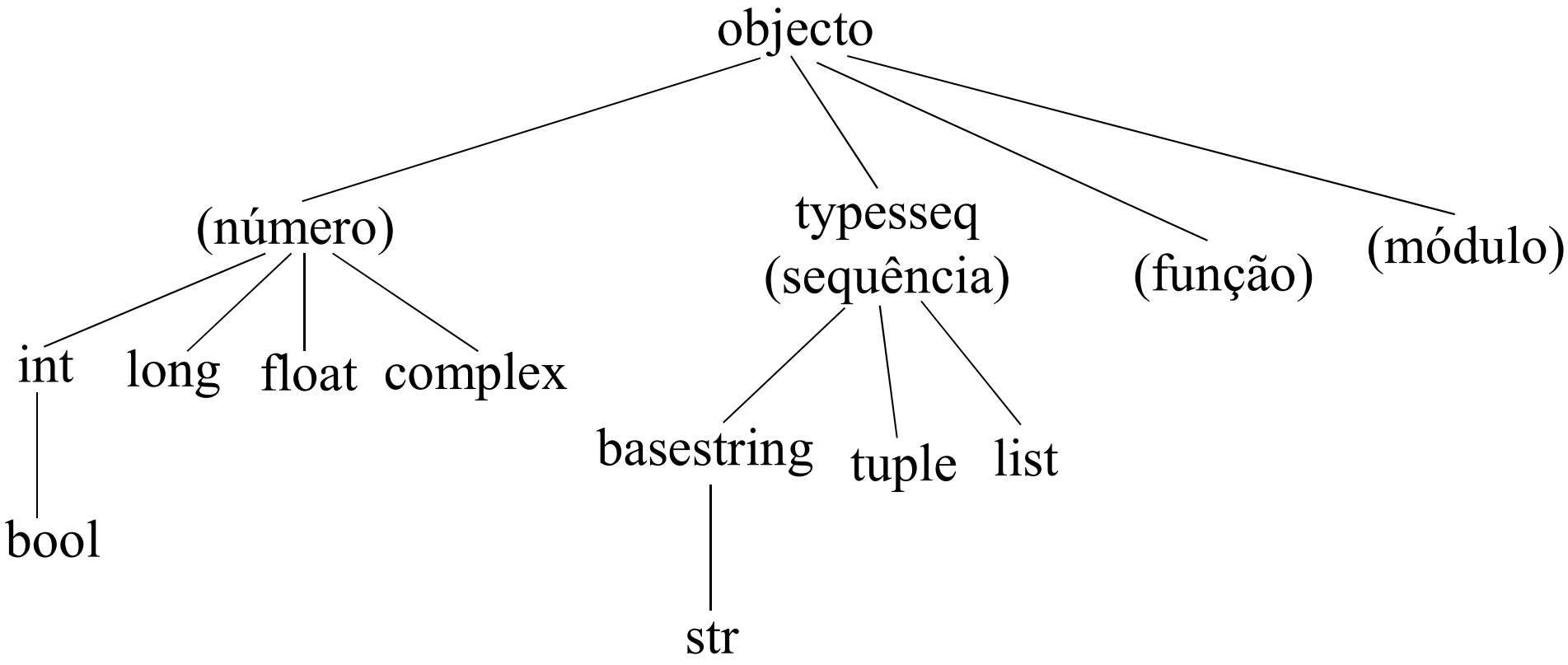
Pyhton - utilização

- Está incluída nas principais distribuições de Linux
  - Está também disponível para outras plataformas
- Pode ser obtida, juntamente com documentação em www.python.org
- IDLE – ambiente de desenvolvimento para Python

Dados, ou “objectos”

- Objecto – no contexto de Pyhton, esta designação é aplicada a qualquer dado que possa ser armazenado numa variável, ou passado como parâmetro a uma função
- Cada objecto é caracterizado por
  - Identidade ou referência (identifica a posição de memória onde está armazenado),
  - Tipo e
  - Valor
- Alguns tipos de objectos podem ter atributos e métodos
- Alguns tipos (classes) de objectos podem ter sub-tipos (sub-classes)

Objectos


objecto
(número)
typesseq
(sequência)
(função)
(módulo)
int
long
float
complex
bool
basestring
tuple
list
str

Tipos de dados elementares

- bool
  - Tem os valores True e False
  - Mas, valores de diferentes tipos podem também ser usados como valores de verdade
    - Exemplo: o inteiro 0 (zero) vale o mesmo que False
  - Função bool() converte qualquer valor para bool

Tipos de dados elementares (cont.)

- Números
  - `int` – números inteiros de 32 bits
    - equivale ao `long` da linguagem C
  - `long` – inteiros de precisão ilimitada
  - `float` – números reais
    - equivale ao `double` da linguagem C
  - `complex` – números reais, em que a parte real e a parte imaginária são números reais
    - Exemplos: `1.5+0.5j`, `7.2+4.0J`
    - As partes de um número complexo `z` podem ser obtidas através das expressões `z.real` e `z.imag`

Tipos de dados elementares (cont.)

- Números (cont.)
  - Funcionam com os operadores habituais: +, -, *, /
  - Quociente da divisão inteira: //
  - O operador de divisão (/) também fornece um quociente da divisão inteira, mas neste caso o resultado é arredondado para o lado de $-\infty$
    - Exemplos: `1/2` → `0`, `1/(-2)` → `-1`
  - Resto da divisão inteira: %
  - Potência: ** (equivale à função pré-definida `pow()`)
    - Exemplo: `5 ** 3` → `125`
  - Funções de conversão:
    - `int()`, `long()`, `float()`

Sequências de dados

- Cadeias de caracteres (str)
  - Podem aparecer entre aspas (") ou pelicas (')
  - Exemplos:
    - "abc d x"
    - 'abc d x'
    - 'Ele disse "sim" !'
    - "A palavra 'Maria' é um nome próprio.“
  - As cadeias de caracteres são imutáveis: não podemos modificar os caracteres em posições individuais

Sequências de dados (cont.)

- Tuplos (tuple) – agregados ou composições de vários elementos, que podem ser de tipos diferentes
  - Funcionam como registos ou estruturas sem nome
  - São imutáveis: não podemos modificar elementos em posições individuais do tuplo
  - Os elementos são separados por vírgulas (,) e opcionalmente delimitados por parênteses curvos
  - Exemplos:
    - 1, 2, 'a'
    - ("maria", 33)
    - 27,
    - 'lisboa', ("colinas", 7)
    - ( )

Sequências de dados (cont.)

- Listas (list) – sequências de elementos, que podem ser de tipos diferentes
  - Combinam a funcionalidade usual das listas na programação declarativa com a funcionalidade usual dos vectores na programação imperativa
  - É possível modificar elementos individuais das listas
  - Os elementos são separados por vírgulas (,) e delimitados por parênteses rectos
  - Exemplos:
    - [1, 2, 'a']
    - [("maria", 33), (“josé", 40)]
    - ['lisboa', [7, "colinas"]]
    - [ ]

Sequências de dados (cont.)

- Algumas funcionalidades básicas
  - `x in s`
    - Expressão que retorna `True` se existir um elemento na sequência `s` que seja igual a `x`, caso contrário retorna `False`
  - `x not in s`
    - O contrário da anterior
  - `len(s)`
    - Função que retorna o comprimento da sequência `s`
  - `s1 + s2`
    - Retorna a concatenação das sequências `s1` e `s2`

Função pré-definida `type()`

- Dado um objecto qualquer, devolve o respectivo tipo

Variáveis

- Não são declaradas
- Não têm tipo
- Praticamente tudo pode ser atribuído a uma variável (incluindo funções, módulos e classes)
- Similarmente ao que acontece nas linguagens imperativas, e ao contrário do que acontece nas linguagens funcionais, em Python o valor das variáveis pode ser alterado
- Não se pode ler o valor da variável se ela não tiver sido inicializada

Instrução de atribuição - I

- Tal como é habitual na programação imperativa, e ao contrário do habitual na programação declarativa, Python possui instrução de atribuição
- Exemplos:
  - `n = 10`
  - `a = b = c = 0`
  - `x = 7.25`
  - `cad = "cadeia"`
  - `t = (n, x)`
  - `lista = [1, 2, 'quatro', 8.0]`

Instrução de atribuição - II

- Podemos usar a operação de atribuição para decompor estruturas
- Exemplo:
  - `triplo = (1, 2, 3)`
  - `(i, j, k) = triplo`
    - Como resultado, `i=1`, `j=2`, `k=3`
- Outro exemplo:
  - `(q,r) = divmod(16,3)`
    - A função pré-definida `divmod()` devolve um tuplo com o quociente e o resto de uma divisão inteira
    - Como resultado, `q=5`, `r=1`

Operadores de comparação

- Igual: ==
- Diferente: != (ou <>)
- Menor/maior: <, <=, >, >=
- Objectos de tipos diferentes nunca são iguais
  - Exceptuam-se os diferentes tipos de números
- Comparação de sequências baseia-se num critério lexicográfico
- Mais info: *Python Tutorial*, sec. 5.8

Operadores lógicos

- Conjunção: `and`
- Disjunção: `or`
- Negação: `not`

- Nota: Na conjunção e na disjunção, o segundo argumento só é avaliado se for necessário para determinar o resultado

Acesso a sequências

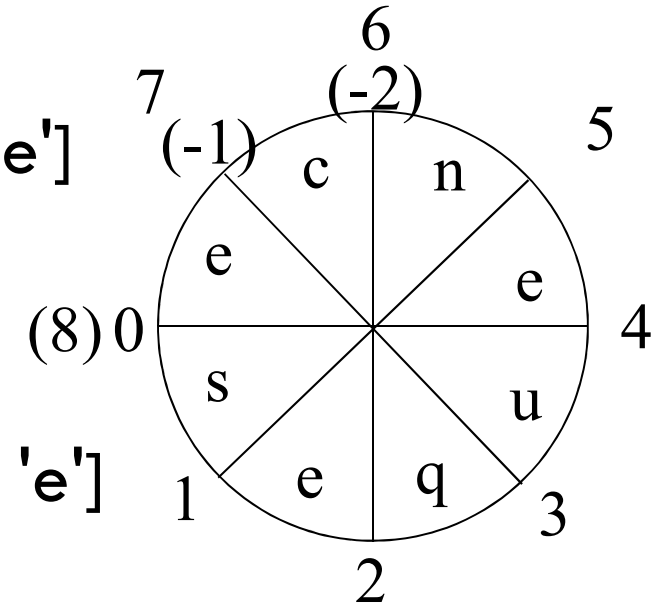
- Os elementos das sequências são acedidos através de índices inteiros consecutivos
  - O primeiro elemento tem o índice 0 (zero)
- Exemplo:
  - a = [12, 4, "abc"]
  - a[0] → 12
  - a[2] → "abc"

Acesso a sequências (cont.)

- É possível extrair "fatias" das sequências
  - Formato: `seq[inf:sup]` – fatia da sequência `seq`, compreendida entre o elemento com índice `inf` e o elemento com índice `sup-1`
  - A fatia é uma cópia do conteúdo da sequência original entre `inf` e `sup-1`
- A indexação é circular, o que permite aceder ao último elemento da sequência `s` pelo índice `len(s)-1` ou simplesmente pelo índice `-1`

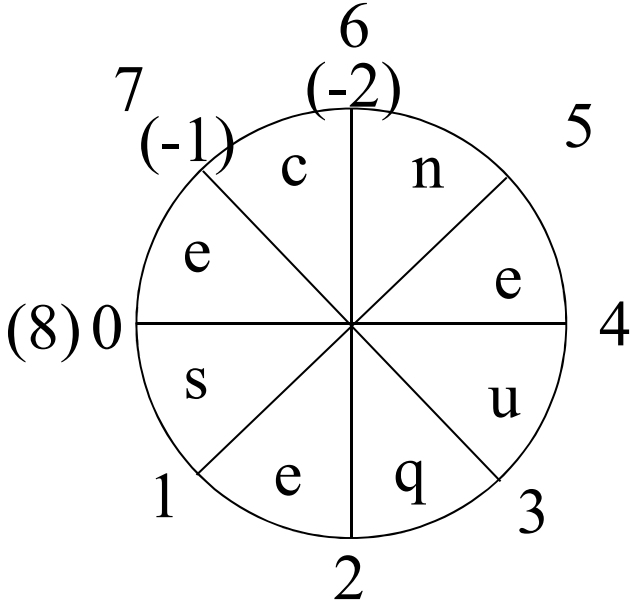
Acesso a sequências (cont.)

- Exemplos:
  - lista = ['s', 'e', 'q', 'u', 'e', 'n', 'c', 'e']
  - lista[0] → 's'
  - lista[2:5] → ['q', 'u', 'e']
  - lista[1:] → ['e', 'q', 'u', 'e', 'n', 'c', 'e']
  - lista[-2:] → ['c', 'e']
  - lista[:3] → ['s', 'e', 'q']
  - lista[:] → ['s', 'e', 'q', 'u', 'e', 'n', 'c', 'e']



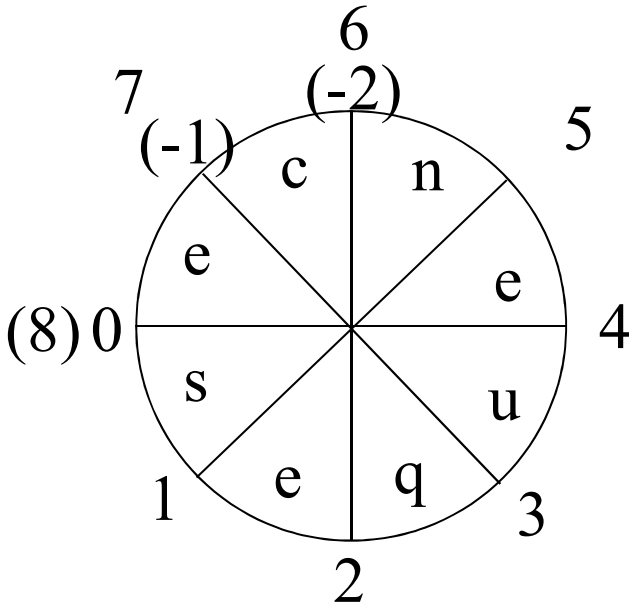
Modificação de listas

- Faz-se por atribuição
  - lista[0] = 'p'
  - lista → ['p', 'e', 'q', 'u', 'e', 'n', 'c', 'e']
- Pode-se modificar fatias
  - lista[-2:] = ['o', 's']
  - lista → ['p', 'e', 'q', 'u', 'e', 'n', 'o', 's']



Modificação de listas (cont.)

- Pode-se remover uma fatia
  - lista[0:3] = [ ]
  - lista → ['u', 'e', 'n', 'o', 's']
- Inserir uma fatia
  - lista[3:3] = ['a', 'a']
  - lista → ['u', 'e', 'n', 'a', 'a', 'o', 's']
- Substituir por fatia de tamanho diferente
  - lista[1:6] = ['c', 'h']
  - lista → ['u', 'c', 'h', 's']



Instrução de atribuição - III

- Detalhe importante:
  - A instrução de atribuição, em vez de copiar valores, limita-se a associar um dado identificador a um dado objecto
  - Assim, a atribuição de uma variável `x` a uma variável `y` apenas tem como resultado associar `y` ao mesmo objecto ao qual `x` já estava associada
  - No caso de objectos mutáveis, há que ter cuidado com efeitos como este:
    - `a = [1,2,3]`
    - `b = a`
    - `b[1:2] = []`
    - `a` → `[1,3]`

Análise condicional: instrução if-elif-else

- Síntaxe:

```
if <condição_1>:
        <instruções_1>
elif <condição_2>:
        <instruções_2>
else:
        <instruções_n>
```

- Notas:
  - Pode haver 0 (zero) ou mais ramos **elif** (= else if)
  - O ramo **elif** / **else** é opcional

Análise condicional: expressão if-else

- Síntaxe
  - <*expressão1*> if <*condição*> else <*expressão2*>
- Exemplos:
  - x if x>=0 else -x
  - 'pos' if a>0 else 'nul' if a==0 else 'neg'

Definição de funções

- Inicia-se com a palavra `def`
- Passagem de parâmetros "por referência dos objectos" (ver adiante)
- A instrução `return` permite definir o resultado da função
  - Quando não é indicado um resultado no `return`, ou quando não há `return`, o resultado é `None` (um identificador pré-definido)
  - Funções sem `return`, ou com `return` vazio, são também conhecidas como "*procedimentos*"
- As funções podem ser recursivas
- As funções são objectos do tipo `function`

Definição de funções – exemplos (I)

```
# retorna o primeiro elemento de uma lista, caso exista,
# ou None, caso contrário
def cabeca(lista):
    if lista==[]:
        return None
    return lista[0]

# retorna um tuplo com o primeiro elemento de uma lista
# e a lista dos restantes (retorna None caso a lista seja vazia)
def cabeca_e_cauda(lista):
    if lista==[]:
        return None
    return (lista[0],lista[1:])
```

Definição de funções – exemplos (II)

```
# concatena duas listas
def concatenar(lista1,lista2):
    conc = lista2[:]
    conc[:0] = lista1
    return conc
```

```
# Nota: a concatenação de listas é disponibilizada
# em Python através do operador +
```

Passagem de parâmetros

- Os parâmetros são passados às funções segundo um mecanismo de "passagem por valor" ("*call by value*")
- Mas, detalhe importante: Neste contexto, o valor é na verdade a referência do objecto!!!
  - Se atribuirmos um novo objecto a uma variável passada por parâmetro, essa atribuição ocorre apenas no espaço de nomes da função (acetato seguinte, esquerda)
  - Se modificarmos um objecto passado por parâmetro (por exemplo, apagar um elemento de uma lista), isto não altera a referência do objecto, e portanto vai permanecer após o retorno da função (acetato seguinte, direita)

Passagem de parâmetros – exemplos (I)

```
>>> def incr(x):
...     x=x+1
...     return x
...
>>> n=10
>>> incr(n)
11
>>> n
10
```

```
>>> def acresc(l,x):
...     l[0:0]=[x]
...     return l
...
>>> lista=[5,12]
>>> acresc(lista,30)
[30,5,12]
>>> lista
[30,5,12]
```

Passagem de parâmetros – exemplos (II)

- O problema anterior resolve-se trabalhando sobre uma cópia local
- No caso de uma lista l, a fatia l[:] dá-nos uma cópia integral

```
>>> def acresc(l,x):
...     aux=l[:]
...     aux[0:0]=[x]
...     return aux
...
>>> lista=[5,12]
>>> acresc(lista,30)
[30,5,12]
>>> lista
[5,12]
```

Parâmetros com valores por defeito

- Exemplo:

```
def custoCombustivel(dist,cons=8,preco=1.5):
    return (dist/100.0) * cons * preco
```

- Na chamada, pode-se omitir alguns parâmetros, caso em que temos que indicar os nomes dos que estão para a frente:

```
>>> custoCombustivel(30,preco=1.6)
2.56
```

- Nas chamadas em que são fornecidos os primeiros *k* de *n* parâmetros, não é preciso indicar os nomes:

```
>>> custoCombustivel(20,7.5)
2.25
```

Funções recursivas – exemplos - I

```
# devolve factorial de um número n
def factorial(n):
    if n==0:
        return 1
    if n>0:
        return n*factorial(n-1)

# devolve o comprimento de uma lista
def comprimento(lista):
    if lista==[]:
        return 0
    return 1+comprimento(lista[1:])
```

```
    comprimento([1,2,3])
=
    1 + comprimento([2,3])
=
    1 +( 1 + comprimento([3]))
=
    1 + (1 + (1 + comprimento([]))
=
    1 + (1 + (1 + 0))
=
    1 + (1 + 1 )
=
    1 + 2
=
    3
```

Funções recursivas – exemplos - II

```
# verifica se um elemento é membro de uma lista
def membro(x,lista):
    if l==[]:
        return False
    return (lista[0]==x) or membro(x,lista[1:])

# devolve uma lista com os elementos da lista
# de entrada por ordem inversa
def inverter(lista):
    if lista==[]:
        return []
    inv = inverter(lista[1:])
    inv[len(inv):] = [lista[0]]
    return inv
```

Expressões Lambda – I

- São expressões cujo valor é uma função
- São um "ingrediente" clássico da programação funcional
- Exemplos:
  - `lambda x : x+1`
    - Função que dado um valor `x`, devolve `x+1`
  - `m = lambda x, y : math.sqrt(x**2+y**2)`
    - Função que calcula o módulo de um vector (x,y), função esta atribuída à variável `m`
  - `(lambda lista : lista[-1]-lista[0]) [5,7,11,19,38]`
    - Função que calcula a diferença entre o primeiro e o último elemento de uma lista, função esta logo aplicada a uma lista concreta
    - Resultado: `33`

Expressões Lambda – II

- Como qualquer objecto, uma expressão lambda pode ser passada como parâmetro a uma função
- Exemplo:
  - Uma função h que, dada uma função f e um valor x, produz f(x)*x

```
def h(f,x): return f(x)*x
```

  - Exemplo de utilização:

```
h(lambda x : x+1,7)
```

  - Resultado: 56

Expressões Lambda – III

- As expressões lambda podem ser produzidas por outras funções:
  - Exemplo: Dado um inteiro n, a função seguinte produz uma função que soma n à sua entrada

```
def faz_incrementador(n):
    return lambda x : x+n
```

  - Exemplo de utilização:

```
suc = faz_incrementador(1)
suc(10)
```

  - Resultado: 11

Expressões Lambda – IV

- As expressões lambda são também conhecidas como expressões funcionais
- As funções que recebem expressões lambda como entrada e/ou produzem expressões lambda como saída são conhecidas como funções de ordem superior

- Nota importante: As expressões lambda só são úteis enquanto são simples. Uma função complexa merece ser escrita de forma clara numa definição (`def`) à parte

Exercício

- Pesquisa dicotómica de uma raiz de uma função *f* num intervalo [*a*,*b*]
  - Assume-se que a função é contínua em [*a*,*b*]
  - Assume-se que *f*(*a*) e *f*(*b*) são de sinais opostos
  - Implementa-se uma função que divide ao meio o intervalo e se chama a si própria recursivamente sobre a metade do intervalo em cujos extremos *f* tem valores de sinal contrário
  - A função *f* é um parâmetro de entrada da função que procura a raiz
  - O processo termina quando o valor *b*-*a* for suficientemente pequeno

Aplicar uma função a uma lista

- Aplicar uma função f a cada um dos elementos de uma lista, devolvendo uma lista com os resultados:

```
def aplicar(f,lista):
    if lista=[]:
        return []
    return [f(lista[0])] + aplicar(f,lista[1:])
```

- Exemplo de utilização: Dada uma lista de inteiros, obter a lista dos dobros

```
aplicar(lambda x : 2*x, [2,-4,17])
```

  - Resultado: [4,-8,34]

- Corresponde à função pré-definida map()
  - Em Pyhton3, esta função retorna um iterador que pode ser convertido para lista

Filtrar uma lista

- Dada uma função booleana `f` e uma lista, devolve uma lista com os elementos da lista de entrada para os quais `f` devolve `True`:

```
def filtrar(f,lista):
    if lista=[]:
        return []
    if f(lista[0]):
        return [lista[0]] + filtrar(f,lista[1:])
    return filtrar(f,lista[1:])
```

- Exemplo: Dada uma lista de inteiros, obter a lista dos pares

```
filtrar(lambda x : x%2==0, [2,-4,17])
```

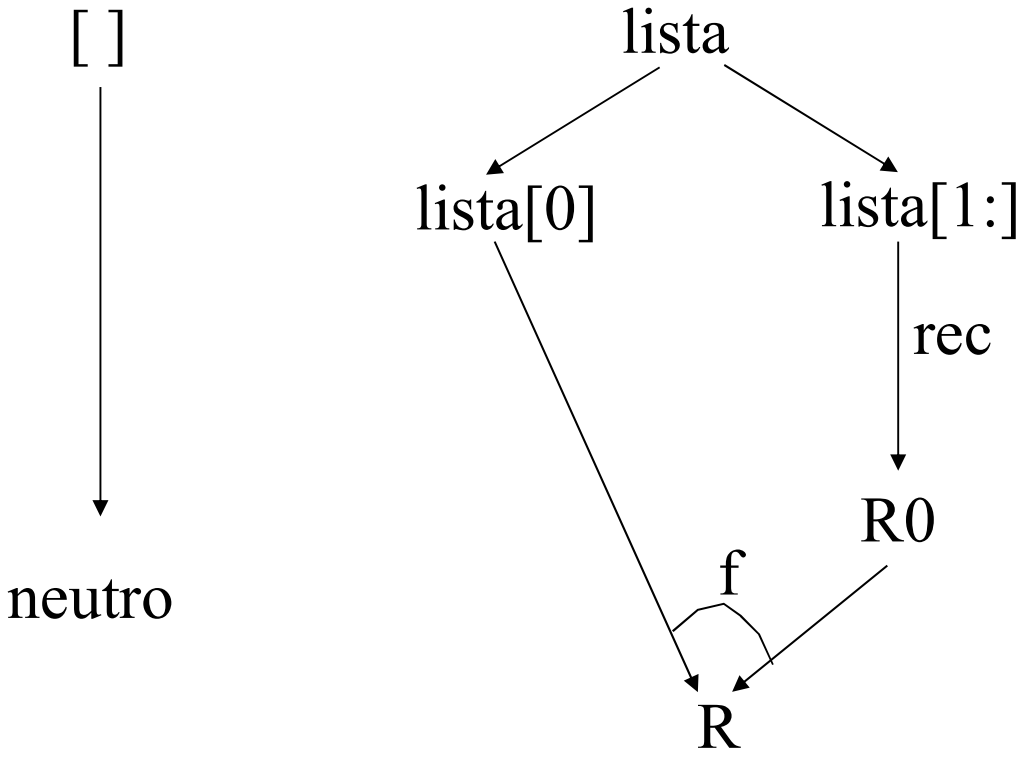
  - Resultado: [2,-4]

- Corresponde à função pré-definida `filter()`
  - Em Pyhton3, esta função retorna um iterador que pode ser convertido para lista

Reduzir uma lista a um valor - I

- Muitos procedimentos que actuam sobre listas têm em comum a seguinte estrutura:
  - No caso de a `lista` ser vazia, o resultado é um valor "neutro" pré-definido;
  - No caso de a `lista` ser não vazia, o resultado da função depende de combinar a cabeça da lista (`lista[0]`) com o resultado da chamada recursiva sobre os restantes elementos (`lista[1:]`).

Reduzir uma lista a um valor - II



Exemplo:
Se quisermos somar os elementos de uma lista de inteiros:

```
neutro = 0
f = lambda x, s : x+s
```

Reduzir uma lista a um valor - III

- Dada uma função de combinação f, uma lista e um valor neutro, devolve a redução da lista:

```
def reduzir(f,lista,neutro):
    if lista=[]:
        return neutro
    return f(lista[0],reduzir(f,lista[1:],neutro))
```

- Exemplo: Dada uma lista de inteiros, obter a respectiva soma

```
reduzir(lambda x, s : x+s, [2,-4,17], 0)
```

  - Resultado: 15

- Corresponde à função pré-definida reduce()
  - Em Python3, esta função está na biblioteca functools

Listas de compreensão

- Do inglês "*list comprehension*"
- Mecanismo compacto para processar alguns ou todos os elementos numa lista
  - "importado" da linguagem funcional Haskell
  - Pode ser aplicado a listas, tuplos e cadeias de caracteres
  - O resultado é uma lista
- Síntaxe (caso simples):

  [<expr> for <var> in <sequência> if <condição>]

Listas de compreensão (cont.)

- Podem funcionar como a função map()
- Exemplo: Obter os quadrados dos elementos de uma dada lista:

```
>>> map(lambda x : x**2, [2,3,7])
[4,9,49]
>>> [x**2 for x in [2,3,7]]
[4,9,49]
```

Listas de compreensão (cont.)

- Podem funcionar como a função filter()
- Exemplo: Obter os elementos pares existentes numa dada lista

```
>>> filter(lambda x : x%2==0, [2,3,7,6])
[2,6]
>>> [x for x in [2,3,7,6] if x%2==0]
[2,6]
```

Listas de compreensão (cont.)

- Podem combinar as funcionalidades de map() e filter()
- Exemplo: Obter os quadrados de todos os elementos positivos de uma dada lista

```
>>> [x**2 for x in [3,-7,6] if x>0]
[9,36]
```

Listas de compreensão (cont.)

- Podem percorrer várias sequências
- Exemplo: Obter todos os pares de elementos, um de uma lista e outro de outra, em que a soma seja ímpar

```
>>> [(x,y) for x in [3,7,6]
...             for y in [2,8,9] if (x+y)%2!=0]
[(3,2), (3,8), (7,2), (7,8), (6,9)]
```

Classes

- As classes em Python possuem as características mais comuns nas linguagens orientadas a objectos
  - Uma classe define um conjunto de objectos caracterizados por diversos atributos e métodos
  - É possível definir hierarquias de classes com herança
- As classes surgem na linguagem Python com pouca sintaxe adicional

Classes (cont.)

- Sintaxe:

```
class <nome-classe>:
    <declaração-1>
    ...
    <declaração-N>
```

Classes – exemplo

```
class UmTeste:
    def dizer_ola(self):
        print "Ola"
```

Por convenção, as palavras no nome de uma classe iniciam-se com maiúscula

Exemplo de definição de um método

- Utilização

```
>>> x = UmTeste()
>>> x.dizer_ola()
Ola
```

Criação de uma instância e atribuição a uma variável

Invocação do método

Classes com construtor – exemplo

```
class Complexo:
    def __init__(self,real,imag):
        self.r = real
        self.i = imag
```

O construtor é o método que inicializa um objecto no momento da sua criação; chama-se obrigatoriamente "__init__";

O primeiro parâmetro (self) de qualquer método é a própria instância na qual o método é chamado

- Utilização

```
>>> c = Complexo(-1.5,13.1)
>>> c.r, c.i
(-1.5,13.1)
```

Criação de uma instância e atribuição a uma variável

Classes – atributos

- No exemplo anterior, a classe `Complexo` tem os atributos `r` e `i`
- Tal como acontece com as variáveis normais, também os atributos das classes não são declarados
- Acesso aos atributos numa instância é feito com o ponto (“.”), como no exemplo anterior
- A todo o tempo, pode-se criar um novo atributo numa instância, bastando para isso atribuir-lhe um valor

Classes derivadas / herança

- Sintaxe:

  class <*nome-classe*> (<nome-classe-mãe>):

  <*declaração-1*>

  …

  <*declaração-N*>

- A classe derivada herda os métodos e atributos da classe mãe
- É possível uma classe ter várias classes mães

Exemplo de aplicação: expressões aritméticas

- Considere a seguinte expressão:

  2*x+1

- Pode-se representar em Python da seguinte forma:

  Soma(Produto(Const(2),Var()),Const(1)

- Em que Soma, Produto, Const e Var são classes definidas pelo programador para representar
  - somas de expressões,
  - produtos de expressões,
  - constantes e
  - ocorrências da variável

Exemplo de aplicação: expressões aritméticas (cont.)

- Como definir os construtores das classes referidas?
- Como definir métodos para avaliar as expressões, dado um certo valor da variável?
- Como definir métodos para simplificar expressões?
- Como definir métodos para derivar expressões?

Exemplo de aplicação: expressões aritméticas (cont.)

- Exemplo:

```
class Soma:
    def __init__(self,e1,e2):
        self.arg1 = e1
        self.arg2 = e2
    def avaliar(self,v):
        return self.arg1.avaliar(v) + self.arg2.avaliar(v)
```

Classes – conversão para cadeia de caracteres

- Relevante para visualização
- Consegue-se através da implementação de um método "__str__()" (nome obrigatório)
- Na classe Soma (ver acetato anterior), poderia ser assim:

```
def __str__(self):
    return str(self.arg1) + "+" + str(self.arg2)
```

- Utilização:

```
>>> s = Soma(Const(2),Const(1))
>>> str(s)
2+1
```

Métodos e atributos pré-definidos

- Métodos
  - `__init__()` – construtor
  - `__str__()` – implementa a conversão para cadeia de caractéres; suporta a função de conversão `str()`
  - `__repr__()` – define a representação em cadeia de caractéres que aparece na consola do interpretador; suporta a função `repr()`
- Atributos
  - `__class__` - identifica a classe de um dado objecto
    - Também se pode usar a função `isinstance(<instance>,<class>)`

O tipo list de Python é uma classe

- Tem os seguintes métodos:
  - list.append(x) – acrescenta x ao fim da lista
  - list.extend(L) – acrescenta elementos da lista L no fim da lista
  - list.insert(i,x) – insere x na posição i
  - list.remove(x) – remove a primeira ocorrência de x
  - list.index(x) – remove a posição da primeira ocorrência de x
  - list.sort() – ordena a lista (modifica a lista)
  - …

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, "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 involve [Albus,1995]:
  - Como adquir, 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 planear 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 xadrês
- 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 (reconstruia a estrutura de moléculas orgânicas)
- 1986 – Robôs comportamentalistas
- 1986 – Ressurgimentos das redes neuronais
- 1997 – O IBM Deep Blue vence uma partida de xadrês 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

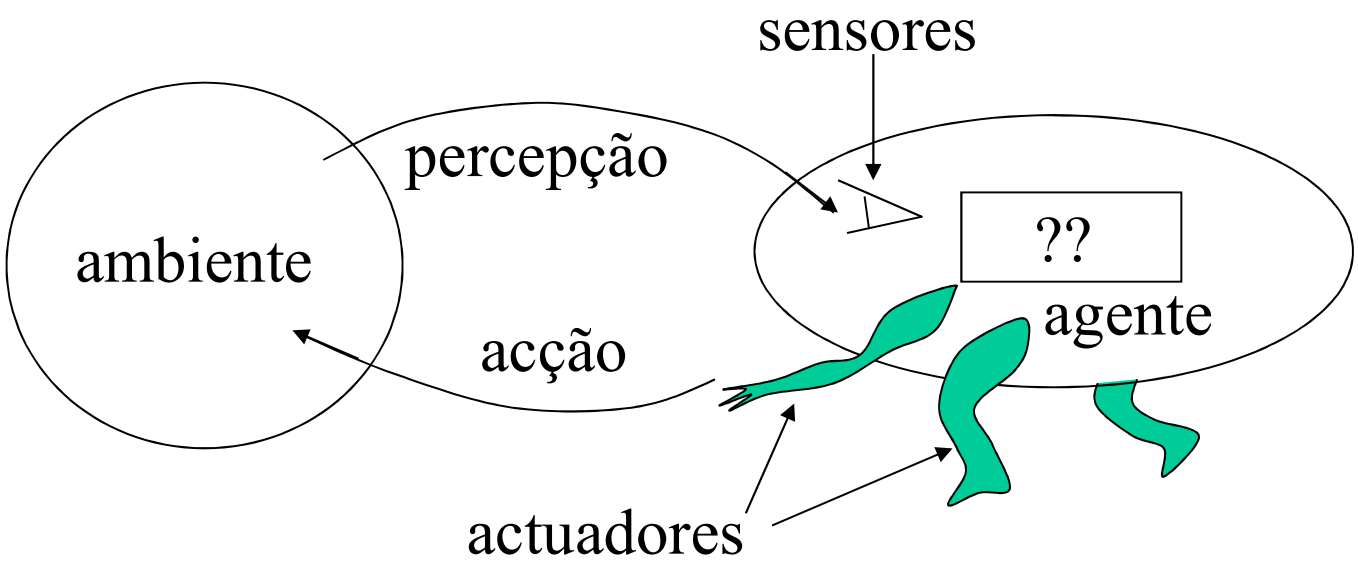
Definição de "Agente"

- Nesta disciplina estudamos técnicas úteis no desenvolvimento de "agentes inteligentes"
- Segundo www.dictionary.com, 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 caractéres chineses a serem introduzidas na sala e respostas inteligentes a essas mensagens a serem devolvidas do interior da sala.

O argumento de Searle

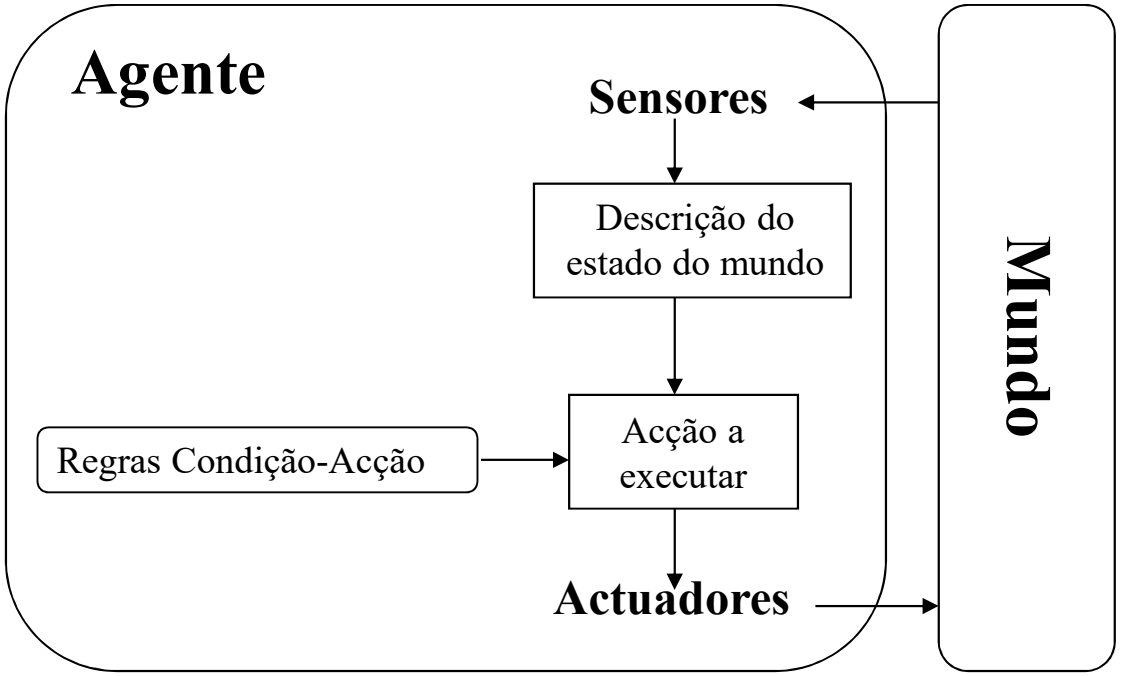
- O humano não percebe chinês
- A sala não percebe chinês
- O livro de regras e as folhas de papel também não percebem chinês
- Logo, não há qualquer compreensão de chinês naquela sala

- 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 arquitecturas de agentes

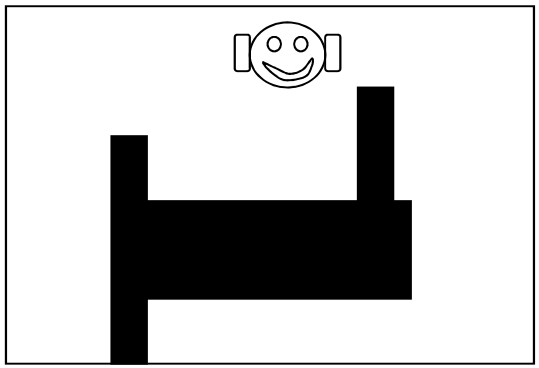
- Tipos de agentes
  - Reactivos simples
  - Reactivos com estado
  - Deliberativos orientados por objectivos
  - Deliberativos orientados por funções de utilidades
- Arquitecturas
  - Subsunção
  - Três torres
  - Três camadas
  - CARL

Agente reactivo: simples

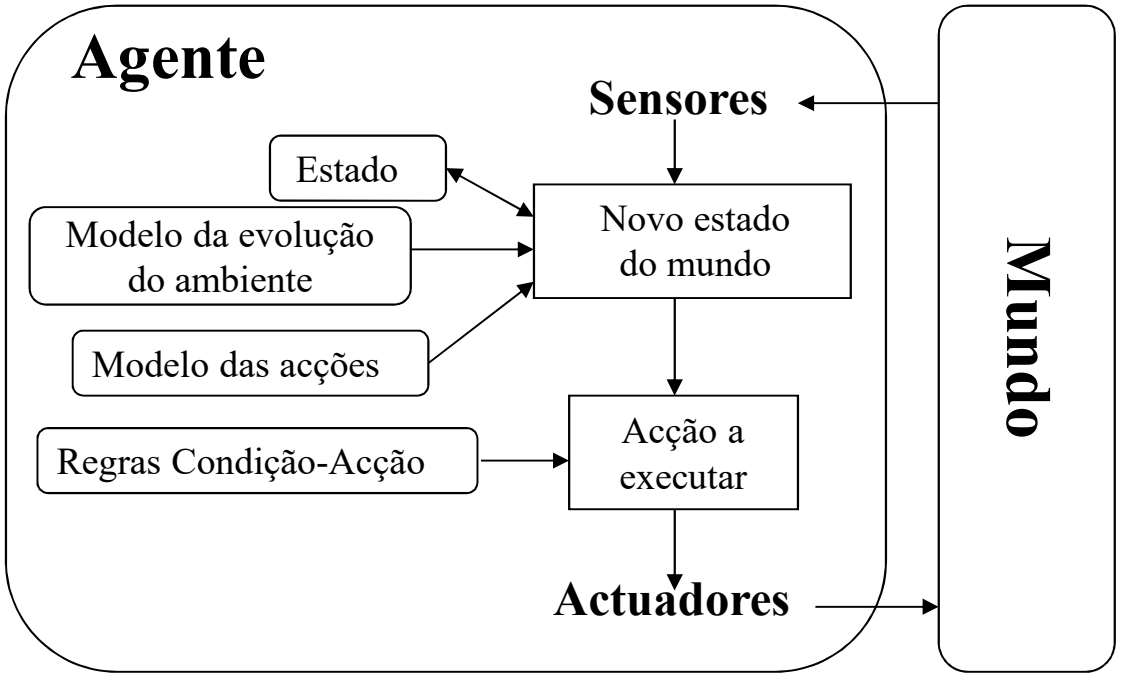


- O conceito de "regra de condição-acção" é também conhecido como "regra de situação-acção" ou "regra de produção".
- Os agentes ou sistemas reactivos simples são também conhecidos como "sistemas de estímulo-resposta" ou "sistemas de produção"

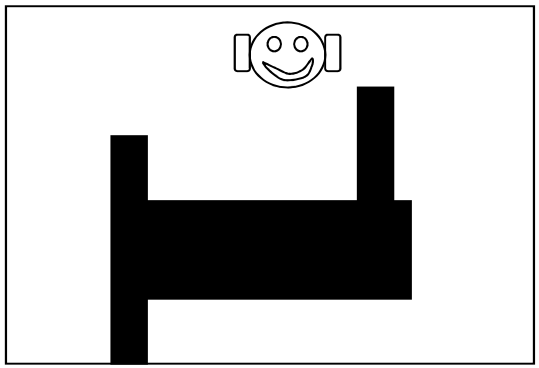
Representando a percepção através de um vector de características



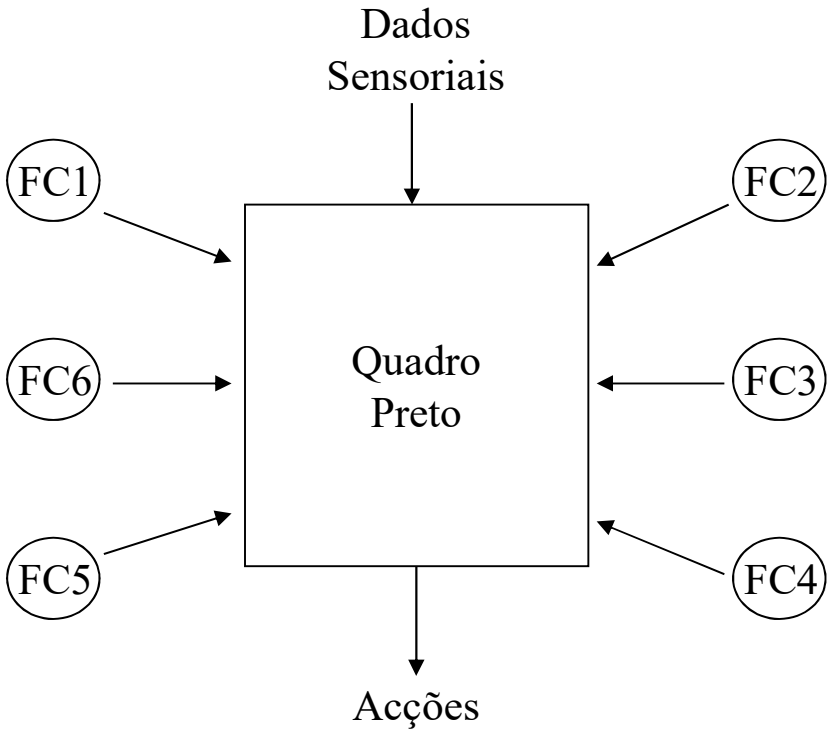
Agente reactivo: com estado interno



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

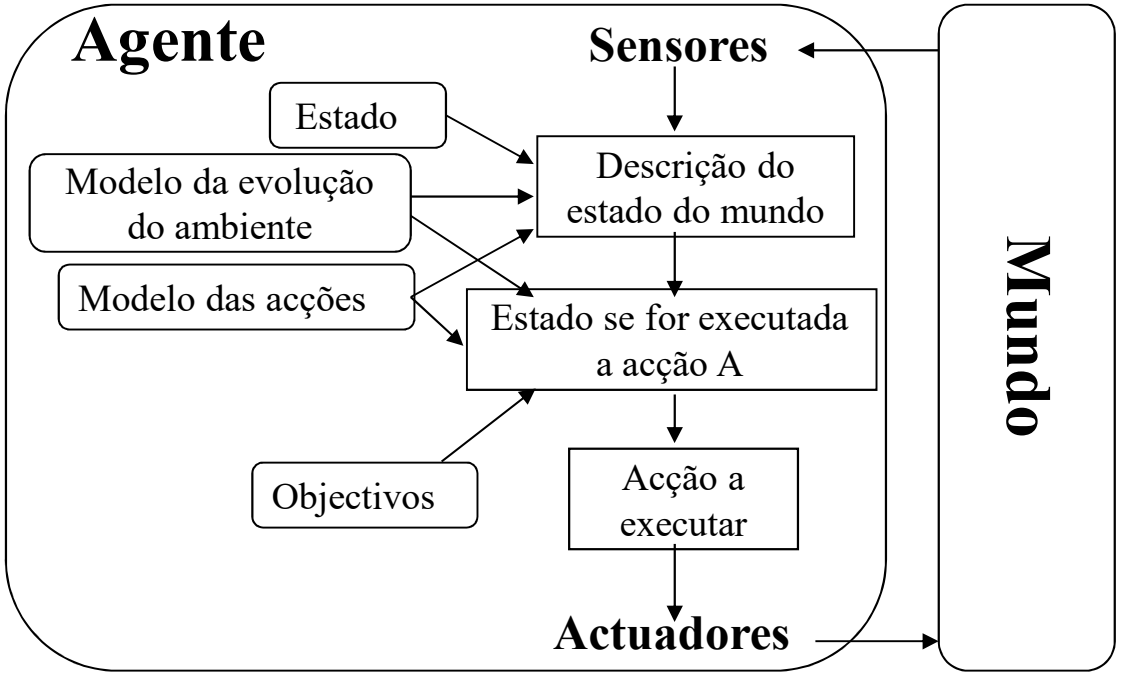


Sistemas de Quadro Preto

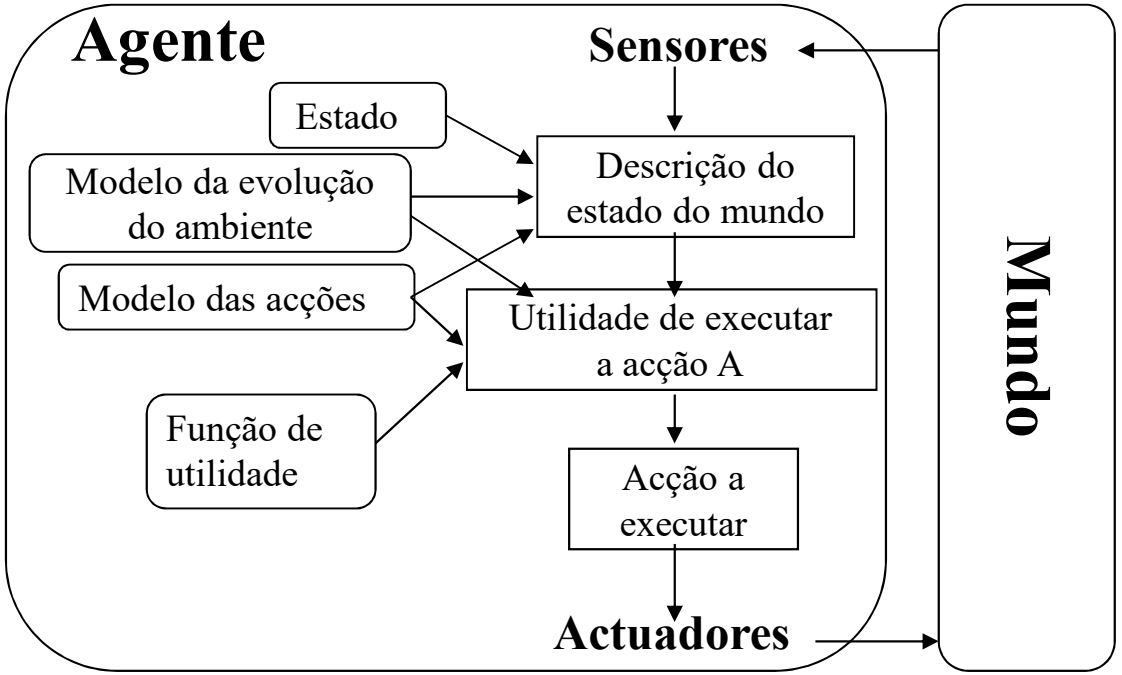


- Podem ser vistos como uma elaboração dos sistemas reactivos com estado interno.
- Uma "fonte de conhecimento" (FC) é um programa que vai fazendo alterações no Quadro Preto.
- Uma FC pode ser vista como um especialista num dado domínio.
- Tipicamente, cada FC rege-se por um conjunto de regras de situação-acção.

Agente deliberativo: orientado por objectivos



Agente deliberativo: orientado por função de utilidade



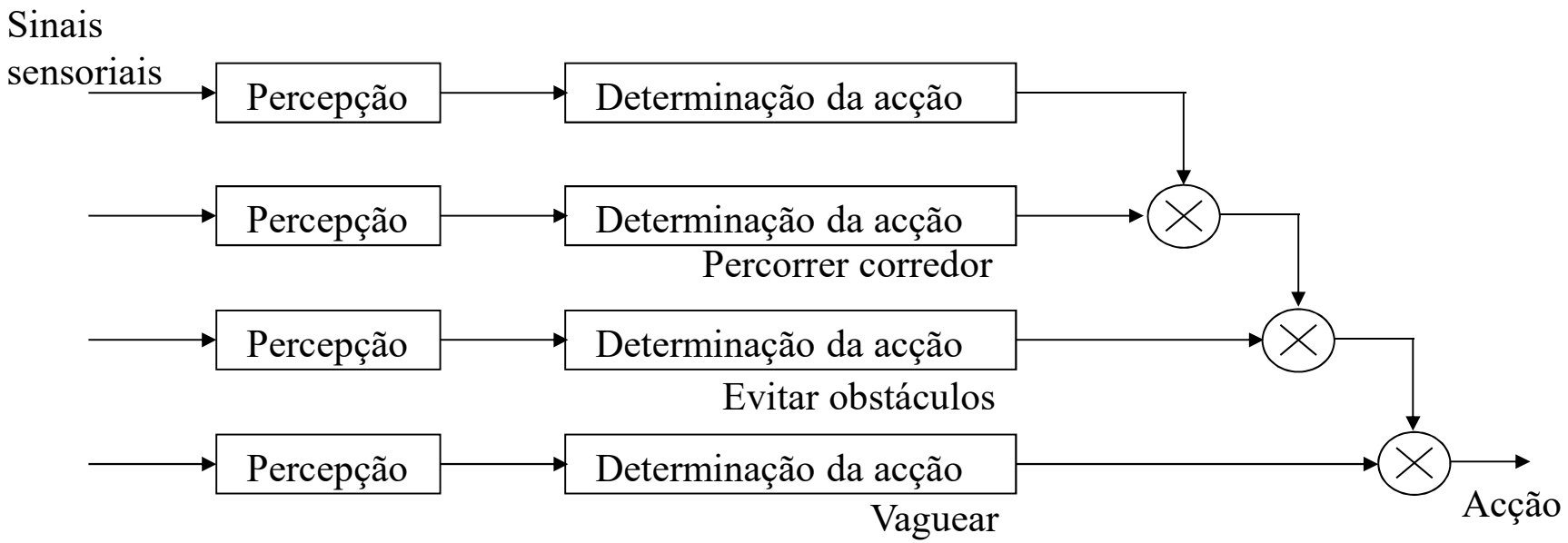
Propriedades do mundo de um agente

- 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 "continuo" quando a evolução do estado do mundo é um processo continuo ou sem saltos; caso contrário o mundo diz-se "discreto".

Mundo de um agente: Exemplos

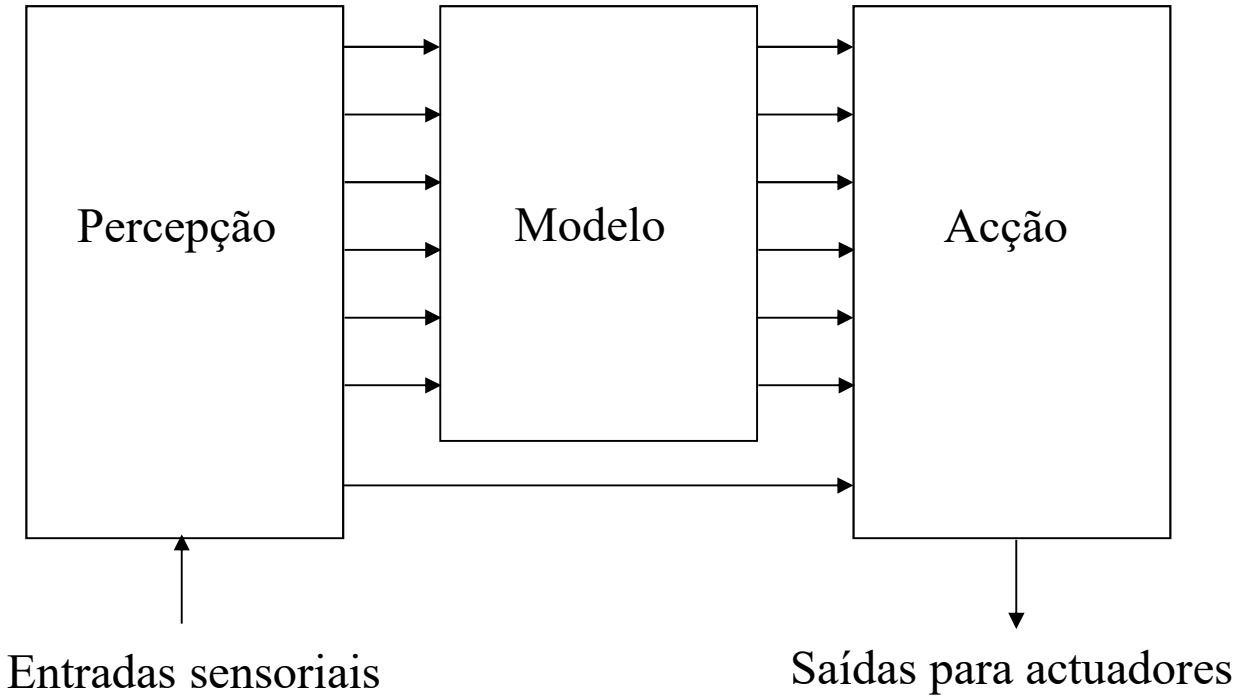
Mundo	Acessível	Determinístico	Episódico	Dinâmico	Continuo
Xadrês s/ relógio	Sim	Sim	Não	Não	Não
Xadrês c/ relógio	Sim	Sim	Não	Semi	Não
Poker	Não	Não	Não	Não	Não
Condução de carro	Não	Não	Não	Sim	Sim
Diagnóstico médico	Não	Não	Não	Não	Sim
Sistema de análise de imagem	Sim	Sim	Sim	Semi	Sim
Manipulação 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

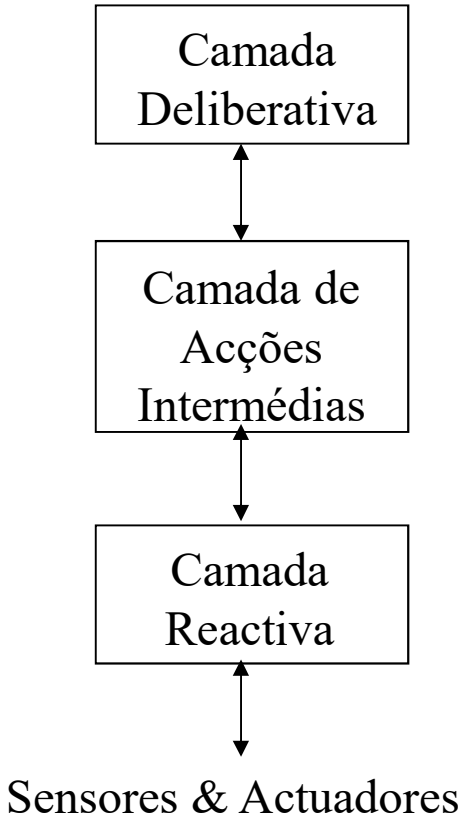


- A arquitectura de subsunção [Brooks,1986; Connell,1990] procura estabelecer a ligação entre percepção e acção a vários níveis – daqui resulta uma organização em camadas.
- A camada mais baixa é a mais reactiva.
- O peso da componente deliberativa aumenta à medida que se sobe na estrutura de camadas.

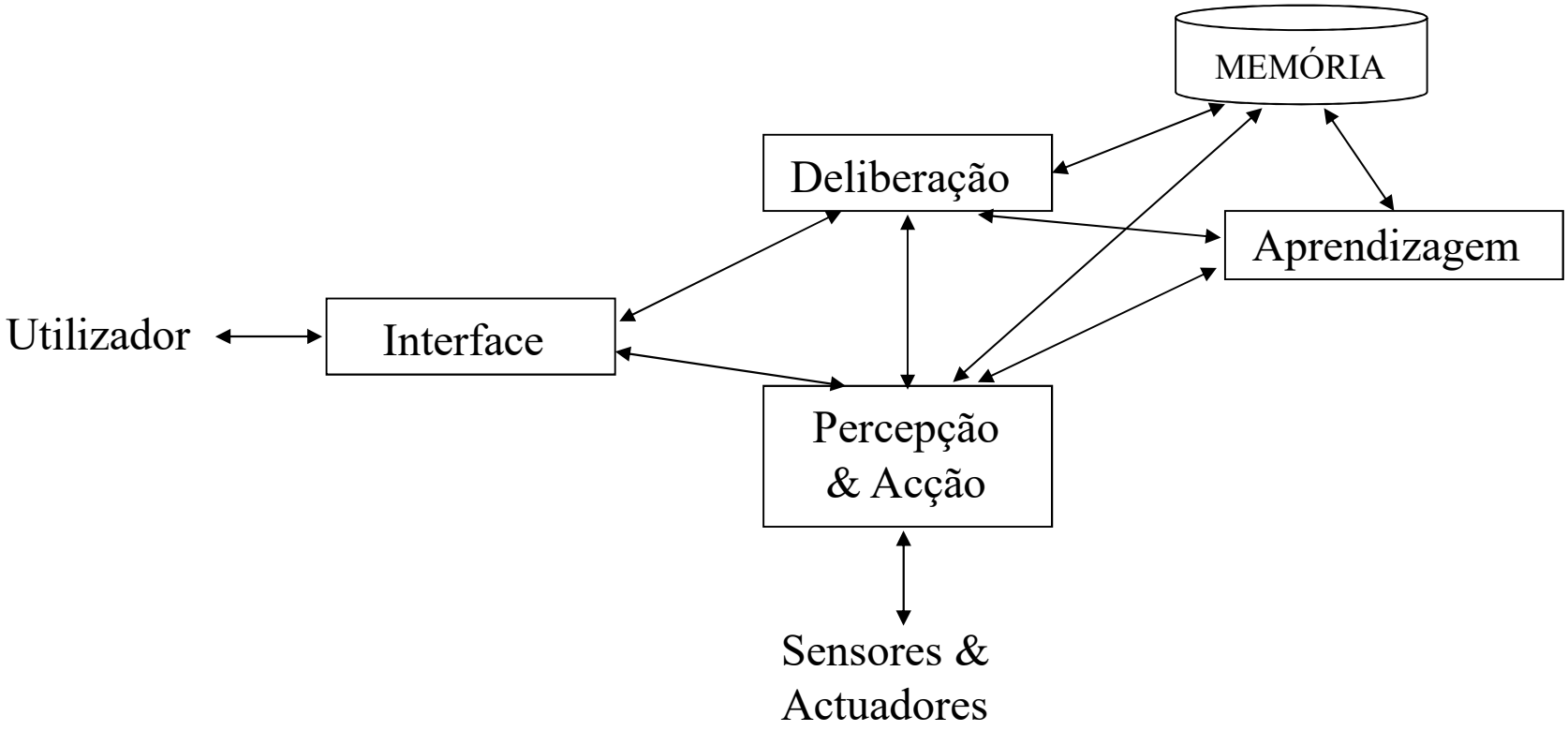
Arquitecturas de Agentes: Três Torres



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 raciocício 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

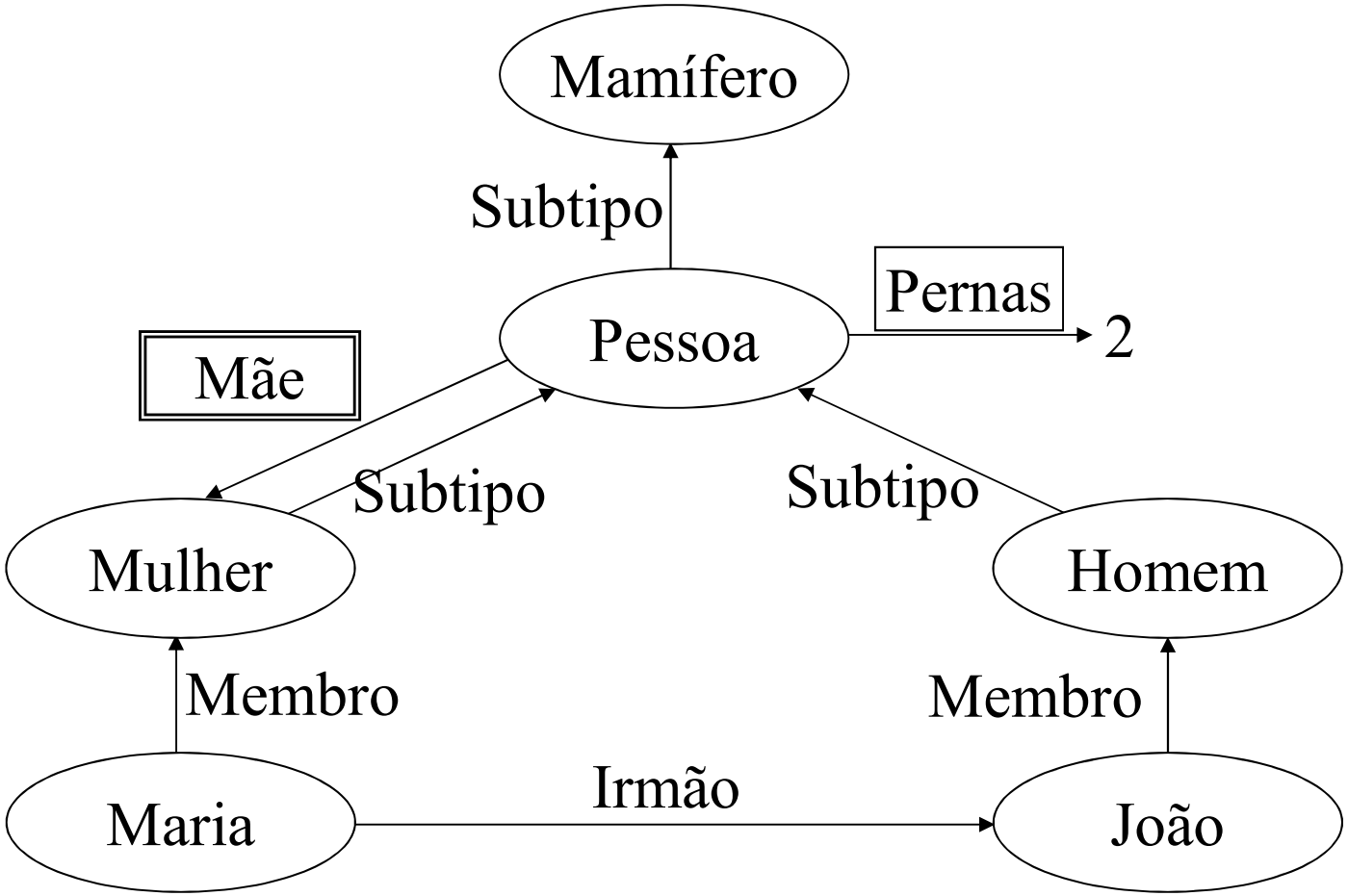
Redes Semânticas

- 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:
  - $\forall x\ x \in A \Rightarrow \exists y\ (y \in B \wedge R(x,y))$

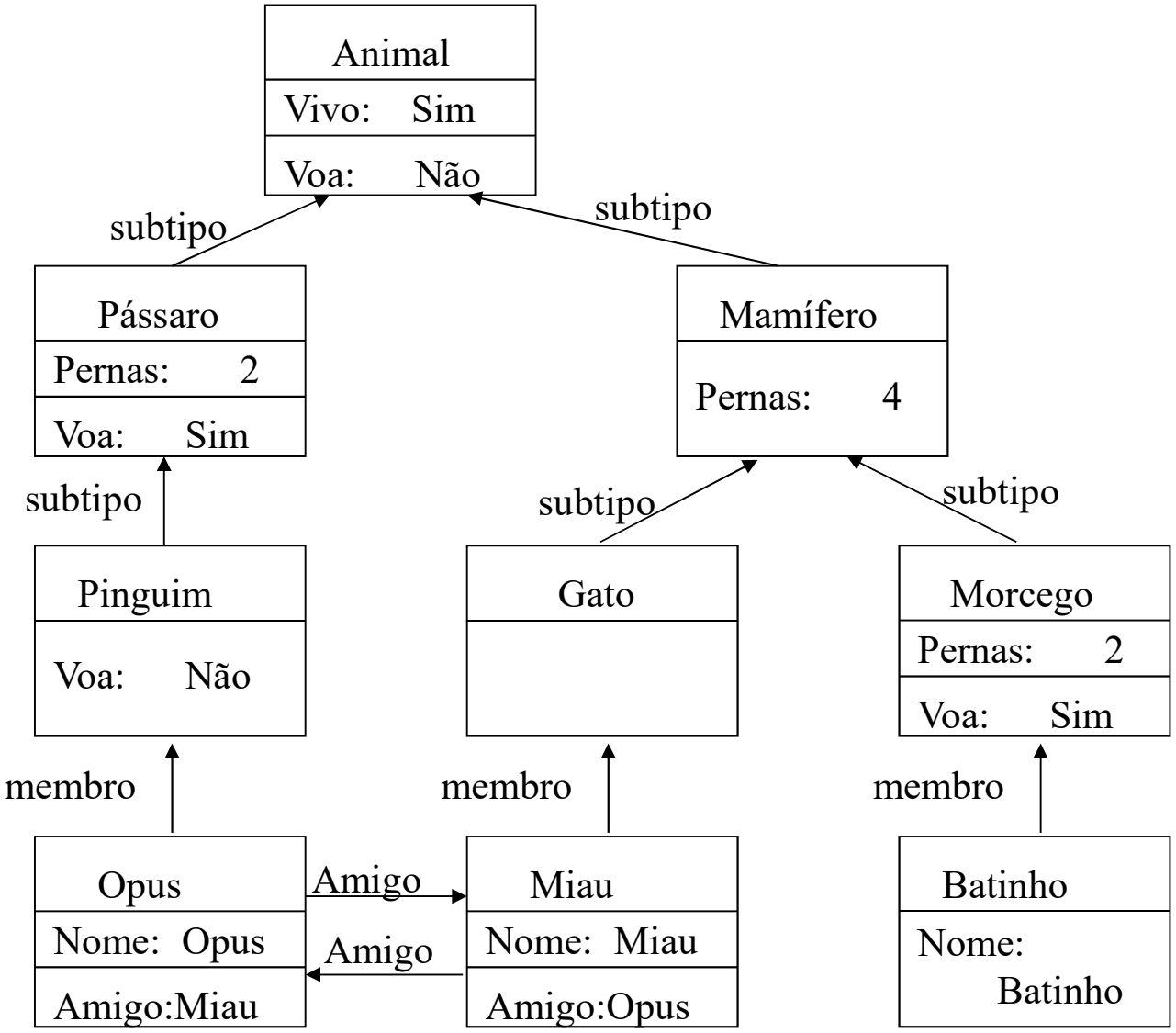
Redes semânticas – exemplo



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 cancelamente 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 automáticamente 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

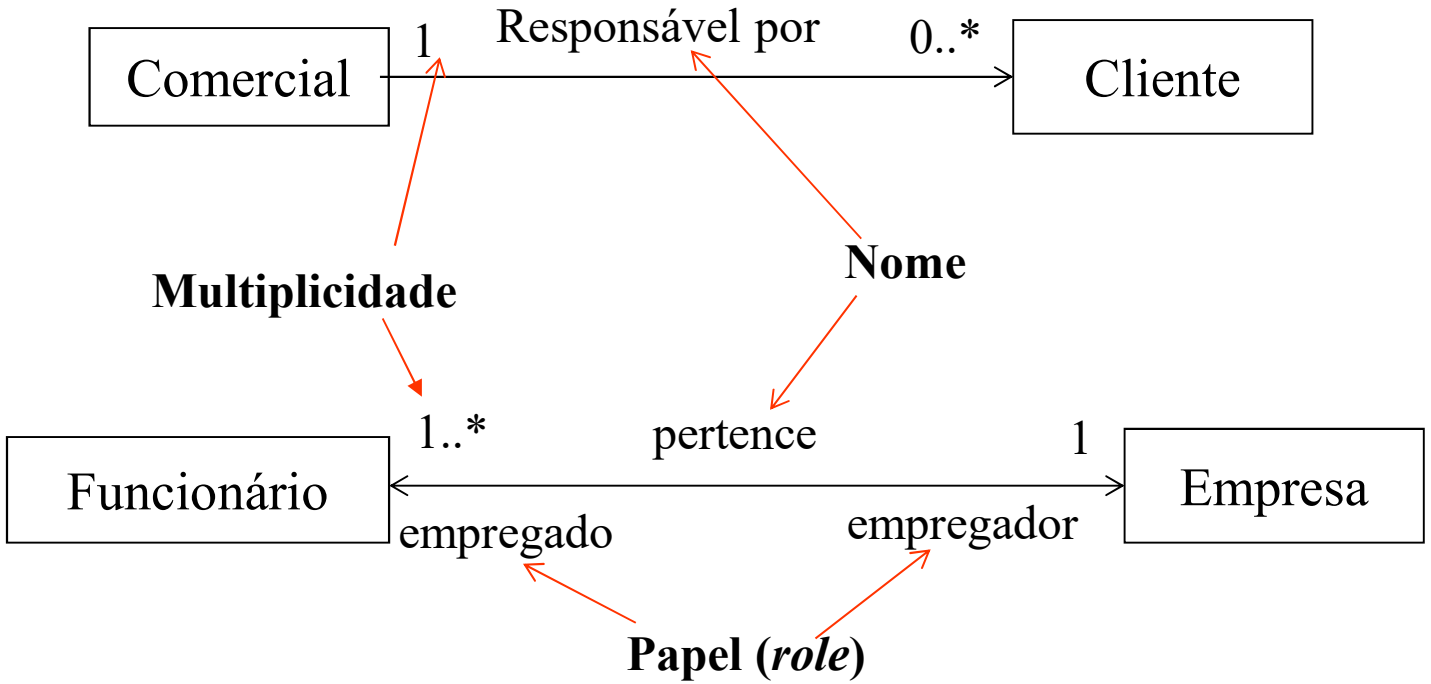
UML / Diagramas de Classes - 1

- 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

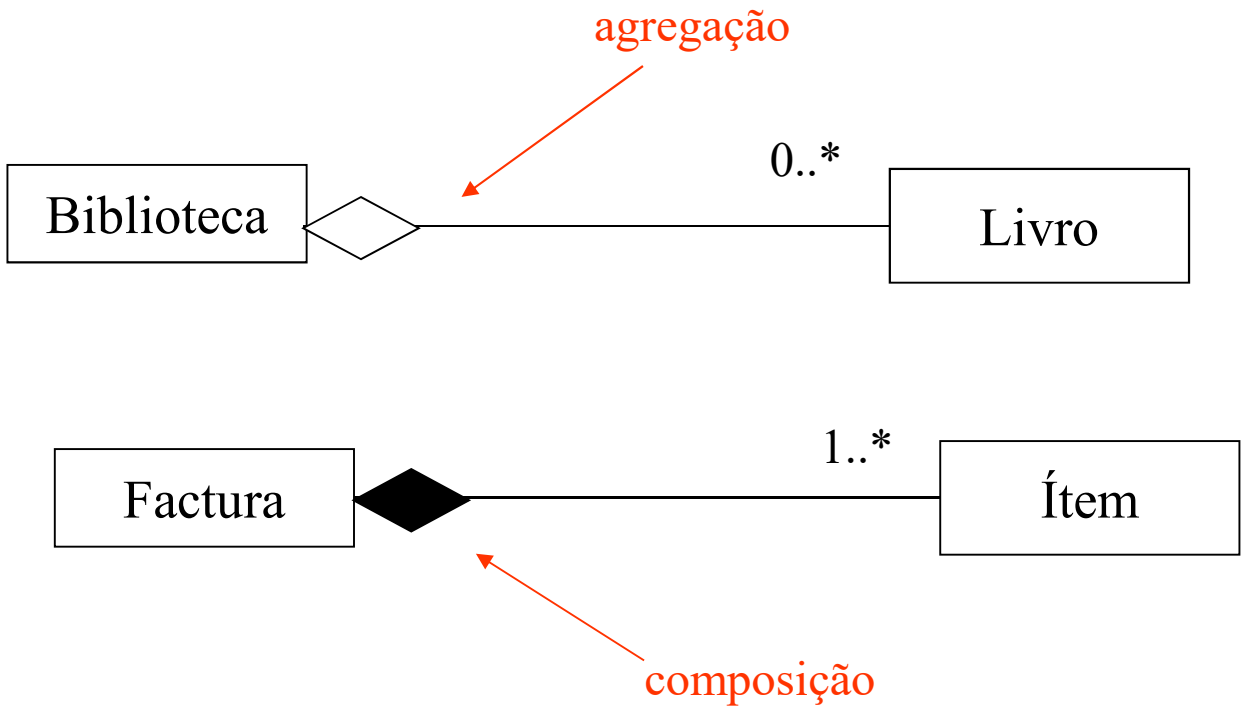
UML / Diagramas de Classes – 2: Relações

- 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)

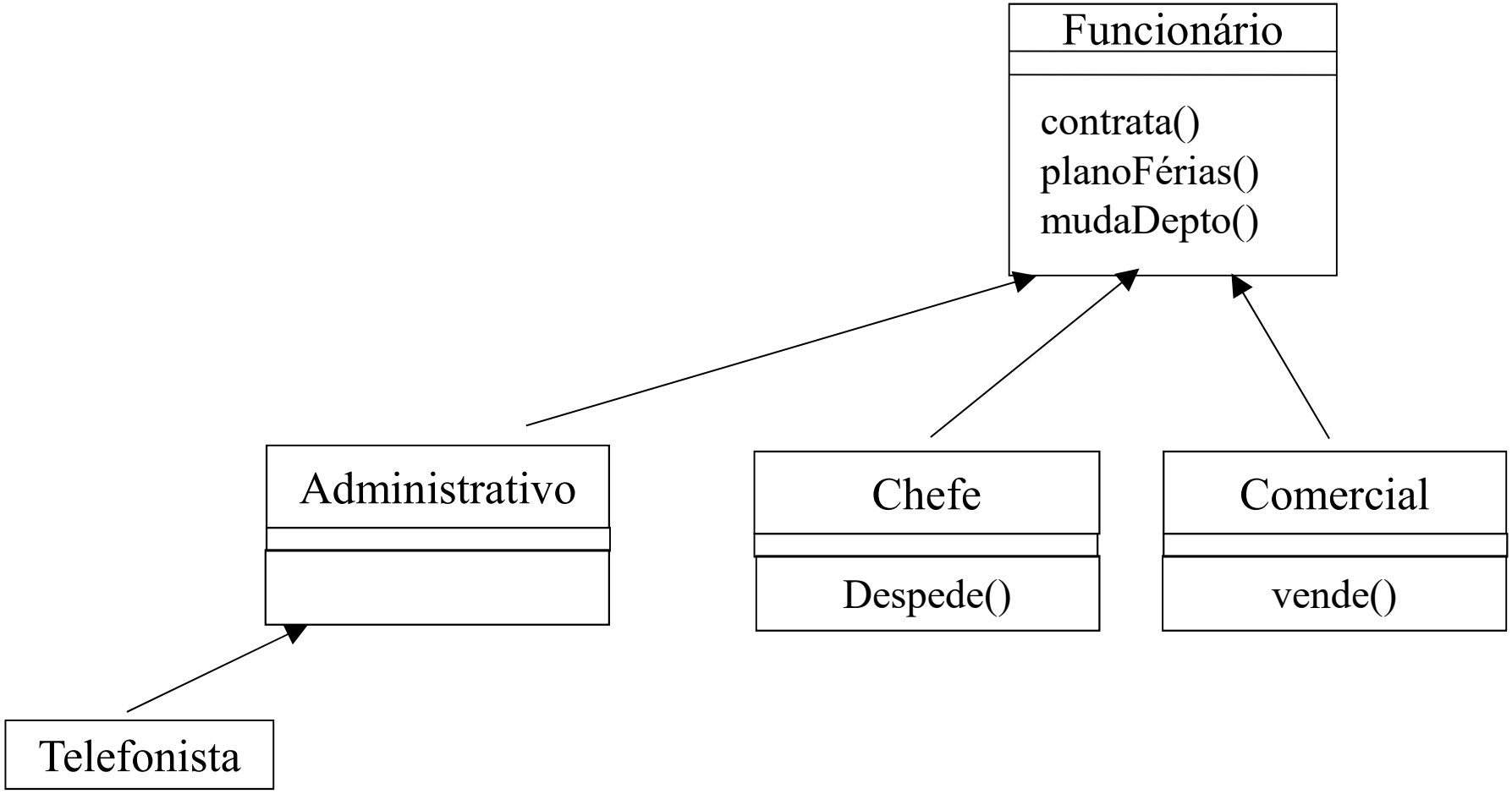
UML / Diagramas de Classes – 3: Associação



UML / Diagramas de Classes – 4: Agregação e Composição



UML / Diagramas de Classes – 5: Generalização



Redes semânticas *versus* UML

Redes semânticas	*UML*
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

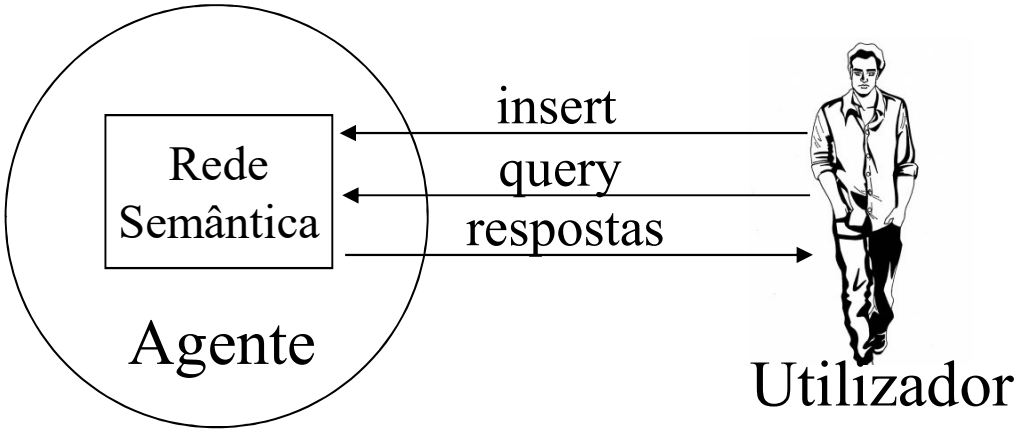
- Dedução – permite inferir casos particulares a partir de regras gerais
  - Preserva a verdade
  - As regras de inferência apresentadas anteriormente são regras dedutivas
- Indução – é o oposto da dedução; permite inferir regras gerais a partir de casos particulares
  - É a base principal da aprendizagem

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

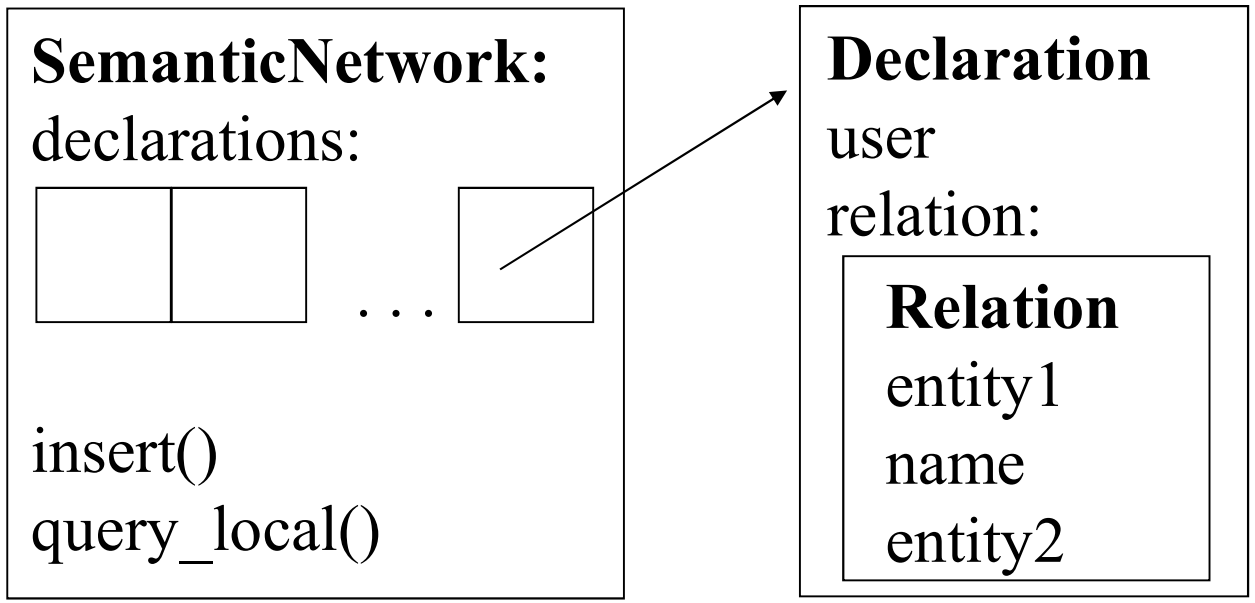
- Vamos criar uma rede semântica, definida como um conjunto de declarações
- Cada declaração associa uma relação semântica ao indivíduo que a declarou
  - Declaration(user,relation)



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 raciocício 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 → *FórmulaAtómica*

| *Fórmula Conectiva Formula*

| *Quantificador Variável*, ... *Fórmula*

| ‘¬’ *Fórmula*

| ‘(’ *Fórmula* ‘)’

FórmulaAtómica → *Predicado* ‘(’ *Termo* ‘,’ ...)’ | *Termo* ‘=’ *Termo*

Termo → *Função* ‘(’ *Termo* ‘,’ ...)’ | *Constante* | *Variável*

Conectiva → ‘⇒’ | ‘∧’ | ‘∨’ | ‘⇔’

Quantificador → ‘∃’ | ‘∀’

Constante → A | X1 | Paula | ...

Variável → a | x | s | ...

Predicado → Portista | Cor | ...

Função → Registo | Mãe | ...

Exemplos

- "Todos em Oxford são espertos":
  - $\forall x$ Estuda(x,Oxford) $\Rightarrow$ Esperto(x)
  - Erro comum: Usar $\wedge$ em vez de $\Rightarrow$

    $\forall$x Estuda(x,Oxford) $\wedge$ Esperto(x)

    Significa "Todos estão em Oxford e todos são espertos"
- "Alguém em Oxford é esperto":
  - $\exists x$ Estuda(x,Oxford) $\wedge$ Esperto(x)
  - Erro comum: Usar $\Rightarrow$ em vez de $\wedge$

    $\exists$x Estuda(x,Oxford) $\Rightarrow$ Esperto(x)

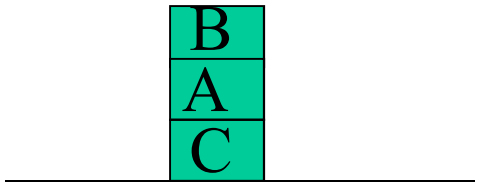
    qualquer estudante de outra universidade forneceria uma interpretação verdadeira.
- "Existe uma pessoa que gosta de toda a gente"
  - $\exists x \; \forall y$ Gosta(x,y)

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 sse 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 *Conjuntive 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 atómica (literal positivo) ou a negação de uma fórmula atómica (literal negativo).
  - Nota: na lógica proposicional uma fórmula atómica 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:
  - Fórumula 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:
$\forall x\ \forall y\ \neg(\ p(x,y) => \forall y\ q(y,y)\ )$

Variáveis renomeadas:
$\forall a\ \forall b\ \neg(\ p(a,b) => \forall c\ q(c,c)\ )$

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) , ¬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 fórnulas em Δ, e escreve-se

    $$\Delta \models A,$$

    se A toma o valor 'verdadeiro' em todas as interpretações para as quais cada uma das fórmulas em Δ toma também o valor verdadeiro.
- Definição de Prova
  - Uma sequência de fórmulas $\{ A_1, A_2, \ldots, A_n \}$ é uma prova (ou dedução) de A_n a partir de um conjunto de fórmulas Δ sse cada uma das fórmulas A_i está em Δ ou pode ser inferida a partir das fórmulas A_1 .. 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 $\{ A_1, A_2, \ldots, A_n \} \models B$, então $A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B$, e vice-versa.
- Redução ao absurdo:
  - Se o conjunto de fórmulas Δ 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 necessáriamente verdadeiras)

$$\{ A \vee B, \neg B \vee C \} \mid\!- A \vee C$$

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

$$\{ A \wedge B \} \mid\!= A \vee B$$

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* $s = \{ t_1/x_1, \ldots, t_n/x_n \}$ a uma fórmula W denota-se $SUBST(W,s)$ ou Ws; Significa que todas as ocorrências das variáveis $x_1, \ldots, x_n$ em W são substituidas pelos termos $t_1, \ldots, t_n$
- 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:

  $\{ A \vee B, \neg C \vee D \} \text{ |- } SUBST(A \vee D, g)$

  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 Claúsulas 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 $\quad \neg A \vee B$
    $\neg A \vee B \vee \neg C$ $\quad \neg A \vee \neg B$
- 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 raciocício 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 … 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: construida 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) ?1) ?x)
- Definição de relações (=predicados)
  - (defrelation *r* (*v1* .. *vn*) := *p*)
  - etc.
  - Exemplos:

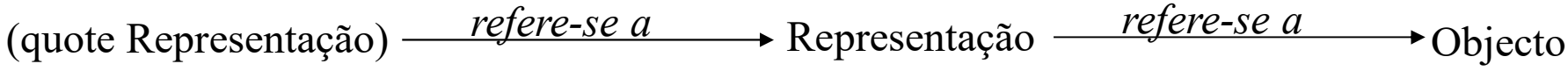
```
(defrelation null (?1) := (= ?1 (listof)))
(defrelation list (?x) :=
        (exists (@1) (= ?x (listof (@1)))))
```

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 - atómica, 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 functorese as relações são constantes), *ordem superior* (os functores 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 raciocício 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 *aquisição 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 conhciemento 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

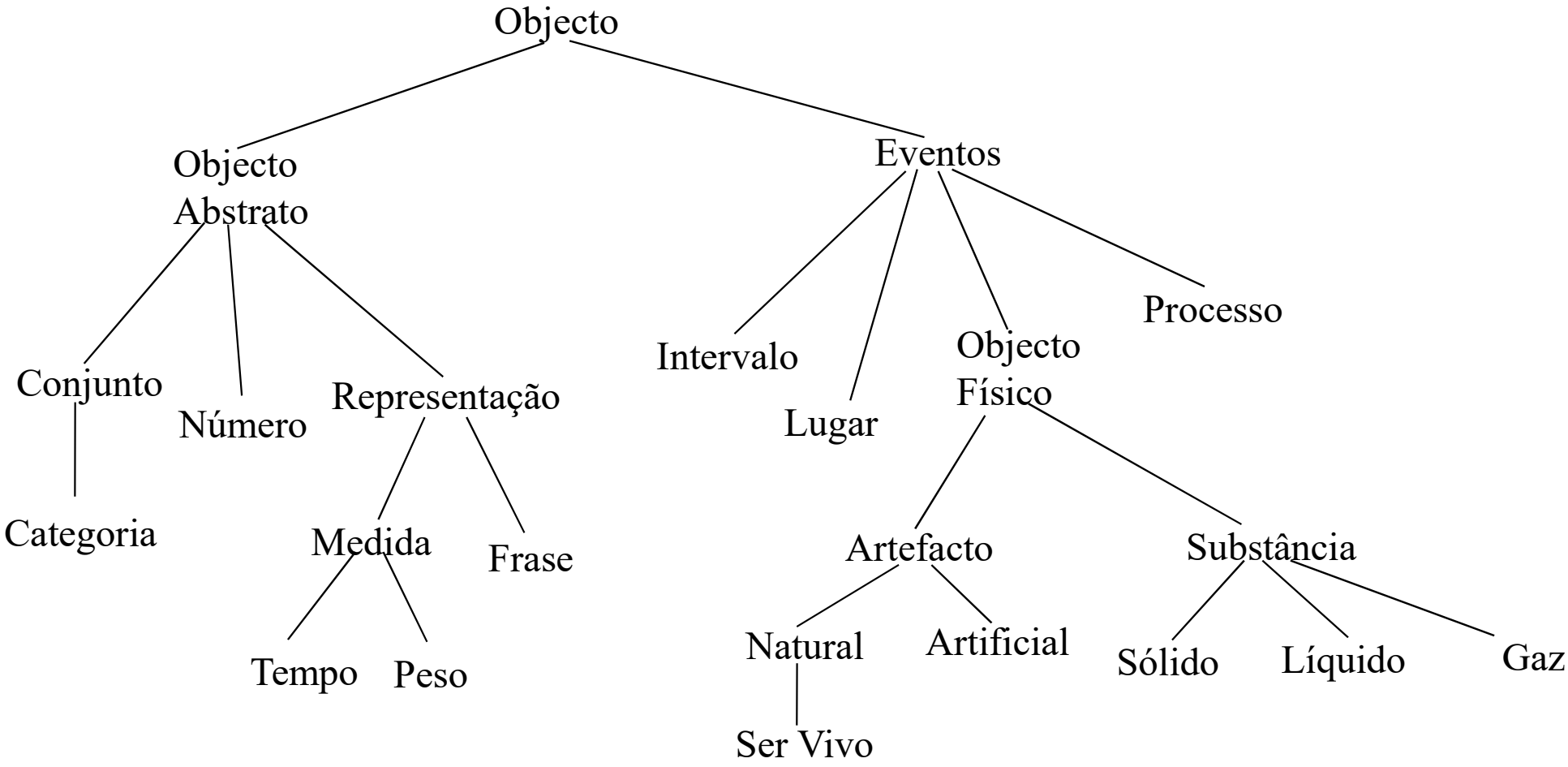
Ontologias

- 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 raciocício 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

$$P(true) = 1$$

- Proposições necessariamente falsas têm probabilidade 0

$$P(false) = 0$$

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

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

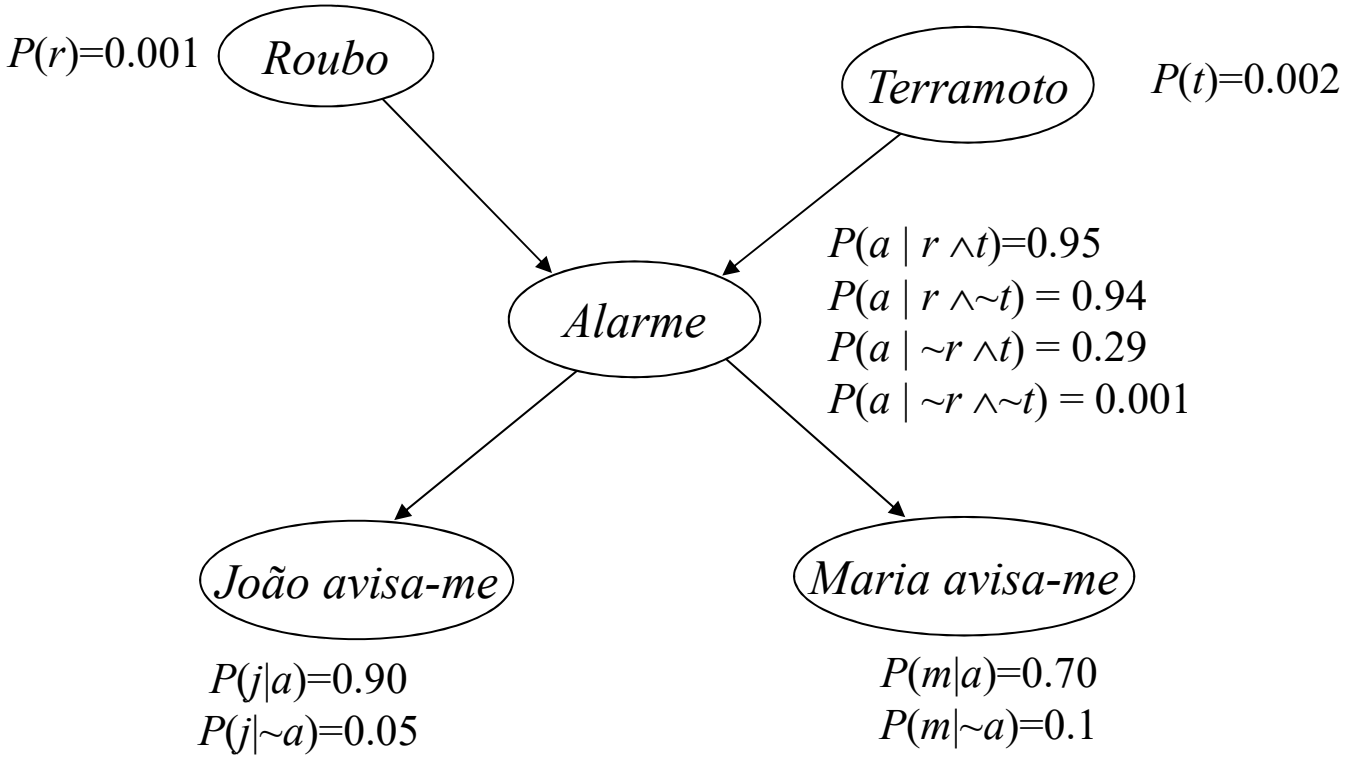
Probabilidades condicionadas

- 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:

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

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:

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

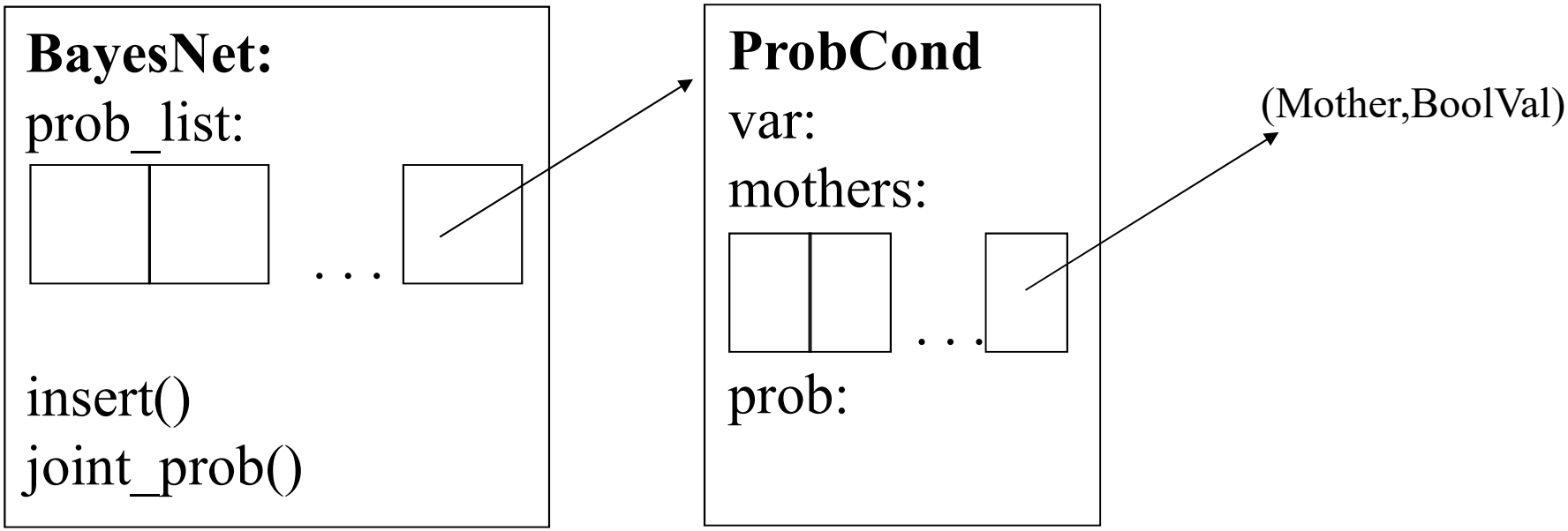
- Assim, no exemplo anterior, a probabilidade de o alarme tocar e o João e a Maria ambos avisarem num cenário em que não há roubo nem terramoto, é dada por:

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

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

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

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

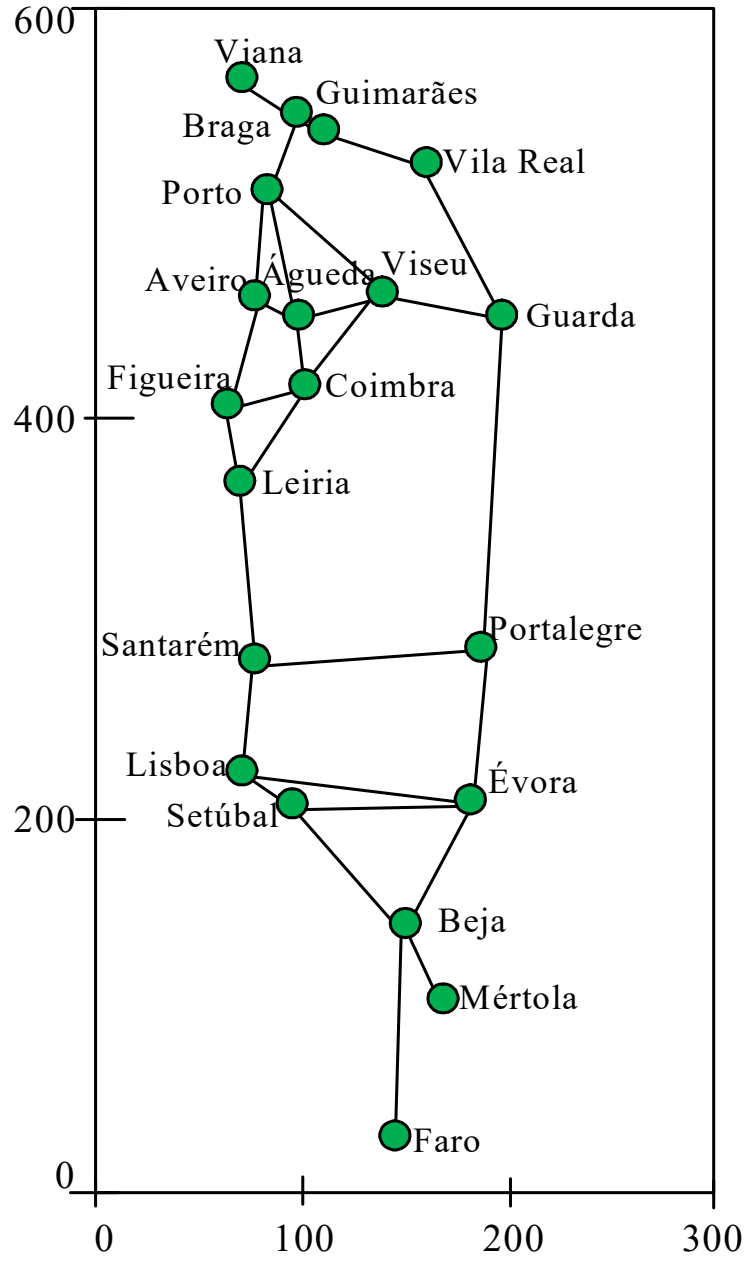
- Um *problema* é algo cuja solução não é imediata
- Exemplos de problemas:
  - Dado um conjunto de axiomas, demonstrar um novo teorema
  - Dado um mapa, determinar o melhor caminho entre dois pontos.
  - Dada uma situação num jogo de xadrez, determinar uma boa jogada.
  - Determinar a melhor distribuição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

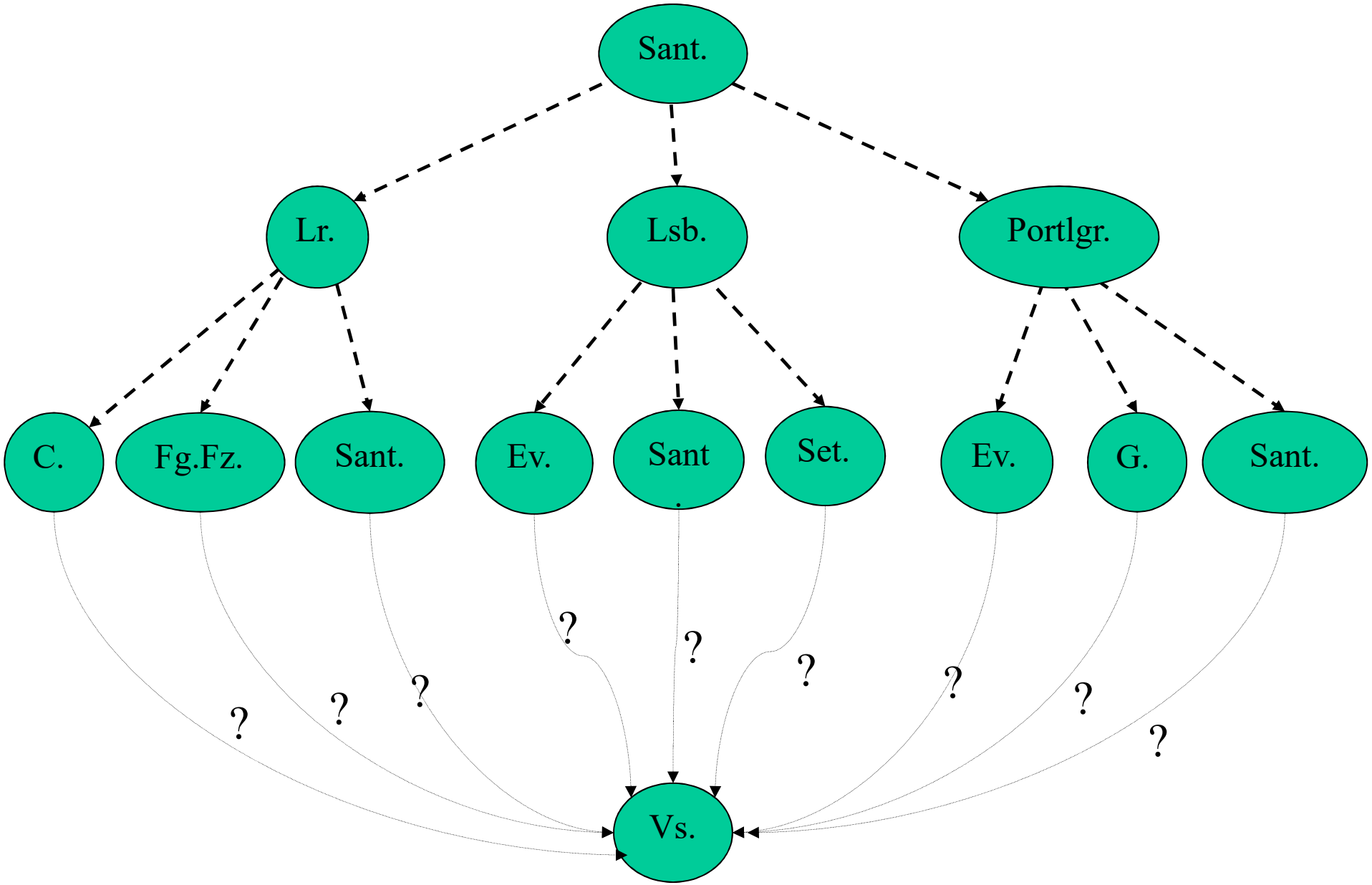
Formulação de problemas e pesquisa de soluções

- 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




Sant.
Lr.
Lsb.
Portlgr.
C.
Fg.Fz.
Sant.
Ev.
Sant.
Set.
Ev.
G.
Sant.
?
?
?
?
?
?
?
?
?
?
?
?
Vs.

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 ← árvore de pesquisa inicializada com o estado inicial do Problema

Ciclo:

se não há candidatos para expansão, **retornar** 'falhou'

Folha ← uma folha escolhida de acordo com Estratégia

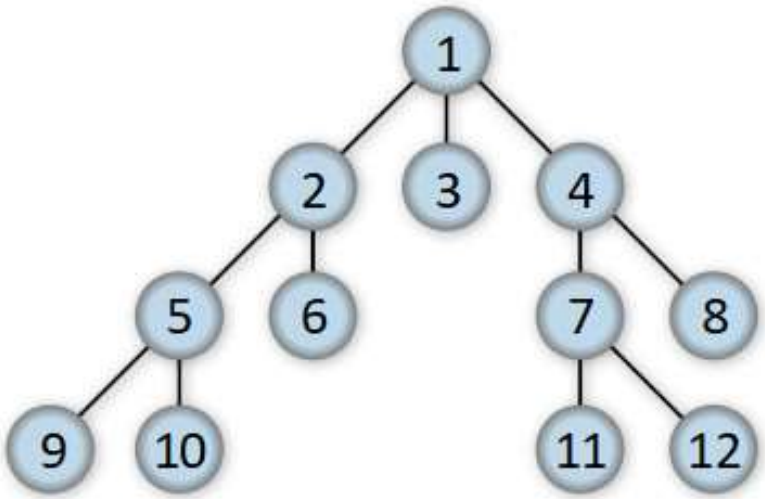
se Folha contem um estado que satisfaz o objectivo

então retornar a Solução correspondente

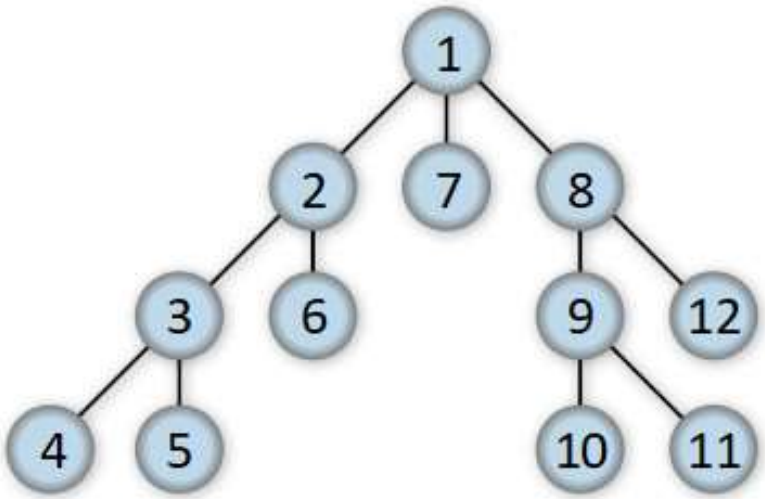
senão expandir Folha e adicionar os nós resultantes à Árvore

Fim do ciclo;

Percursos na árvore de pesquisa



Pesquisa em largura



Pesquisa em profundidade

(crédito das figuras: Alexander Drichel / Wikipedia)

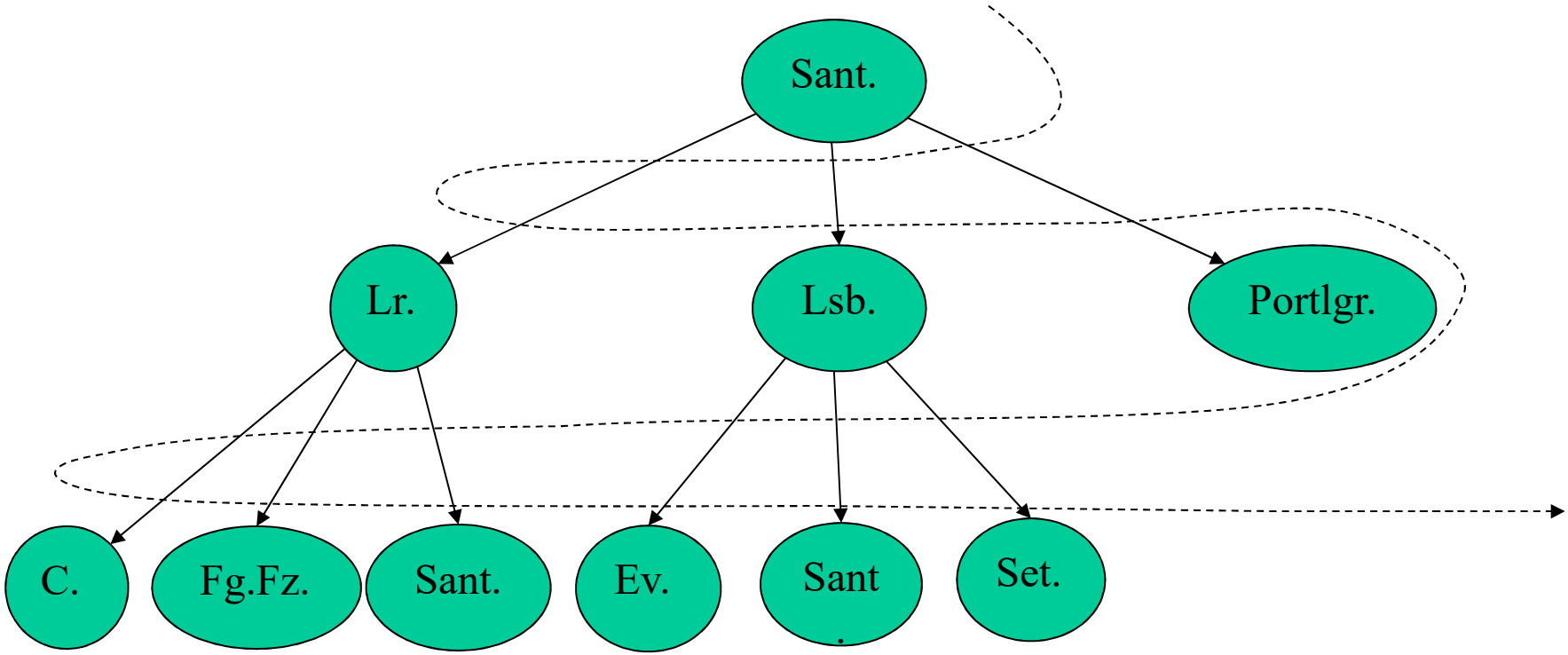
Pesquisa em árvore – implementação baseada numa fila

```
pesquisa_em_arvore(Problema,AdicionarFila) retorna a Solução, ou 'falhou'
    Fila ← [ fazer_nó(estado inicial do Problema) ]
    Ciclo
        se Fila está vazia, retornar 'falhou'
        Nó ← remover_cabeça(Fila)
        se estado(Nó) satisfaz o objectivo
            então retornar a solução(Nó)
            senão Fila ← AdicionarFila(Fila, expansão(Nó))
```

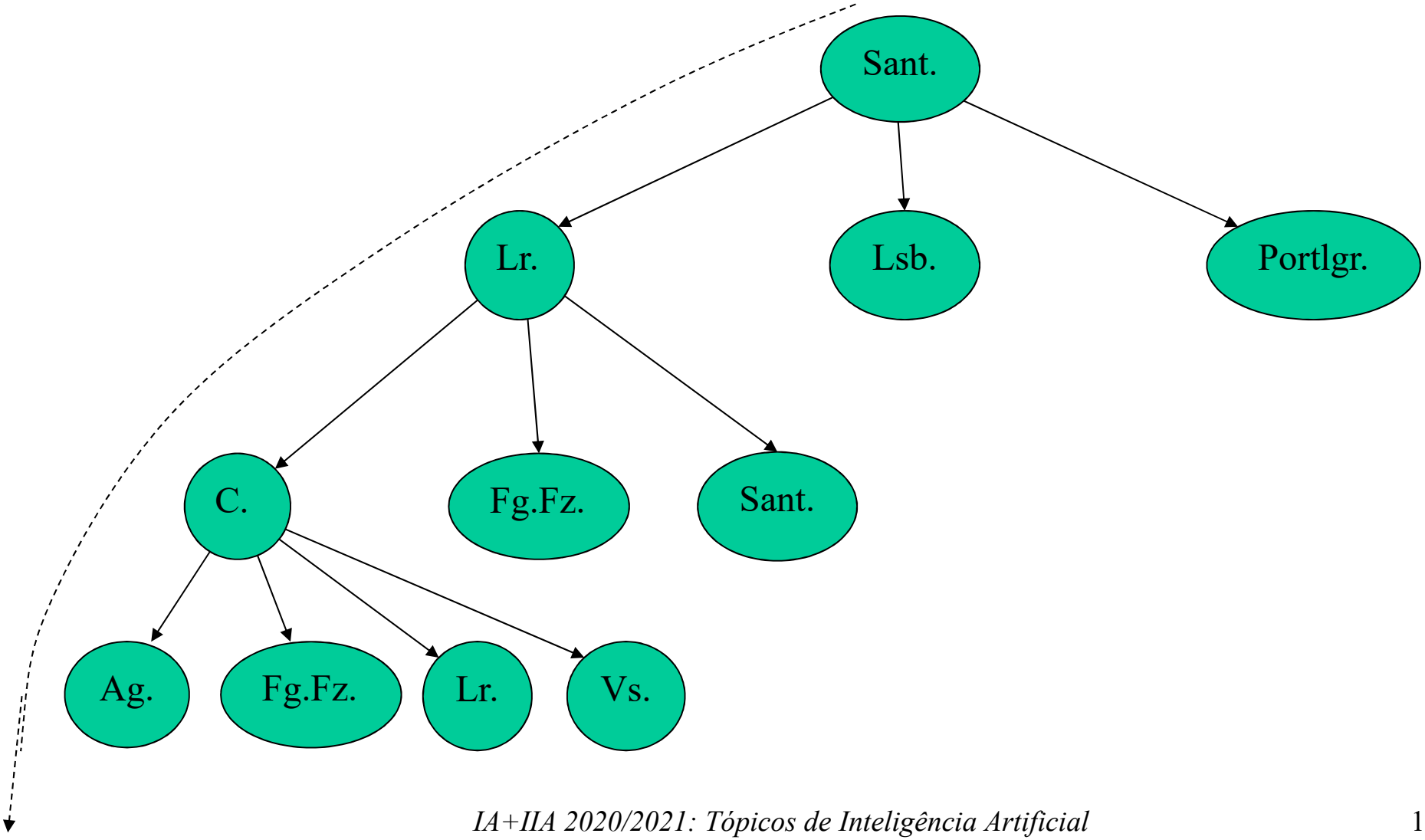
```
pesquisa_em_largura(Problema) retorna a Solução, ou 'falhou'
retornar pesquisa_em_arvore(Problema,juntar_no_fim)
```

```
pesquisa_em_profundidade(Problema) retorna a Solução, ou 'falhou'
retornar pesquisa_em_arvore(Problema,juntar_à_cabeça)
```

Pesquisa em largura



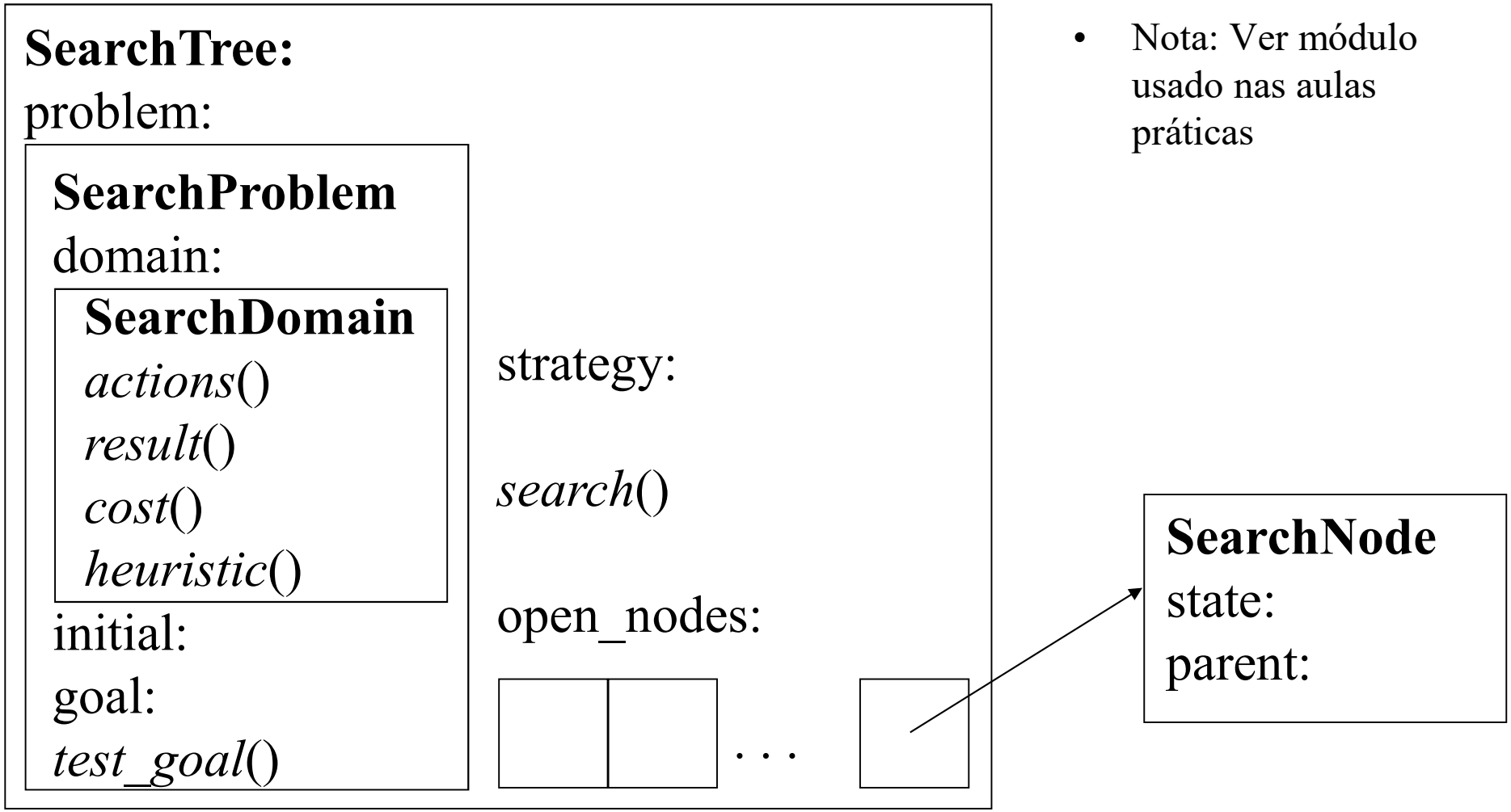
Pesquisa em profundidade



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



- 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 ← estratégia de gestão de fila de acordo com FuncAval
pesquisa_em_arvore(Problema,Estratégia)

Avaliação das estratégias de pesquisa

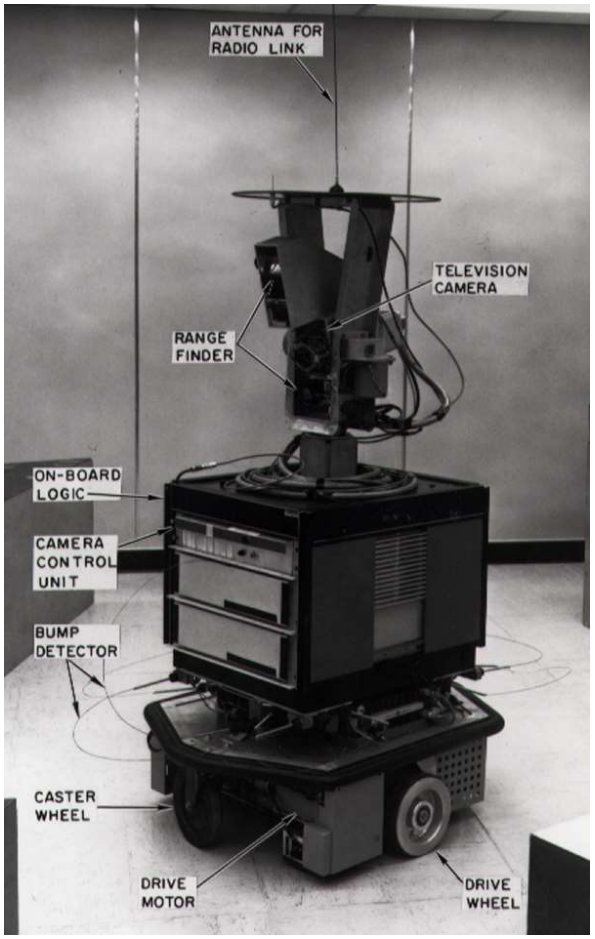
- Completude – uma estratégia é completa se é capaz de encontrar uma solução quando existe uma solução
- Complexidade temporal – quanto tempo demora a encontrar a solução
- Complexidade espacial – quanto espaço de memória é necessário para encontrar uma solução
- Optimalidade – a estratégia de pesquisa 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 optimizar 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 óptima
- *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 óptima

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* ← nó do estado inicial; *ABERTOS* ← { *N0* }
  - *FECHADOS* ← { }
- 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* ← 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

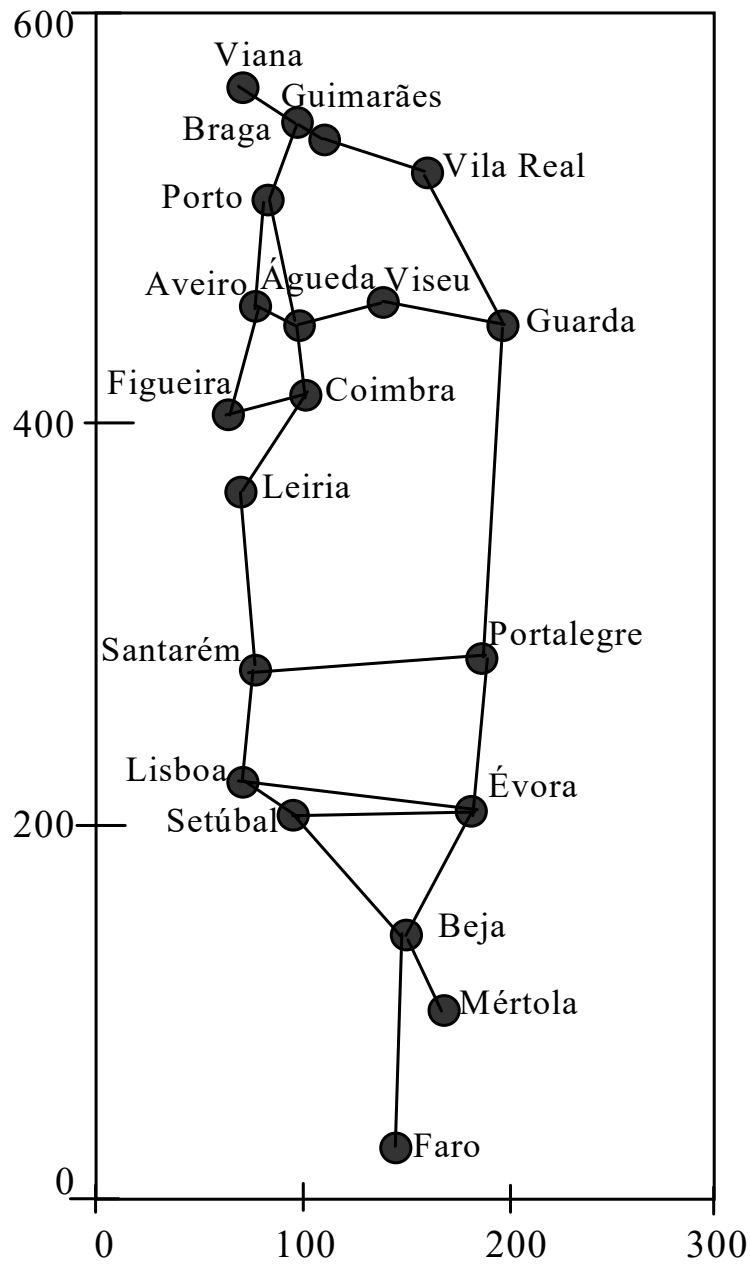
- Seja:
  - N – número de nós da árvore de pesquisa no momento em que se encontra a solução
  - X – Número de nós expandidos (não terminais)
  - d – comprimento do caminho na árvore correspondente à solução
- *Ramificação média* – número médio de filhos por nó expandido:

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

  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: $1 + B + B^2 + ... + B^d = N$ ou seja: $\frac{B^{d+1} - 1}{B - 1} = N$ (resolve-se por métodos numéricos).
  - O factor de ramificação efectiva é um indicador da eficiência da técnica de pesquisa utilizada.

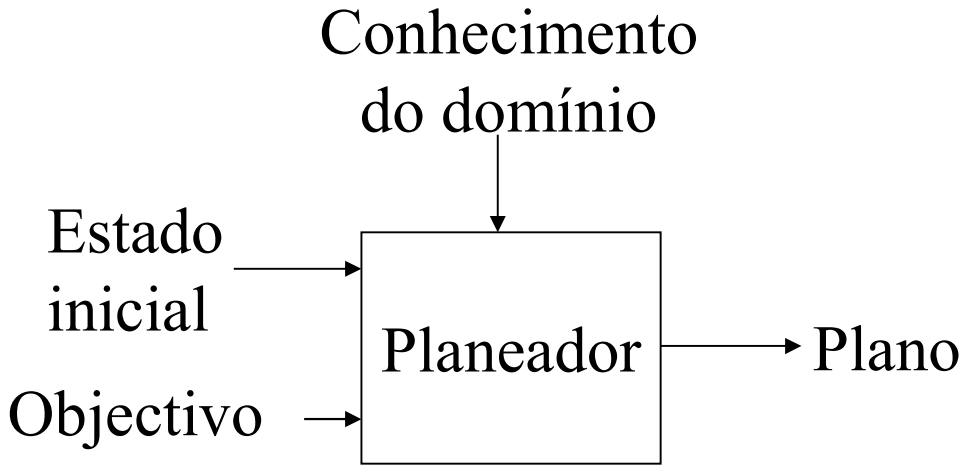
Aplicação: planear um passeio turístico

- 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.

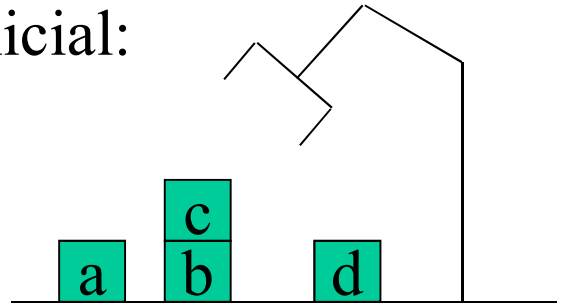


Representação de acções em problemas de planeamento

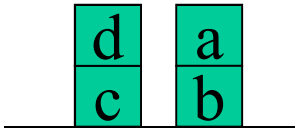
- STRIPS – planeador desenvolvido por volta de 1970, por Fikes, Hart e Nilsson
- A funcionalidade de um dado tipo de operação é definida, no formalismo STRIPS, através de uma estrutura chamada *operador*, que inclui a seguinte informação:
  - *Pré-condições* - um conjunto de fórmulas atómicas que representam 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, …, h_n), podemos combiná-las numa nova heurística:
  - $H(n) = \max(\{h_1(n), \ldots, h_n(n)\})$
- 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

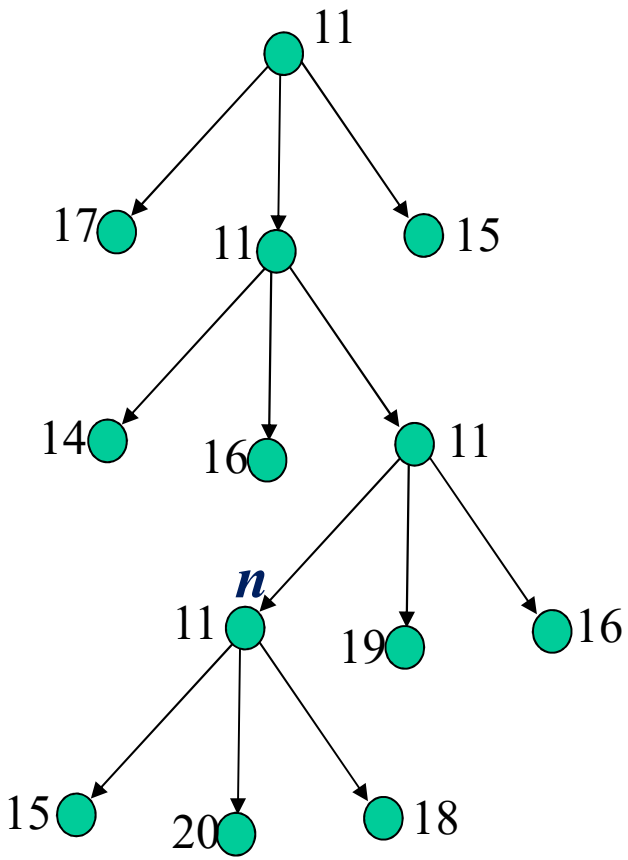
IDA*

- 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

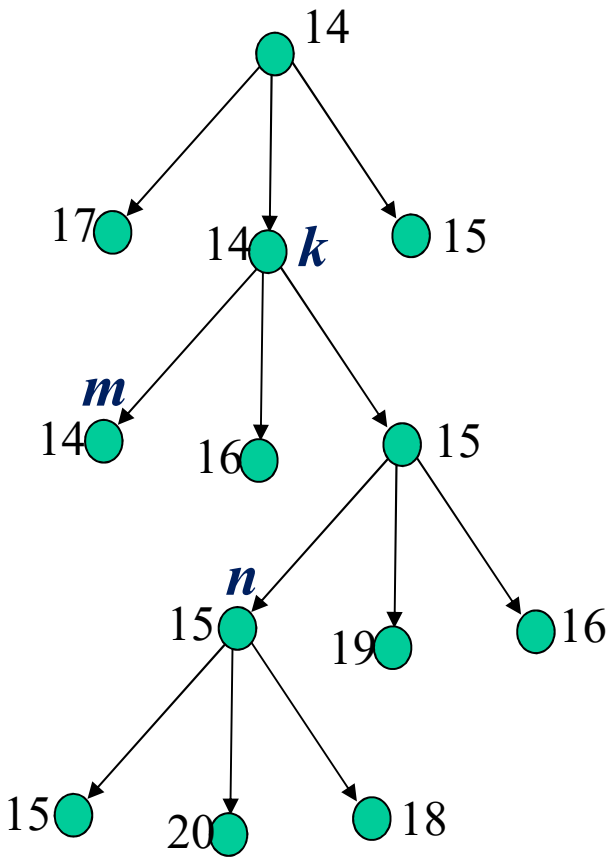
RBFS

- 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*

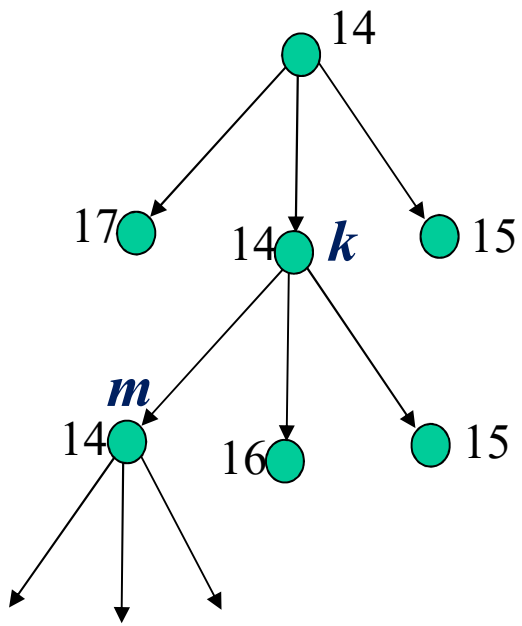
RBFS – exemplo



Nó *n* acaba de ser expandido



Custos foram actualizados

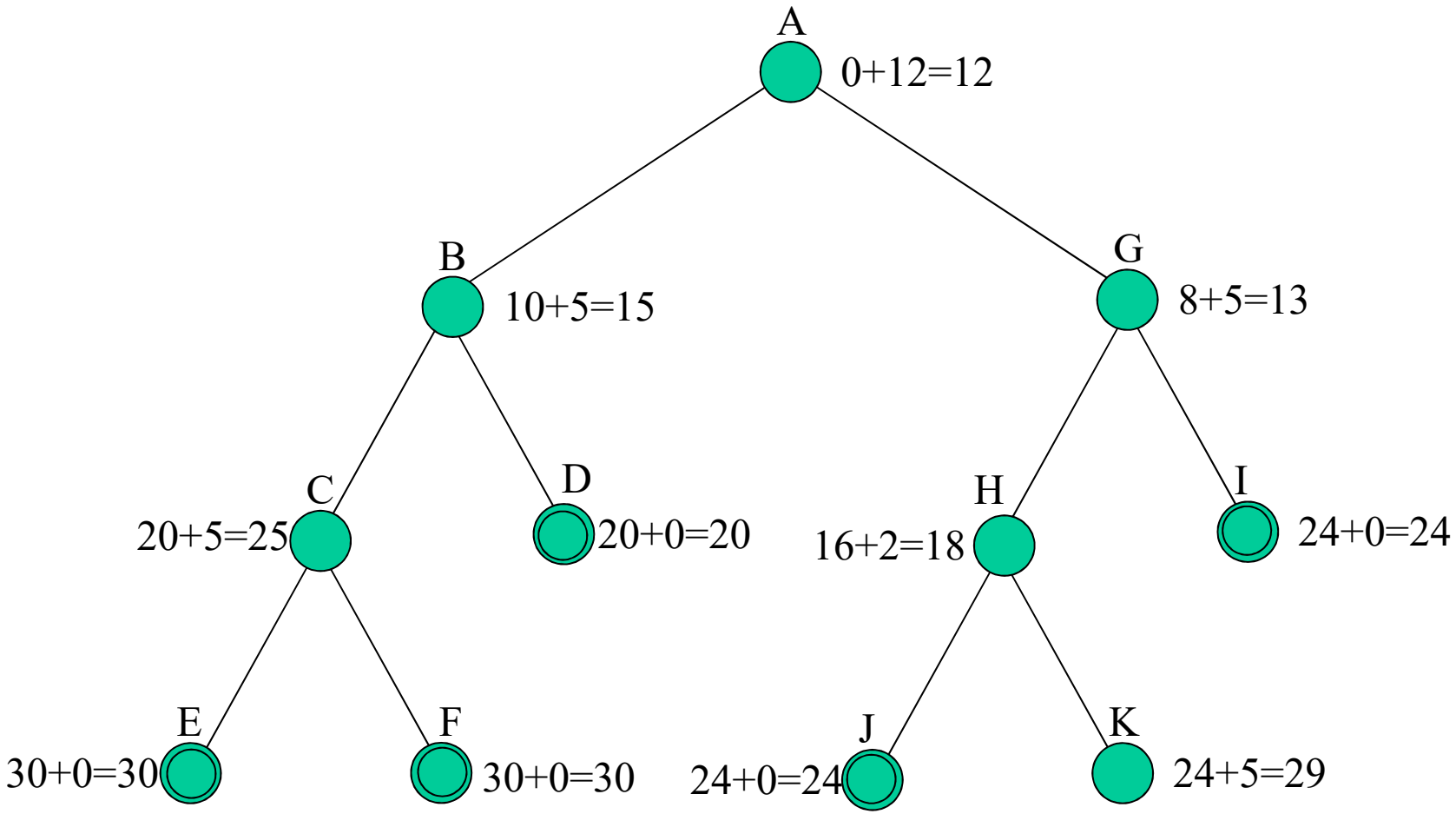


Algoritmo retrocedeu até ao nó *k*;
Expansão segue pelo nó *m*

SMA*

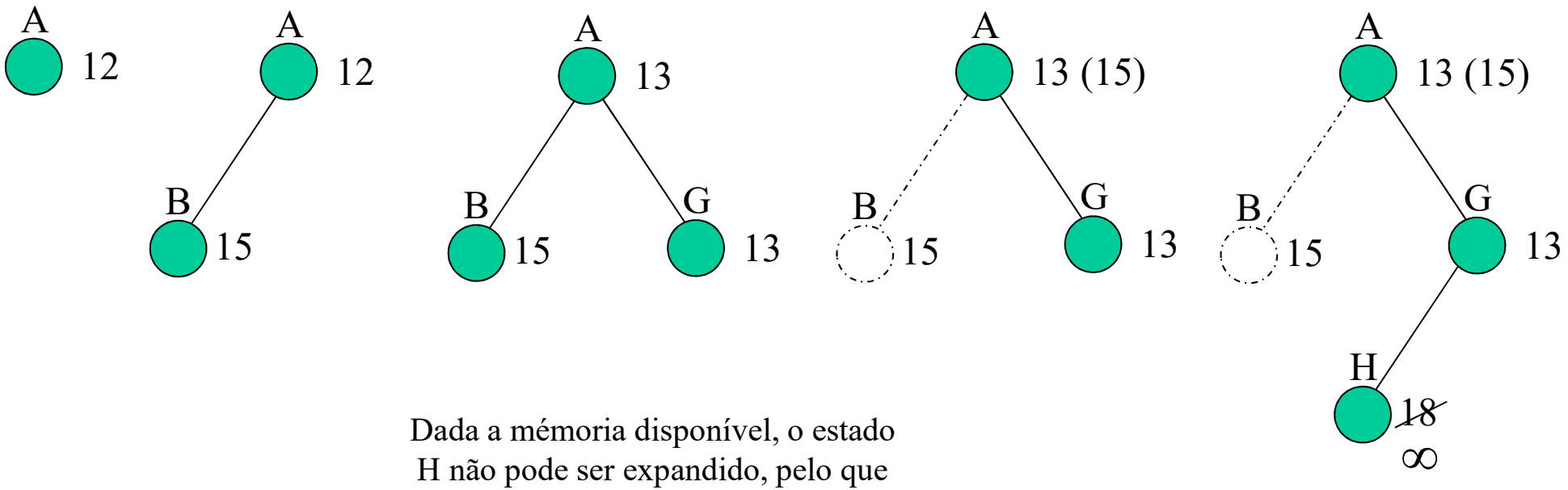
- A* com memória limitada simplificado (*simplified memory-bounded A**)
- Usa a memória disponível
  - Contraste com IDA* e RBFS: estes foram desenhados para poupar memória, independentemente de ela existir de sobra ou não
- Quando a memória chega ao limite, esquece (remove) o nó n com maior custo $f(n)=g(n)+h(n)$, 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



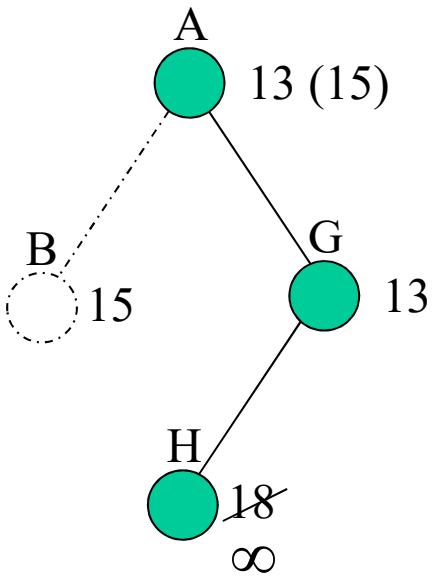
SMA* - exemplo

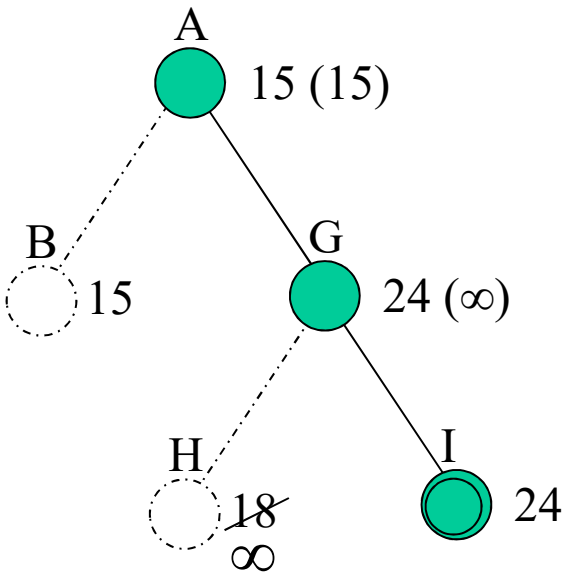
- Neste exemplo: memória = 3 nós
  - Melhores custos de nós esquecidos anotados entre parêntesis



Dada a mémoria 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) ← 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

```
  T W O
+ T W O
-------
F O U R
```

- 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 $2 \cdot O = R + 10 \cdot X_1$ pode ser transformada no seguinte conjunto de restrições:
  - Restrições binárias:
    - O = primeiro(*Aux*)
    - R = segundo(*Aux*)
    - X_1 = terceiro(*Aux*)
  - Restrição unária:
    - 2· primeiro(*Aux*) = segundo(*Aux*) + 10· terceiro(*Aux*)
- *Aux* é uma variável auxiliar cujo domínio é o produto cartesiano dos domínios de O, R e X_1.
  - Ou seja: $Aux \in \{ 0 .. 9 \} \times \{ 0 .. 9 \} \times \{ 0, 1 \}$
  - 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
- Planeamento

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 optimizar 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

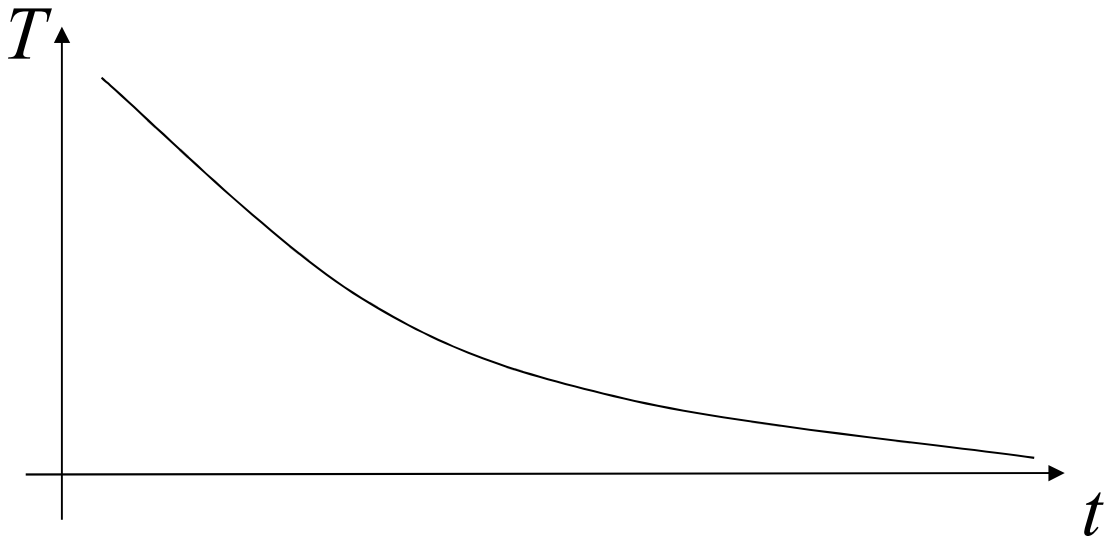
- *Recozimento simulado* (*Simulated Annealing*) é uma variante da pesquisa por montanhismo na qual podem ser aceites refinamentos que, localmente, piorem a solução.
  - O nome inspira-se no processo industrial chamado *recozimento*.
    - Recozer = "deixar esfriar lentamente (um produto de cerâmica 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 .. ∞: {
    T ← *Regime_termico*(*t*)
    se *T*=0, **retornar** a solução de *Nó*
    Prox ← um sucessor de Nó gerado aleatoriamente
    Ganho ← *Aval*(*Prox*)-*Aval*(*Nó*)
    se *Ganho*>0, *Nó* ← *Prox*
    senão, com probabilidade exp(*Ganho*/*T*), **fazer**: *Nó* ← *Prox*
}

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

Recozimento simulado: regime térmico

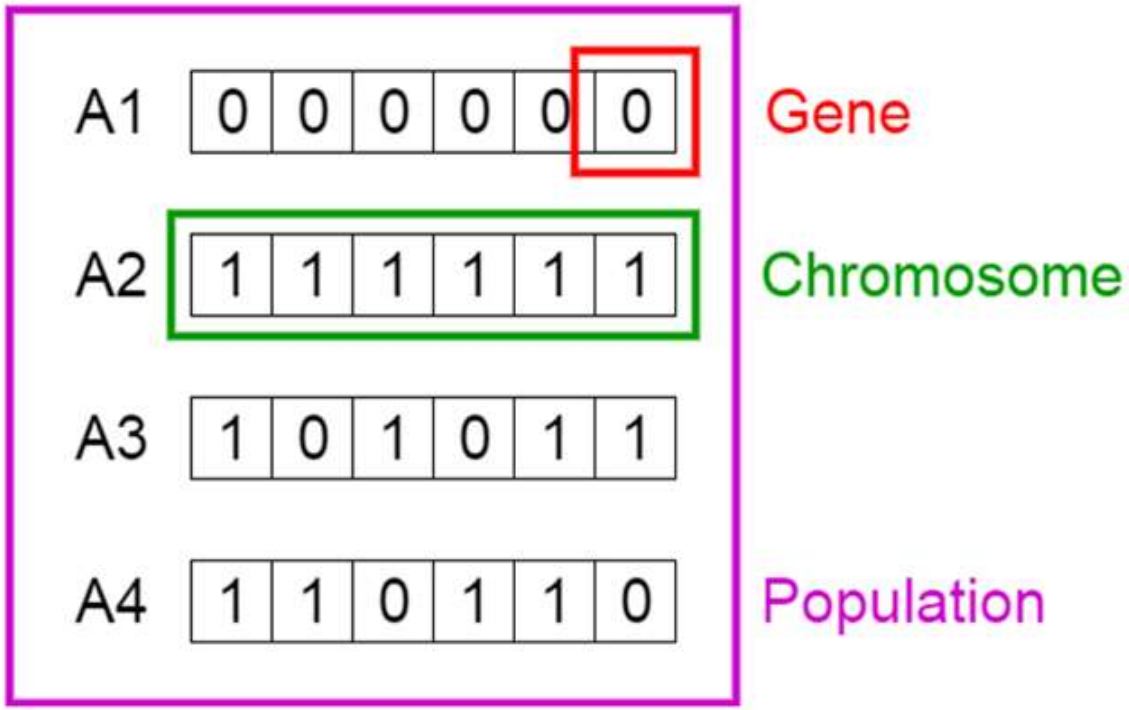


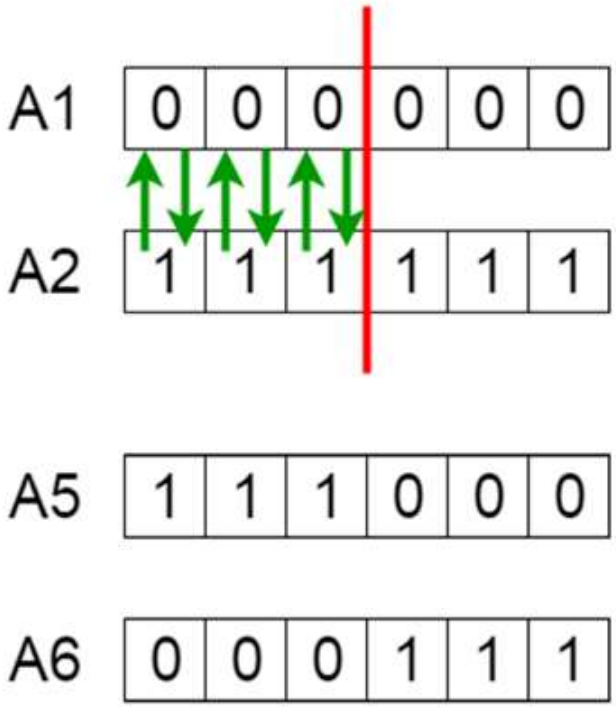
- $t \rightarrow \infty$
- $T \rightarrow 0$
- $Ganho/T \rightarrow -\infty$ (dado que o Ganho é negativo)
- Probabilidade: $\exp(Ganho/T) \rightarrow 0$
- Ou seja: À medida que o tempo passa, a pesquisa arrisca cada vez menos quanto a aceitar alterações com ganho negativo

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

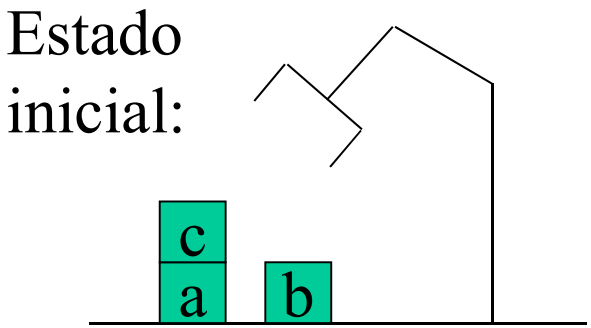
- Pesquisa em árvore
- Pesquisa com propagação de restrições
- Pesquisa por melhorias sucessivas
- Planeamento

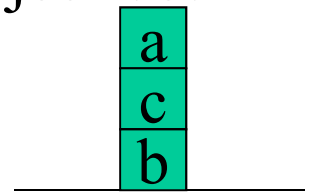
Planeamento: STRIPS, o primeiro planeador

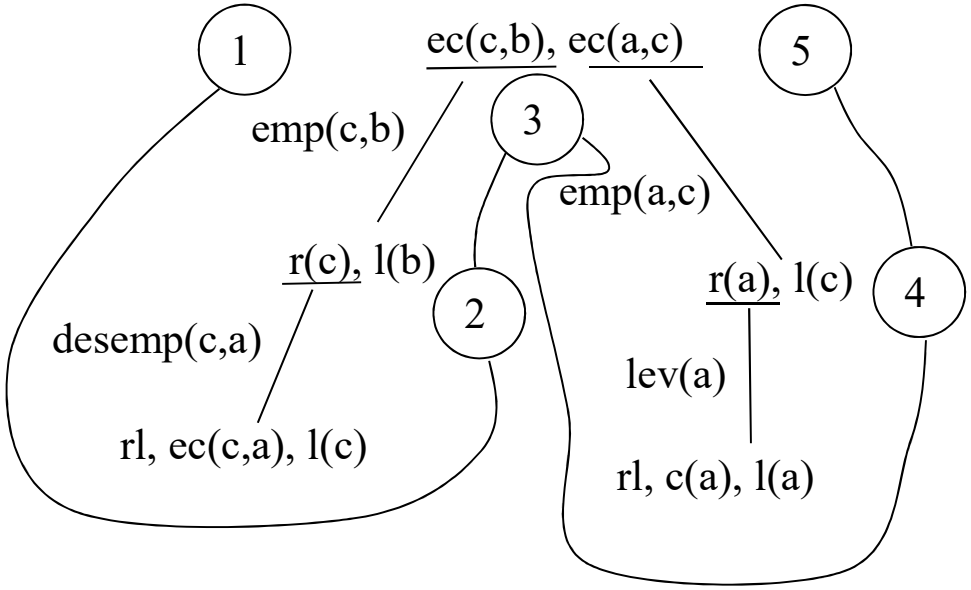
STRIPS(*EI*,*Objectivos*) % *EI* é argumento de entrada/saída
Plano ← [ ]
repetir {
    C ← uma condição em *Objectivos* não satisfeita em *EI*
    OP ← um operador que pode ter *C* como efeito positivo
    A ← acção, dada por uma completa instanciação de *OP*
    PC ← pré-condições de *A*
    SubPlano ← STRIPS(*EI*,*PC*)
    Plano ← concatenar(*Plano*,concatenar(*SubPlano*,[*A*]))
    EI ← 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

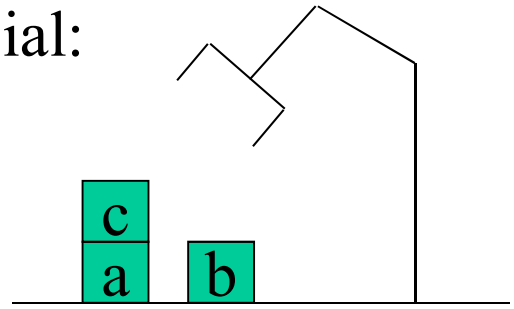




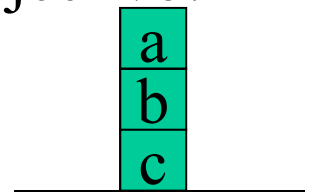


A Anomalia de Sussman

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 atribuido a cada uma das condições que podem aparecer na descrição do estado do mundo.
- Algoritmo:

  1. *CM* ← valor inicial para o nível de criticalidade mínima.

  2. Gerar um um plano que satisfaça todas as condições com nível de criticalidade ≥ CM.

  3. *CM* ← *CM*-1

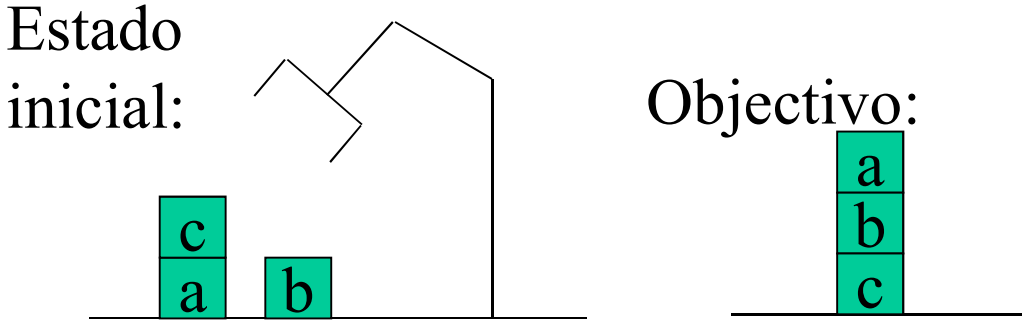
  4. Usando o plano anterior como guia, gerar um plano que satisfaça todas as condições com criticalidade ≥ *CM*.

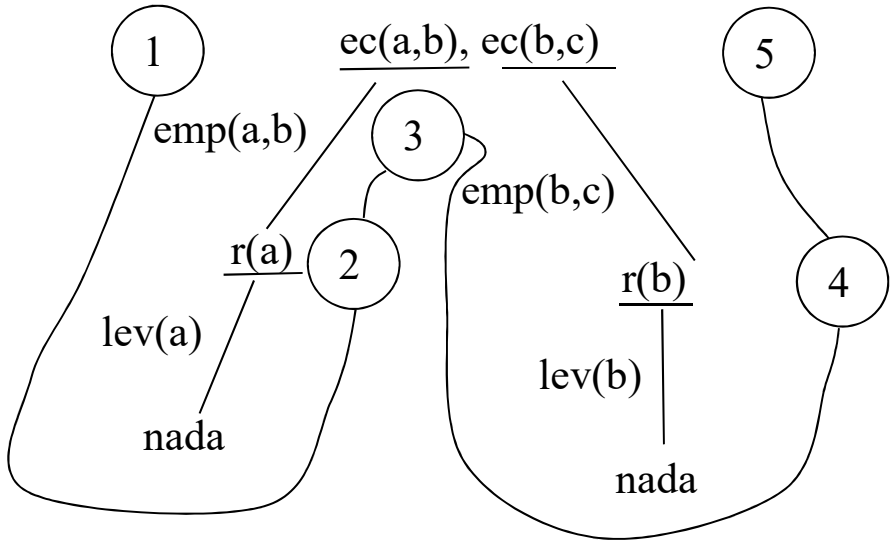
  5. **se** todas as condições estão satisfeitas, **retornar** a solução.

  6. **voltar** ao passo 3.

ABSTRIPS: exemplo – planeamento inicial para CM=2

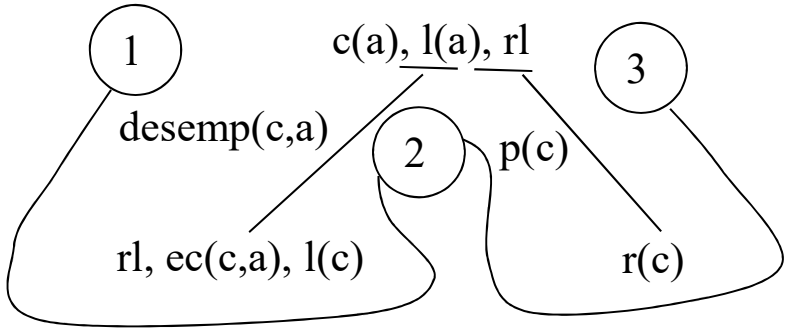
- Dois níveis de criticalidade:
  - ec(A,B) - 2
  - c(B) -1
  - r(X) - 2
  - rl - 1
  - l(X) - 1
- Plano inicial:
  - lev(a), emp(a,b), lev(b), emp(b,c)
- Sucessão de estados:
  - 1: ec(c,a)
  - 2: ec(c,a), r(a)
  - 3: ec(c,a), ec(a,b)
  - 4: ec(c,a), ec(a,b), r(b)
  - 5: ec(c,a), ec(a,b), ec(b,c)
-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





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 atómicas e condicionais

- Literal – uma formula atómica (literal positivo) ou negação de uma fórmula atómica (literal negativo)
- Fórmula de aplicabilidade do operador pode ser:
  - Fórmula atómica
  - 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))))))
```

Aprendizagem

- Aprendizagem é qualquer mudança num sistema que lhe permite ter um melhor desempenho ao executar pela segunda vez uma tarefa [Simon, 1983]
- Aprendizagem é um processo orientado por objectivos através do qual se melhora o conhecimento usando a experiência e o próprio conhecimento [Michalski, 1994]

Aprendizagem = Inferência + Memorização

Aprendizagem: tipos de inferência

- Inferência **dedutiva** – preserva a verdade
  - Especialização dedutiva – restringir o conjunto de referência
    Se $\{ \forall x \ \ x \in A \Rightarrow p(x), B \subset A \} \mid\!- \forall x \ \ x \in B \Rightarrow p(x)$
  - Generalização dedutiva – alargar o conjunto de referência
  - Dedução simples
  - Abstracção
- Inferência **indutiva** – não preserva a verdade, mas é essencial para a aprendizagem
  - Generalização indutiva – alarga o conjunto de referência; é o inverso da especialização dedutiva
    Se $\{ \forall x \ \ x \in B \Rightarrow p(x), B \subset A \} \mid\!- \forall x \ \ x \in A \Rightarrow p(x)$
  - Especialização indutiva – o inverso da generalização dedutiva
  - Abdução – gera uma premissa a partir da qual se poderá deduzir uma dada observação
  - Concretização – Adiciona detalhes sobre o conjunto de referência.

Aprendizagem: níveis de supervisão

- Supervisão
  - Aprendizagem **supervisionada** – cada exemplo contém uma instância do conceito a aprender, que está devidamente identificado
    - Redes neuronais, árvores de decisão, etc.
  - Aprendizagem **semi-supervisionada** – apenas uma (pequena) pequena parte dos exemplos contém informação do conceito a aprender
  - Aprendizagem **por reforço** – o agente aprende o seu comportamento tendo em conta as recompensas (positivas ou negativas) que recebe pelas suas ações
  - Aprendizagem **não supervisionada** – neste caso é o próprio processo de aprendizagem que descobre um novo conceito
    - Algoritmos de agrupamento (*clustering*)

Aprendizagem: os dados

- Quantidade de exemplos
  - Muitos exemplos
    - Redes neuronais, árvores de decisão
  - Um ou poucos exemplos
    - Aprendizagem baseada em explicações (EBL): usa generalização dedutiva
    - Aprendizagem analógica / baseada em casos (CBR)
- Utilização de símbolos e números
  - Aprendizagem simbólica – o conhecimento aprendido está representado numa forma equivalente a lógica proposicional ou de primeira ordem
    - Regras, árvores de decisão, EBL, CBR
  - Aprendizagem conectionista / sub-simbólica
    - Redes neuronais, redes de Bayes, árvores de regressão

Aprendizagem baseada em colecções de exemplos: motivação

- Como aprender a prever qual vai ser a evolução do lucro numa empresa de produtos informáticos?

Idade	Competição	Tipo	Lucro
Velha	Não	Software	Desce
Intermédia	Sim	Software	Desce
Intermédia	Não	Hardware	Sobe
Velha	Não	Hardware	Desce
Nova	Não	Hardware	Sobe
Nova	Não	Software	Sobe
Intermédia	Não	Software	Sobe
Nova	Sim	Software	Sobe
Intermédia	Sim	Hardware	Desce
Velha	Sim	Software	Desce

Aprendizagem com colecções de exemplos: protocolo básico (I)

- Um conjunto de **atributos** ou características
  - $A = \{ A_1, ..., A_n \}$
- Cada atributo pode assumir valores dentro de um conjunto finito de valores simbólicos.
  - $A_i = \{ A_{i1}, ..., A_{ik} \}$
- Os objectos do domínio estão organizados em **classes**
  - $C = \{ C_1, ..., C_m \}$
- O problema é aprender a **reconhecer a classe** (=classificar) do objecto dada uma descrição desse objecto em termos dos atributos em A

Aprendizagem com colecções de exemplos: protocolo básico (I)

- O processo de aprendizagem baseia-se numa colecção de exemplos de treino, *S*.
- É gerada uma função $f : A \rightarrow C$ tal que:
  - $\forall x \; x \in S \; f(x)=classe(x)$
- Por generalização indutiva chega-se a:
  - $\forall$x f(x)=classe(x)
- Isto chama-se **Aprendizagem por Indução**

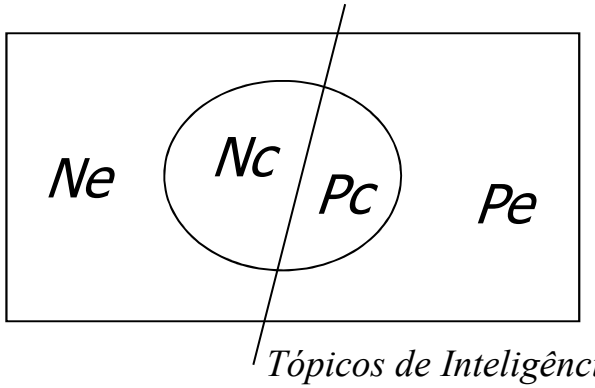
Aprendizagem de regras: pesquisa em profundidade (gulosa)

- Para cada classe *C*, fazer o seguinte

  1. *Regras* $\leftarrow \varnothing$
  2. *Pos* $\leftarrow$ conjunto dos exemplos da classe *C*
  3. *Neg* $\leftarrow$ restantes exemplos
  4. *Antecedente* $\leftarrow$ *Verdade*
  5. Selecionar um novo *Teste*
  6. *Antecedente* $\leftarrow$ *Antecedente* $\wedge$ *Teste*
  7. Se *Antecedente* ainda cobre alguns exemplos em *Neg*, voltar a 5.
  8. *Regras* $\leftarrow$ *Regras* $\cup$ { *Antecedente* $\Rightarrow$ *C* }
  9. *Pos* $\leftarrow$ *Pos* – { exemplos cobertos pelo *Antecedente* }
  10. Se *Pos* $\neq \varnothing$, voltar a 4.

Aprendizagem de regras: critérios

- Estão nesta categoria algoritmos bem conhecidos como o AQ e o CN2
- Critérios de comparação e selecção de regras para refinamento
  - *Pc*/*Tc* – sendo *Pc* o número de exemplos positivos cobertos e *Tc* = *Pc*+*Nc* o número total de exemplos cobertos
  - *Pc*+*Ne* – sendo *Pc* o número de exemplos positivos cobertos e *Ne* o número de exemplos negativos excluídos



Aprendizagem de árvores de decisão: algoritmo

- A árvore de decisão é gerada através de um processo recursivo descendente (*TDIDT – Top-Down Induction of Decision Trees*)

TDIDT(*Exemplos*)
  se todos os *Exemplos* pertencem a uma classe *C*,
    então *Arvore.classe* = *C*;
  senão:
    $A \leftarrow$ atributo de teste para *Exemplos*
    Arvore.teste $\leftarrow A$
    para cada valor a_i de A:
      $E_i \leftarrow$ subjconjunto de *Exemplos* em que $A=a_i$
      $Arvore.subarv_i \leftarrow$ TDIDT(E_i)
  retornar Arvore

Árvores de decisão: selecção do atributo de teste (I)

- Podemos ver o domínio dos exemplos como uma fonte de mensagens, cada uma delas representando uma das classes possíveis
- Baseado na **Teoria da Informação**
  - Entropia *apriori*: $H(C) = -\Sigma\, p(C_i) \times log_2(p(C_i))$
  - Entropia *aposteriori*, dado o valor de um atributo: $H(C|a_{j,k}) = -\Sigma_i\, p(C_i|a_{j,k}) \times log_2(p(C_i|a_{j,k}))$
  - Entropia global *aposteriori*: $H(C|A_j) = \Sigma_k\, p(a_{j,k}) \times H(C|a_{j,k})$

Árvores de decisão: selecção do atributo de teste (III)

- **Ganho de informação**
  - Ou seja, redução da entropia

$$I(C;A_j) = H(C)\text{-}H(C|A_j)$$

- As probabilidades podem ser estimadas com base nos exemplos disponíveis
- Nota: Este método funciona mal quando os atributos têm muitos valores possíveis

Árvores de decisão: selecção do atributo de teste (II)

- **Razão do ganho**
  - $H(A_j) = -\Sigma\, p(a_{j,k}) \times log_2(p(a_{j,k}))$
  - $R(C;A_j) = I(C;A_j) \,/\, H(A_j)$
  - Resolve o problema dos atributos com muitos valores.
  - Quando $H(A_j)$ se aproxima de zero, a razão do ganho fica instável; por isso, são excluídos à partida os atributos cujo ganho de informação seja inferior à média

Árvores de decisão: selecção do atributo de teste (III)

- **Critério GINI**
  - Impureza *apriori*
    - $G = \Sigma_{m \neq n}\, p(C_m) \times p(C_n)$
  - Impureza *aposteriori*:
    - $G(A_j) = \Sigma\, p(a_{j,k}) \times \Sigma_{m \neq n}\, p(C_m \,|a_{j,k}) \times p(C_n|a_{j,k})$

Alguns problemas (I)

- Tratamento do **ruído** – por vezes, os exemplos de treino contém ruido, ou seja, particularidades não representativas do domínio que podem levar o algoritmo de aprendizagem a fazer uma generalização incorrecta.
- **Atributos numéricos** – como usá-los nas regras ou nas árvores de decisão?
- Atributos com valores não especificados nos exemplos

Alguns problemas (II)

- Levar em conta o **custo de cálculo** de cada atributo
- **Aprendizagem incremental**
- Aprendizagem por indução em **lógica de primeira ordem**
  - FOIL

Árvores de decisão: tratamento do ruído

- Parar a expansão da árvore quando o número de exemplos disponíveis é inferior a um dado limiar
- Ter um estimativa do erro, e parar a expansão quando a estimativa do erro começa a subir
- Ter um estimativa do erro, e parar a expansão quando essa estimativa sobe para além de um dado limiar.
- Expandir completamente a àrvore e no fim podá-la.

Avaliação de algoritmos de aprendizagem supervisionada

- **Complexidade computacional**
  - Tanto na aprendizagem como na utilização
- **Legibilidade** – a representação do conhecimento aprendido deve ser tão legível quanto possível
  - Especialmente relevante em sistemas de apoio à decisão
- **Precisão** – o conhecimento aprendido deve ser tão preciso quanto possível
  - No caso de problemas de classificação, a precisão é avaliada experimentalmente como a percentagem de erros de classificação num conjunto de exemplos de teste (não usados na aprendizagem).

Avaliação experimental da precisão em aprendizagem supervisionada (I)

- Partição dos exemplos disponíveis em dois subconjuntos:
  - Subconjunto de treino – exemplos usados para a aprendizagem (p. ex. 2/3 de todos os exemplos)
  - Subconjunto de teste – exemplos usados na avaliação experimental da precisão (p. ex. 1/3)

Avaliação experimental da precisão em aprendizagem supervisionada (II)

- **Validação-cruzada-*k***
  - Divide-se o conjunto de exemplos disponíveis em k subconjuntos
  - Para cada subconjunto S_i, treinar usando todos os outros e testar em S_i
  - A precisão é dada pela percentagem global de erros (após todas as iterações treino-teste)
- **Um-de-fora**
  - Equivale à validação-cruzada-k, para o caso em que k é o número total de exemplos disponíveis

Universidade de Aveiro

Introdução à Inteligência Artificial

(MIECT)

Noções de Programação Declarativa

Ano lectivo 2020/2021

Regente: Luís Seabra Lopes

Programação Declarativa

- Os principais paradigmas de programação declarativa são:
  - Programação funcional
    - baseado no cálculo-lambda
    - a entidade central é a função
  - Programação em lógica
    - baseado na lógica de primeira ordem
    - a entidade central é o predicado

Paradigma imperativo

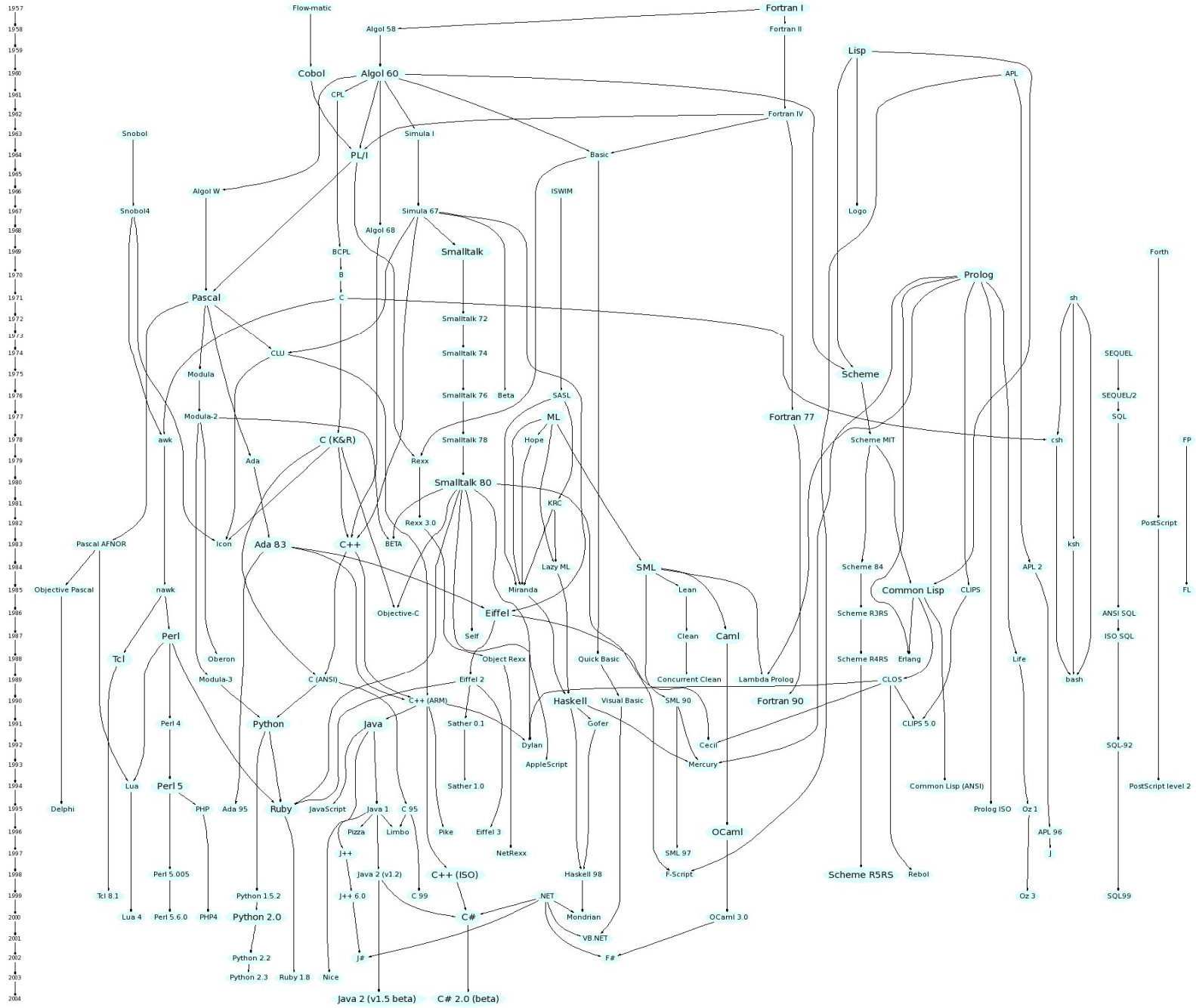
- O fluxo de operações é explicitamente sequenciado
  - Noções de "instrução" e "sequência de instruções"
- Memória
  - Há alterações ao conteúdo da memória (instruções de afectação/atribuição)
  - Pode haver variáveis globais
- Análise de casos: if-then-else, switch/case ...
- Processamento iterativo: while, repeat, for, …
- Sub-programas: procedimentos, funções

Paradigma declarativo

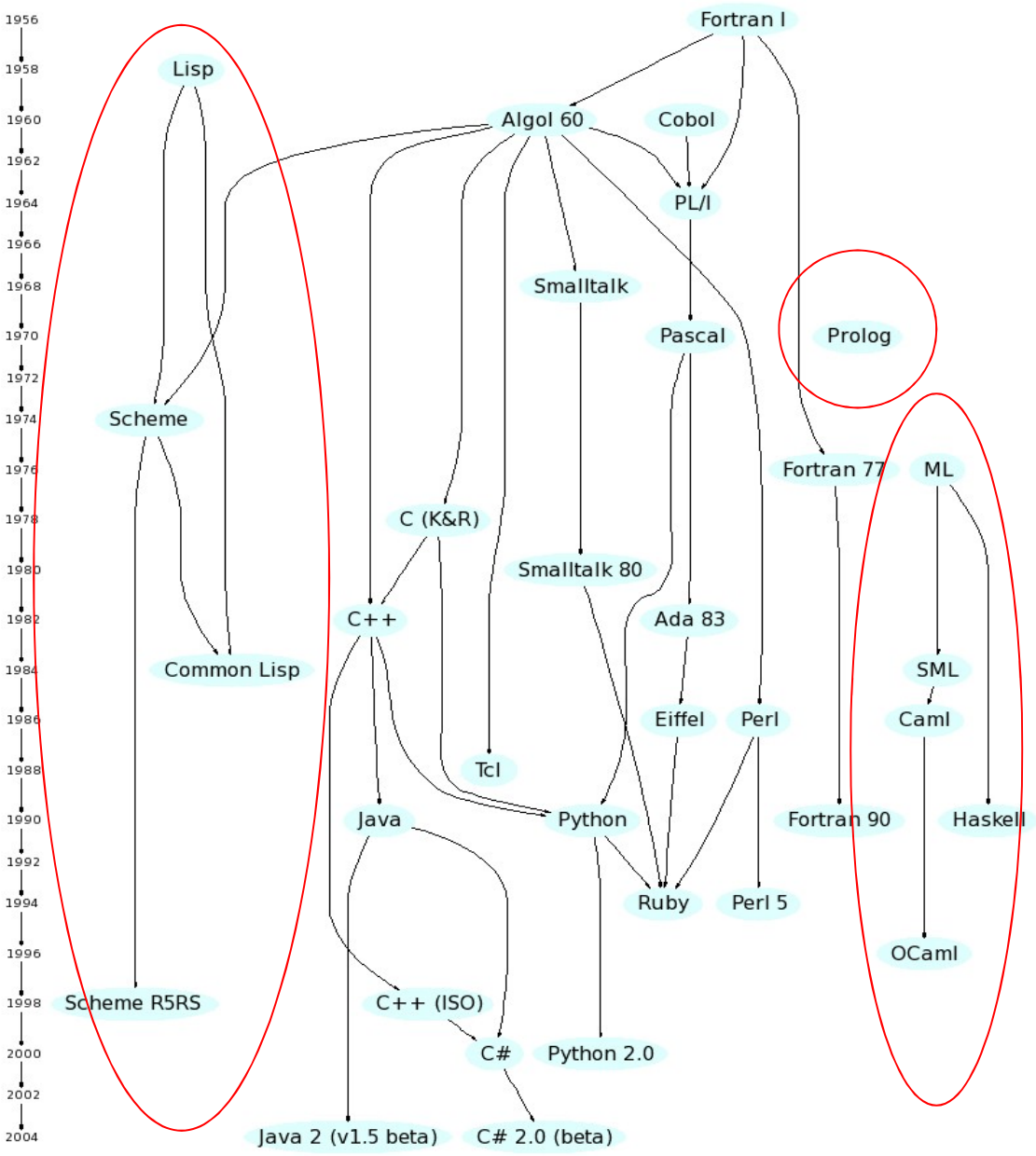
	Funcional	Lógico
Fundamentos	Lambda calculus	Lógica de primeira ordem
Conceito central	Função	Predicado
Mecanismos	Aplicação de funções Unificação uni-direccional Estruturas decisórias	Inferência lógica (resolução SLD) Unificação bi-direccional
Programa	Um conjunto de declarações de funções e estruturas de dados	Um conjunto de fórmulas lógicas (factos e regras)

Programação Declarativa - História

- 1957 - FORTRAN - A primeira linguagem de programação (desenhada na IBM por John Backus)
  - É uma linguagem imperativa
  - Este paradigma foi sendo aperfeiçoado, dando sucessivamente origem a linguagens como Algol (1960), Pascal (1971) e C (1972).
- 1960 - LISP - Inventada por John McCarty, como linguagem para processamento simbólico em problemas de inteligência artificial
- 1970 – Prolog – Inventada por Colmerauer na Univ. Marselha
- 1978 - ML - Desenhada por Robin Milner, como linguagem de comandos para um sistema de prova da correcção de programas
- 1995 – Mercury –Linguagem que integra os paradigmas lógico e funcional, desenvolvida na Univ. Melbourne



IIA: Noções de Programação Declarativa


1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
Fortran I
Lisp
Algol 60
Cobol
PL/I
Smalltalk
Pascal
Prolog
Scheme
Fortran 77
ML
C (K&R)
Smalltalk 80
C++
Ada 83
Common Lisp
SML
Eiffel
Perl
Caml
Tcl
Java
Python
Fortran 90
Haskell
Ruby
Perl 5
OCaml
Scheme R5RS
C++ (ISO)
C#
Python 2.0
Java 2 (v1.5 beta)
C# 2.0 (beta)

Cheiro: Programação Funcional

- Possibilidade de definir funções localmente e sem nome
- Em Lisp:
  - ((lambda (x) (+ (* 2 x) 1)) 6)
  - Resultado: 13
- Em Caml:
  - (fun x -> 2*x+1) 6
  - Equivalente à anterior

Cheiro: Programação em Lógica

- Um programa é uma teoria sobre um domínio
- Exemplo:
  - homem(socrates).
    mortal(X) :- homem(X).
- Pergunta:
  - ?- mortal(socrates).
    Yes

Recursividade "omnipresente"

Caml

```
let rec fact(n) =
  if (n=0)
     then 1
     else n*fact(n-1);;
```

Prolog

```
fact(0,1).
fact(N,FactN):-
      M is N-1,
      fact(M,FactM),
      FactN is N*FactM.
```

C

```
int fact(int n)
{
   if  (n==0) return 1;
   else return n*fact(n-1);
}
```

C (iterativo)

```
int fact(int n)
{
  int f=1, i;
  for(i=1; i<=n ; i++)
    f=f*i;
  return f;
}
```

Atitude do programador

- A programação declarativa, dada a sua elevada expressividade, é pouco compatível com aproximações empíricas (ou "tentativa-e-erro") à programação.
- Convem pensar bem na estrutura do programa antes de começar a digitar
- Aconselham-se os seguintes passos:
  - Perceber o problema
  - Desenhar o programa
  - Escrever o programa
  - Rever e testar

Programação funcional - Características

- A entidade central é a função
- A noção de função é directamente herdada da matemática (*ao contrário, nas linguagens imperativas, o que se chama função é por vezes algo muito diferente de uma função matemática*)
- A estrutura de controlo fundamental é a "aplicação de funções"
- A noção de "tipo da função" captura a noção matemática de domínio (de entrada e de saída)
- Os elementos dos domínios de entrada e saída podem por sua vez ser funções

Origens da Programação Funcional

- Origem na disciplina da lógica matemática, relacionada com o estudo da computabilidade de funções
- 1930 - Alonzo Church inventa o *λ-calculus* (Cálculo Lambda), um sistema formal para descrever funções computáveis
- Já na era do computador, o conceito de função computável tomou o sentido de "função que pode ser implementada através de um programa de computador".
- O Cálculo-Lambda permite definir a semântica das linguagens de programação

Função

- Tem valores de entrada (domínio) e valores de saída (contra-domínio)

Lambda Calculus

- Sistema formal
  - Alonzo Church e Stephen Cole Kleene em ~1930
- Definir formalmente
  - Funções, aplicação de funções, recursividade
- A mais pequena linguagem universal
  - Tudo o que pode ser programado tem equivalente em Lambda calculus
  - Equivalente à máquina de Turing
- Permite provar matematicamente correcção de programas

LISP - I

- LISP = LISt Processing
- Das linguagens de programação que tiveram grande divulgação, LISP é a segunda mais antiga
- Listas são usadas para representar quer os dados quer os programas
- A ideia central é a de “Aplicação de Funções”
- Uso intensivo de funções recursivas
- Permite a definição de funções de ordem superior
- Tem estruturas de decisão condicional
- Não tem um sistema de tipos

LISP - II
Processamento de listas

- A lista é representada como uma sequência de elementos separados por espaços e delimitada por parentesis
  - Exemplo: (a b c d)
- Operações (entre muitas outras):
  - Adicionar elemento à frente: (cons x L)
  - Obter primeiro elemento: (car L)
  - Obter resto: (cdr L)
  - Portanto: (cons (car L) (cdr L)) dá a lista L

LISP - III
Exemplos

- Estrutura de dados
  - (DET 1974 (45 25) (MIECT LTSI MIEET) (MSI))
    - "O DET foi fundado em 1974, tem 45 docentes doutorados, etc.."
- Expressão
  - (f (+ x y) (* u z))
    - Equivalente a f(x+y,u*z)
- Função anónima
  - (lambda(x y) (sqrt (+ (* x x) (* y y))))
- Função com nome:
  - (defun modulo(x y) (sqrt (+ (* x x) (* y y))))

LISP - IV

- LISP foi durante décadas a linguagem dominante no campo da inteligência artificial, e continua a ser usada
- Tradicionalmente interpretada (mas há compiladores há muito tempo também)
- Uma vez que é muito antiga, a linguagem LISP está cheia de "remendos" e tem muitos dialectos:
  - **COMMON LISP** - reune as características comuns aos principais dialectos do LISP
  - **Scheme** - um dialecto do LISP desenhado de raiz por forma a evitar os tais remendos

ML

- ML (= MetaLanguage) - começou por ser uma linguagem de interface para um sistema de prova da correcção de programas
- É essencialmente o formalismo do cálculo-lambda com uma sintaxe mais agradável
- Argumentos avaliados antes da respectiva passagem para o interior da função (*call-by-value*)
- Principais dialectos:
  - SML (= Standard ML) – 1984 – Bell Labs, em cooperação com Edimburgo, Cambridge e INRIA, sob a direcção de Robin Milner
  - Caml – 1987 - desenvolvida no INRIA (França)

Miranda, Haskel

- Constituem um grupo à parte dentro das linguagens funcionais
- Os argumentos são passados não avaliados para o interior das funções – só são avaliados se forem necessários (*lazzy evaluation*)
- Principais linguagens:
  - Miranda (1985)
  - Haskel (1990)

Programação em Lógica

- Um programa numa linguagem baseada em lógica representa uma teoria sobre um problema
- Um programa é uma sequência de frases ou fórmulas representando
  - *factos* - informação sobre objectos concretos do problema / domínio de aplicação
  - *regras* - leis gerais sobre esse problema / domínio
- Implicitamente
  - as frases estão reunidas numa grande conjunção, e
  - cada frase está quantificada universalmente.
- Portanto, ***programação declarativa***.

A linguagem Prolog

- 'Prolog' é acrónimo de 'Programação em Lógica'
- Desenvolvida circa 1970 em Marselha (Colmerauer) e Edimburgo (Kowalski, Pereira, Warren)
- Execução de um programa Prolog é dirigida pela informação necessária para resolver um problema e não pela ordem das instrucções de um programa
  - Um programa Prolog começa com uma pergunta (query)
- Mecanismos centrais:
  - unificação,
  - estruturas de dados baseadas em listas e árvores,
  - procura automática de alternativas

Prolog - programas

- Factos são fórmulas atómicas, ou seja, fórmulas que consistem de um único predicado. Exemplos:
  - lecciona(lsl, iia).
    mulher(joana).
    aluno(alfredo,ect,ua).
- As regras são implicações com um único consequente e um ou mais antecedentes. Exemplo:
  - professor(X) :- lecciona(X,Y).
  - Isto é equivalente à seguinte frase em lógica
    $\forall x\ (\exists y\ \text{Lecciona}(x,y)) \Rightarrow \text{Professor}(x)$
- Sintaxe
  - Constantes começam com minúscula
  - Variáveis começam com maiúscula ou '_'

Mercury

- Linguagem de programação lógica/funcional desenvolvida na Universidade de Melbourne, Austrália
- Primeira versão em 1995
- Modular
- Tipagem forte
- Permite a definição de predicados através de factos e regras (*Horn clauses*) tal como no Prolog e com síntaxe idêntica
- Permite a definição de funções e tipos de dados como em ML, Caml, etc., embora com diferente síntaxe
- Permite a definição de predicados e funções de ordem superior
- Linguagem compilada, ao contrário do que acontece com outras linguagens declarativas

Mercury (cont.)

- Ênfase na pureza declarativa
  - Do ponto de vista do paradigma lógico, é uma linguagem mais pura do que o Prolog
  - Tem pesquisa automática de alternativas (retrocesso), mas não suporta o corte de alternativas (*cut*)
  - Algumas optimizações possibilitadas pela pureza da linguagem permitem maior rapidez de execução