IA – POWERPOINT 2 – RESUMO

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 "sequências de instruções"
- Memória
    - Há alterações ao conteúdo da memória (instruções de afetação/atribuição)
    - Pode haver variáveis globais
- Análise de casos: if-then-else, switch/case, …
- Processamento iterative: 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 Funcional

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

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

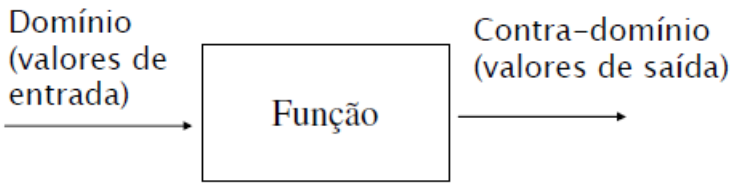
- A programação declarativa, dada a sua elevada expressividade, é pouco compatível com aproximações empíricas (ou "tentativa-e-erro") à programação.
- Convém pensar vem na estrutura d programa antes de começar a digitar
- Aconselham-se os seguintes passos:
    - Perceber o problema
    - Desenhar o programa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

Programação funcional - Características

- A entidade central é a função
- A noção de função é diretamente herdade 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

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 correção de programas

LISP

- 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

ML

- ML (= MetaLanguage) - começou por ser uma linguagem de interface para um sistema de prova da correção de programas
- É essencialmente o formalismo do cálculo-lambda com uma sintaxe mais agradável
- Argumentos avaliados antes da respetiva passagem para o interior da função (call-by-value)
- Principais dialetos:
    - SML (= Standard ML) - 1984 - Bell Labs, em cooperação com Edimburgo, Cambridge e INRIA, sob a direção 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 (lazy 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 objetos 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'
- Execução de um programa Prolog é dirigida pela informação necessária para resolver um problema e não pela ordem das instruções 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

$$\forall x \left(\exists y \, Lecciona(x, y)\right) \Rightarrow Professor(x)$$

- Sintaxe
    - Constantes começam em minúscula
    - Variáveis começam com maiúscula ou '_'

IA – POWERPOINT 3 – RESUMO

Principais características da linguagem de programação Python:
- Interpretada
- Interativa
- Portável
- Funcional
- Orientada a Objetos
- Implementação aberta

Objetivos da linguagem:
- Simplicidade sem prejuízo 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
  - Objetos
  - Métodos
  - Herança
- Programação imperativa / modular
  - Instrução de atribuição
  - Sequências de instruções
  - Análise condicional (if-elif-else)
  - Ciclos for, while
  - Sistema de módulos

Python vs Java
- Código mais conciso
- Verificação de tipos dinâmica
- Desenvolvimento mais rápido
- Não compila para código nativo
- Porém, códigos mais lentos

Python – áreas de aplicação
- Interligação de sistemas

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

Dados, ou "objetos"

- Objeto – no contexto de Python, esta designação é aplicada a qualquer dado que possa ser armazenado numa variável, ou passado como parâmetro a uma função
- Cada objeto é caracterizado por: identidade ou referência (identifica a posição da memória onde está armazenado), tipo e valor
- Alguns tipos de objetos podem ter atributos e métodos
- Alguns tipos (classes) de objetos podem ter sub-tipos (sub-classes)

Sequências de dados

- Cadeias de caracteres (str)
- Tuplos (tuple) – agregados ou composições de cários 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) | ( )
- 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 vetores na programação imperativa
  - é possível modificar elementos individuais das listas
  - Os elementos são separados por vírgulas (,) e delimitados por parênteses retos
  - Exemplos: [1,2,'a'9] | [("maria",33),("josé",40)] | ['lisboa',[7,"colinas"]] | [ ]

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

Acesso a sequências

- 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

Instrução de atribuição

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

Funções recursivas

```
# 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
```

```
# 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

- São expressões cujo valor é uma função
- São um "ingrediente" clássico da programação funcional
- Exemplos:
    1. lambda x : x+1
        - Função que dado um valor x, devolve x+1
    2. 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
    3. (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
- Como qualquer objeto, 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: 8*7=56
- 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
- As expressões lambda também são 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 Python3, 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 Python3, esta função retorna um iterador que pode ser convertido para lista

Reduzir uma lista a um valor

- Muitos procedimentos que atuam 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:]).
- Dada uma função de combinação f, uma lista e um valor neutro, devolve a redução da lista:

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

Listas de compreensão
- Mecanismo compacto para processas 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
- Sintaxe (caso simples):
```
[<expr> for <var> in <sequência> if <condição>]
```
- 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]
```
- 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]
```
- 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]
```
- 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 objetos
  - Uma classe define um conjunto de objetos caracterizados por diversos atributos e métodos
  - É possível definir hierarquias de classes com herança
- As classes surgem na linguagem Python com pouca sintaxe adicional
- Sintaxe:
```
class <nome-classe>:
        <declaração-1>
        …
```

<declaração-N>

- Exemplo

```
class UmTeste:                          Por convenção, as palavras no nome de uma classe iniciam-se com maiúscula
    def dizer_ola(self):                Exemplo de definição de um método
        print "Ola"
```

- Utilização

```
>>> x = UmTeste()                       Criação de uma instância e atribuição a uma variável
>>> x.dizer_ola()                       Invocação do método
Ola
```

Classes com construtor

- Exemplo

```
class Complexo:
    def __init__(self,real,imag):       O construtor é o método que inicializa um objeto 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
        self.r = real
        self.i = imag
```

- Utilização

```
>>> c = Complexo(-1.5,13.1)             Criação de uma instância e atribuição a uma variável
>>> c.r,c.i
(-1.5,13.1)
```

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 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 programados para representar
    - soma de expressões
    - produtos de expressões
    - constantes
    - ocorrências da variável
- 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:

```
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, 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 caracteres; suporta a função de conversão str()
    - __repr__() – define a representação em cadeia de caracteres que aparece na consola do interpretados; suporta a função repr()
- Atributos
    - __class__ - identifica a classe de um dado objeto
        - 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)
    - …

IA – POWERPOINT 4 – RESUMO

Definição de Inteligência

- 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 Artificial

- "Inteligência Artificial" é a disciplina que estuda as teorias e técnicas necessárias ao desenvolvimento de "artefactos" inteligentes

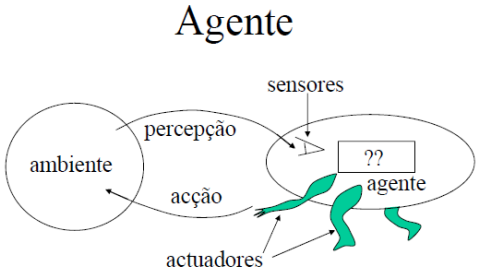
Tópicos de Inteligência Artificial

- Agentes
    - Noção de agente
    - Objetivo da Inteligência Artificial
    - Agentes reativos e deliberativos
    - Propriedades do mundo de um agente
    - Arquiteturas de agentes
- Representação do conhecimento
- Técnicas de resolução de problemas

Definição de "Agente"

- Nesta disciplina estudamos técnicas úteis no desenvolvimento de "agentes inteligentes"
- Um "agente" pode ser:
    - Entidade com poder ou autoridade de agira
    - Entidade que atua em representação de outrem

- Agente – uma entidade com capacidade de obter informação sobre o seu ambiente (através de "sensores") e de executar ações em função dessa informação (através de "atuadores")
- Exemplos:
    - Agente físico: robô anfitrião
    - Agente de software: agente móvel de pesquisa de informação na internet



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 atividades cognitivas.
- 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 objetivo principal da "Inteligência Artificial"?

A "sala chinesa" de Searle

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

O argumento de Searle

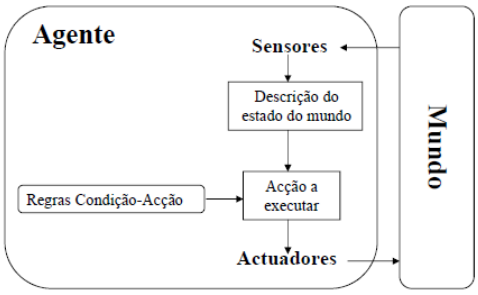
- O humano não percebe chinês
- A sala não percebe chinês
- O livro de regras e as folhas de papel também não percebem chinês
- Logo, não há qualquer compreensão de chinês naquela sala

- No entanto, podemos contra-argumentar: embora individualmente, os componentes do sistema (a sala, o humano, o livro, as folhas de papel) não compreendam chinês, o sistema no seu conjunto compreende chinês.

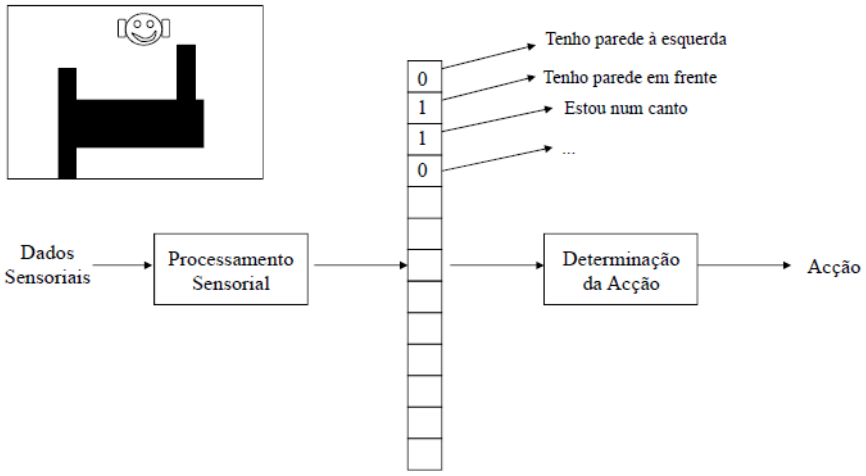
Tipos de arquiteturas de agente

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

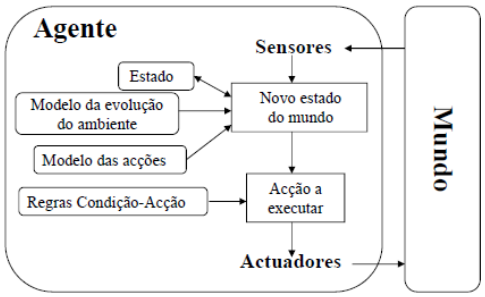
Agente reativo: simples

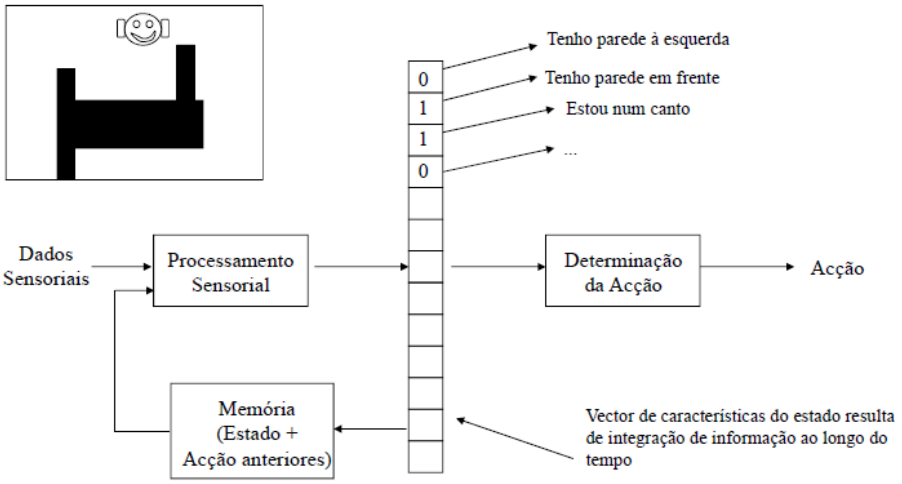


- O conceito de “regra de condição-ação” é também conhecido como “regra de situação-ação” ou “regra de produção”
- Os agentes ou sistemas reativos simples são também conhecidos como “sistemas de estímulo-resposta” ou “sistemas de produção”



Agente reativo: com estado interno

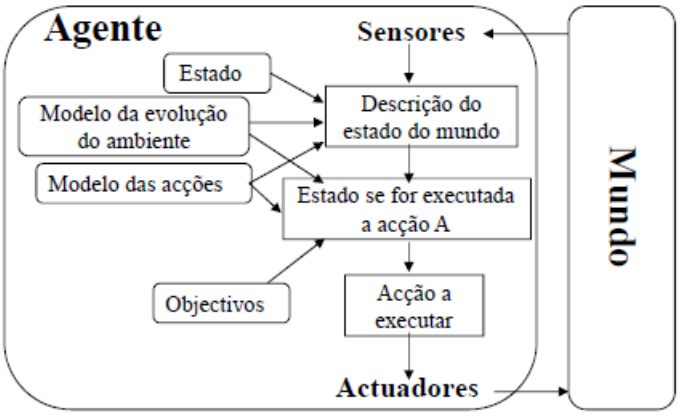




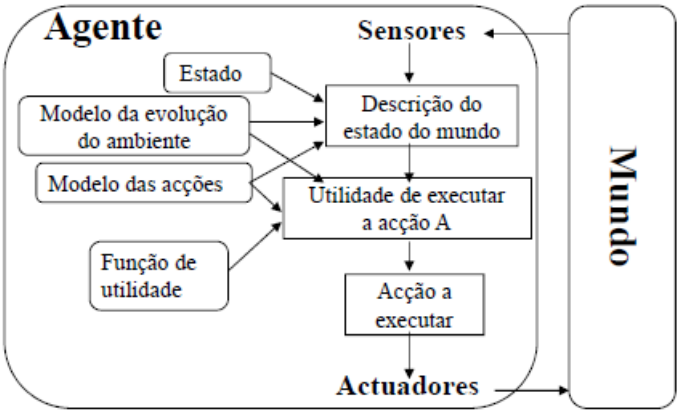
Sistemas de Quadro Preto

- Podem ser vistos como uma elaboração dos sistemas reativos com estado interno.
- Uma "doente 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-ação.

Agente deliberativo: orientado por objetivos



Agente deliberativo: orientado por função de utilidade



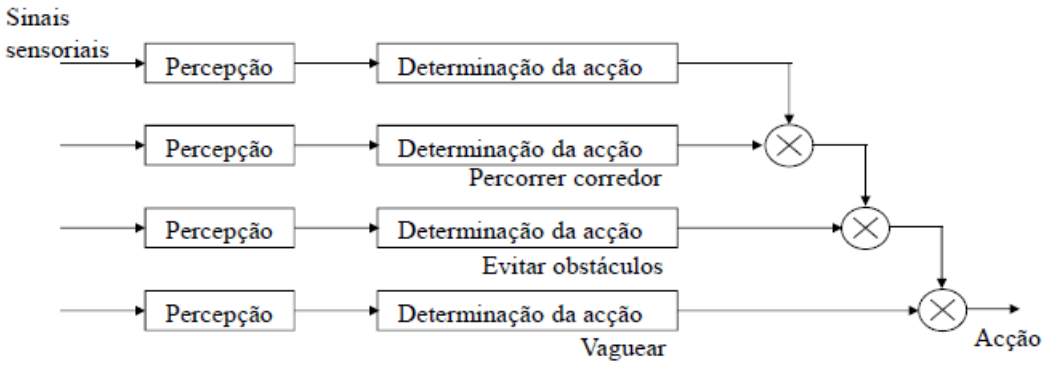
Propriedades do mundo de um agente

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

Mundo de um agente - Exemplos

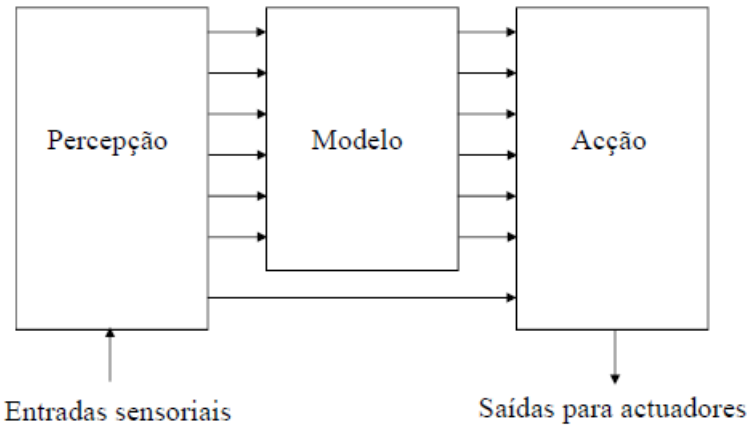
Mundo	Acessível	Determinístico	Episódico	Dinâmico	Continuo
Xadrês s/ relógio	Sim	Sim	Não	Não	Não
Xadrês c/ relógio	Sim	Sim	Não	Semi	Não
Poker	Não	Não	Não	Não	Não
Condução de carro	Não	Não	Não	Sim	Sim
Diagnóstico médico	Não	Não	Não	Não	Sim
Sistema de análise de imagem	Sim	Sim	Sim	Semi	Sim
Manipulação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

Arquiteturas de agentes: Subsunção

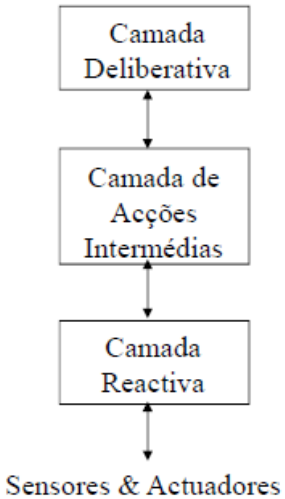


- A arquitetura de subsunção procura estabelecer a ligação entre perceção e ação a vários níveis – daqui resulta uma organização em camadas.
- A camada mais baixa é a mais reativa
- O peso da componente deliberativa aumenta à medida que se sobre na estrutura de camadas.

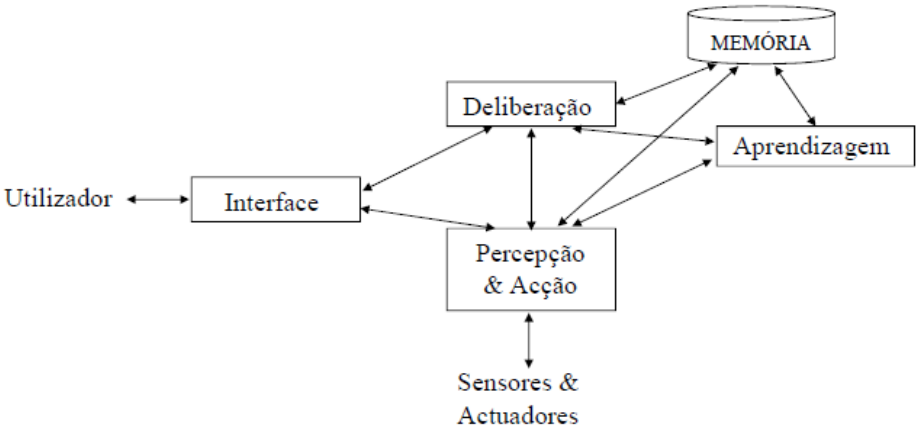
Arquitetura de Agentes: Três Torres



Arquiteturas de Agentes: Três Camadas



Arquiteturas de Agentes: CARL



Tópicos de Inteligência Artificial

- Agentes
    - Noção de agente
    - Objetivo da Inteligência Artificial
    - Agentes reativos e deliberativos
    - Propriedades do mundo de um agente
    - Arquiteturas de agentes
- Representação do conhecimento
- Técnicas de resolução de problemas

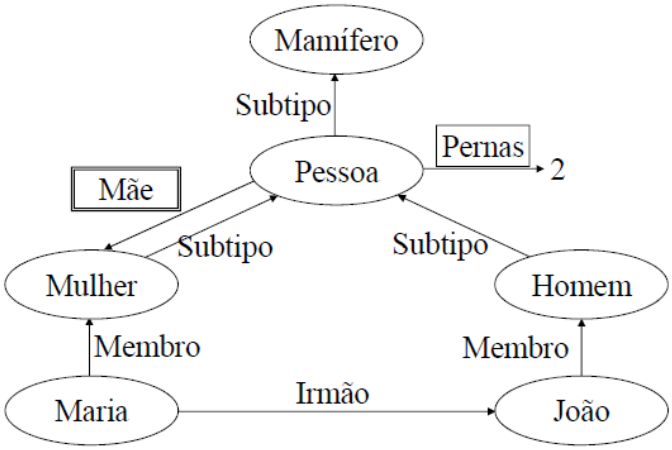
Representação do conhecimento

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

Redes Semânticas

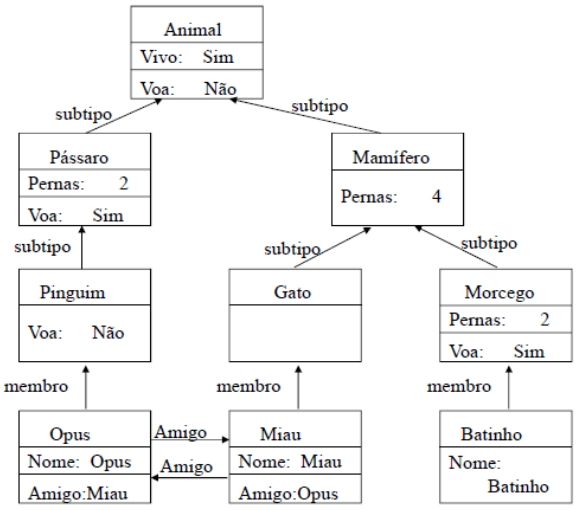
- Redes semânticas são representações gráficas do conhecimento
- Têm a vantagem da legibilidade
- As redes semânticas podem ser tão expressivas quanto a lógica de primeira ordem

Redes semânticas - 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 abstratos dos quais descende
    - A instância herda todas as propriedades do tipo a que pertence
- A inferência pode ser vista como o seguimento das ligações entre entidades com vista à herança de propriedades
- Pode implementar-se raciocínio não monotónico através do estabelecimento de valores por defeito e o correspondente cancelamento da herança
- 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 contrapartida, nomeadamente nos chamados sistemas frames, usam-se métodos e demónios:

- Métodos têm uma semântica similar à da programação orientada por objetos

- Demónios são procedimentos cuja execução é disparada automaticamente quando certas operações de leitura ou escrita são efetuadas.

Redes semânticas vs 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

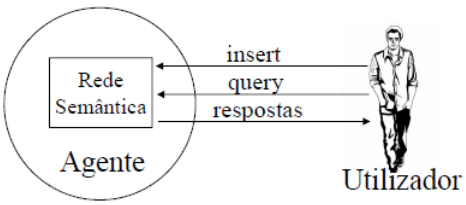
- Dedução - permite inferir casos particulares a partir de regras gerais
    - Preserva a verdade
    - as regras de inferência apresentadas anteriormente são regras dedutivas
- Indução - é o oposto da dedução; permite inferir regras gerais a partir de casos particulares
    - É a base principal da aprendizagem

Indução

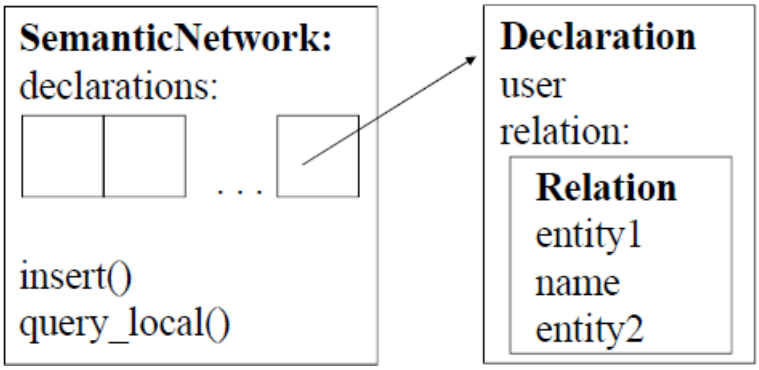
- Exemplo
    - Casos conhecidos
        - O gato Tareco gosta de leite
        - O gato Pirata gosta de leite
    - Regra inferida
        - Os gatos normalmente gostam de leite
    - Nas redes semânticas, a indução pode ser vista como uma "herança de baixo para cima"

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)



- Uma relação pode ser dos três tipos seguintes:
    - Member(obj,type) - um objeto é membro de um tipo
    - Subtype(subtype,supertype) - um tipo é subtipo de outro
    - Association(entity1,name,entity2) - uma entidade (objeto 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 interação com outros agentes.



Representação do conhecimento

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

Lógicas

- Uma lógica tem:
    - Sintaxe - 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 uma frase composta por uma única variável proposicional

Lógica de Primeira Ordem

- Componentes:
    - Objetos ou entidades
        - Exemplos: 1214, DDinis, Aveiro
    - Expressões funcionais
        - Exemplos: Potencia(4,3), Pai-de(Paulo)
        - Nota 1: Os objetos podem ser considerados como expressões funcionais cuja aridade é zero
        - Nota 2: A noção de termo engloba quer os objetos 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 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 predicados podem ser variáveis, usadas para representar termos não especificados

- Exemplos: x, y, pos, soma, pai,…

- Quantificação universal
  - $\forall x$A = “Qualquer que seja x, a fórmula A é verdade”
  - Se A é uma fórmula bem formada, então $\forall x$A também é uma fórmula bem formada

- Quantificação existencial
  - $\exists x$A = “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) ⇒ Esperto(x)
  - Erro comum: Usar ∧ em vez de ⇒
    ∀x Estuda(x,Oxford) ∧ Esperto(x)
    Significa “Todos estão em Oxford e todos são espertos”

- “Alguém em Oxford é esperto”:
  - $\exists x$ Estuda(x,Oxford) ∧ Esperto(x)
  - Erro comum: Usar ⇒ em vez de ∧
    ∃x Estuda(x,Oxford) ⇒ 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 se a fórmula toma o valor 'verdadeiro' para essa interpretação.
- Modelo de uma fórmula - uma interpretação que satisfaz essa fórmula.
- Tautologia - uma fórmula cujo valor é '' verdadeiro em qualquer interpretação

Interpretações em Lógica de Primeira Ordem

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

- 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:

$$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$$

- 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:

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

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

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

CNF e Formal 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 respetivas cláusulas.

Conversão de uma Fórmula Proposicional para CNF e forma clausal

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

- Através dos seguintes passos:
    - Renomear variáveis, de forma 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
    - 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
    - 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 que uma variável não apareça em mais do que uma fórmula
- 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: $\forall a\ \forall b\ \ (\ p(a,b) \wedge \exists c\ \neg q(c,c)\ )$
- Skolemizada aplicada: $\forall a\ \forall b\ \ (\ p(a,b) \wedge \neg q(f(a,b),\ f(a,b))\ )$
- Quantificadores removidos: $p(a,b) \wedge \neg q(f(a,b),\ f(a,b))$
- Convertida para a forma clausal: $\{\ p(a,b)\ ,\ \neg q(f(a,b),\ f(a,b))\ \}$
- Variáveis renomeadas: $\{\quad p(a_1,b_1)\ ,\ \neg q(f(a_2,b_2),\ f(a_2,b_2))\ \}$

Lógica - Regras de Inferência

Modus Ponens: $\{\ A,\ A \Rightarrow B\ \}\ |\!-\ B$

Modus Tolens: $\{\neg B,\ A \Rightarrow B\ \}\ |\!-\ \neg A$

Silogismo hipotético: $\{A \Rightarrow B,\ B \Rightarrow C\ \}\ |\!-\ A \Rightarrow C\ \}$

Conjunção: $\{\ A,\ B\ \}\ |\!-\ A \wedge B$

Eliminação da conjunção: $\{A \wedge B\ \}\ |\!-\ A$

Disjunção: $\{\ A,\ B\ \}\ |\!-\ A \vee B$

Silogismo disjuntivo (ou resolução unitária):
$\{\ A \vee B,\ \neg B\ \}\ |\!-\ A$

Resolução: $\{\ A \vee B,\ \neg B \vee C\ \}\ |\!-\ A \vee C$

Dilema construtivo:
$\{\ (A \Rightarrow B\ ) \wedge (C \Rightarrow D),\ A \vee C\ \}\ |\!-\ B \vee D$

Dilema destrutivo:
$\{\ (A \Rightarrow B\ ) \wedge (C \Rightarrow D),\ \neg B \vee \neg D\ \}\ |\!-\ \neg A \vee \neg C$

Lógica de Primeira Ordem - Regras de Inferência específicas

Instanciação universal:
$\{\forall x\ P(x)\ \}\ |\!-\ P(A)$

Generalização existencial
$\{P(A)\ \}\ |\!-\ \exists x\ P(x)$

Consequências Lógicas, Provas

- Consequência lógica
  - Diz-se que A é consequência lógica do conjunto de fórmulas 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 {A1, A2, …, An} é uma prova (ou dedução) de An a partir de um conjuntos de fórmulas Δ ou pode ser inferida a partir das fórmulas A1 ... Ai-1.
  - Neste caso escreve-se $\Delta \vdash An$

Correção, Completude

- Correção - Diz-se que um conjunto de regras de inferência é correto 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 correto e completo permite tirar consequências lógicas sem ter de analisar caso a caso as várias interpretações.

Metateoremas

- Teorema da dedução:
  - Se {A1, A2, …, An} |= B, então A1∧A2∧…∧An ⇒ B, e vice-versa.
- Redução ao absurdo:
  - Se o conjunto de fórmulas Δ é satisfazível (logo tem pelo menos um modelo) e $\Delta \cup \{\neg A\}$ não é satisfazível, então $\Delta \mid = A$.

Resolução não é Completa

- A resolução é uma regra de inferência correta (gera fórmulas necessariamente 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:

$$\{A \vee B\} \mid - A \vee B$$

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 (metateorema 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 = { t1/x1, …, tn/xn } a uma fórmula W denota-se SUBST(W,s) ou Ws; Significa que todas as ocorrências das variáveis x1, …, xn em W são substituídas pelos termos t1, …, tn
- 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 ∨ B, ¬C ∨ D } |- SUBST(A ∨D, g) em que B e C são unificáveis sendo g a sua substituição unificadora mais geral
- A regra da resolução é correta
- A regra da resolução não é completa

- Tal como na lógica proposicional, também aqui a refutação por resolução é completa

Resolução com Cláusulas de Horn

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

$$A \qquad \neg A \vee B$$
$$\neg A \vee B \vee \neg C \qquad \neg A \vee \neg B$$

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

KIS (=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 mata-conhecimento (ou seja, conhecimento sobre o conhecimento)

KIF - características gerais

- O mundo é conceptualizado em termos de objetos e relações entre objetos.
- Uma relação é um conjunto arbitrário de listas de objetos
    - 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 objetos cuja existência é conhecida, presumida ou suposta
    - Os objetos podem ser concretos ou abstratos

- Os objetos 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
    - Código de caracteres
    - Blocos de códigos de caracteres
    - Cadeias de caracteres
- Expressões
    - Termos - objetos com valor lógico
    - 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 cl 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: treu, false
- Equação
    (= termo1 termo2)
- Inequação
    (/= termo1 termo2)
- Frase relacional
    (relação t1 .. tn)
- Frase lógica: construída com as conectivas lógicas (‘not’, ‘and’, ‘or’, ‘⇐’, ‘⇒’, ‘⇔’)
- Frase quantificada
    (forall var1 … varn frase)
    (exists var1 … varn frase)

KIF - definições

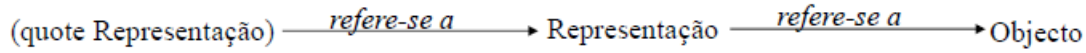
- Definição de objetos
    - Igualdade: (defobject s := t)
        - Exemplo: (defobject nil := (listof))

- Conjunção: (defobject s p1 .. pn)
- etc.

- Definições de funções
  - (deffunction f(v1 .. vn) := t)
  - Exemplo:
    - (deffunction head (?l) := (if (= (listof ?x @items) ?l) ?x)
- Definição de relações (=predicados)
  - (defrelation r (v1 .. vn) := p)
  - etc.
  - Exemplo:
    - (defrelation null (?l) := (= ?l (listof)))
    - (defrelation list (?x) := (exists (@1) (= ?x (listof (@l)))))

KIF - meta-conhecimento

- Pode formalizar-se conhecimento sobre o conhecimento
- O mecanismo da citação (quotation) permite tratar expressões como objetos
- 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 objeto ou pessoa a que ela se refere.
- Outros exemplos:
  - (acredita joão '(material lua queijo))
  - (=> (acredita joão ?q) (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 seleção 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 functores e 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

- Sistema de “frames”
- Herança e raciocínio não-monotónico
- Relação com diagramas UML
- Exemplo para aulas práticas

- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes

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 atividade 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 objetos, conceito e relações que será necessário representar
  - Escolher um vocabulário para entidades, funções e relações (por vezes chamado ontologia)
  - Codificar conhecimento genérico sobre o domínio (um conjunto de axiomas)
  - Codificar descrições para problemas concretos, interrogar o sistema e obter respostas.
  - Por vezes o domínio é tão complexo que não é praticável codificar à mão todo o conhecimento necessário. Neste caso usa-se aprendizagem automática.

Identificação de objetos, conceitos e relações

- 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 objetos, conceitos e relações
  - Substantivos comuns -> conceitos (também chamados classes ou tipos)
  - Substantivos próprios -> objetos (também chamados instâncias)
  - Verbo “ser” -> pode indicar uma relação de instanciação (entre objeto 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
- Convém avaliar a importância para o problema das palavras utilizadas bem como dos objetos, conceitos e relações subjacentes
  - Não considerar substantivos que identifiquem objetos, 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 abstrato pode ser criado atribuindo-lhe o que é comum a outros dois ou mais conceitos previamente identificados

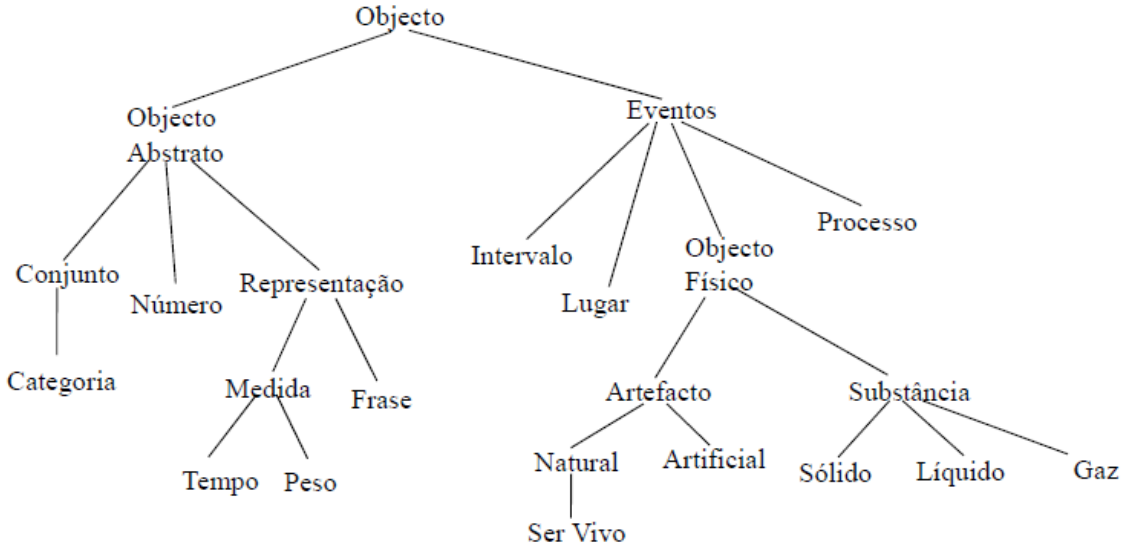
Ontologias

- Uma ontologia é um vocabulário sobre um domínio conjugado com relações hierárquicas como membro e subtipo e eventualmente outras.
- O objetivo 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
  - Objetos compostos
  - Tempo, espaço e mudanças
  - Eventos e processos (eventos contínuos)
  - Objetos físicos
  - Substâncias
  - Objetos abstratos e crenças

Uma possível ontologia geral



Representação do conhecimento

- Redes semânticas
  - Redes semânticas genéricas
  - Sistema de “frames”
  - Herança e raciocínio não-monotónico
  - Relação com diagramas UML
  - Exemplo para aulas práticas

- Lógica proposicional e lógica de primeira ordem
- Linguagem KIF
- Engenharia do conhecimento
- Ontologia geral
- Redes de Bayes

Redes de crença bayesianas

- Também conhecidas simplesmente como “redes de Bayes”
- Permite representar conhecimentos impreciso em termos de um conjunto de variáveis aleatórias e respetivas 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 interceção

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

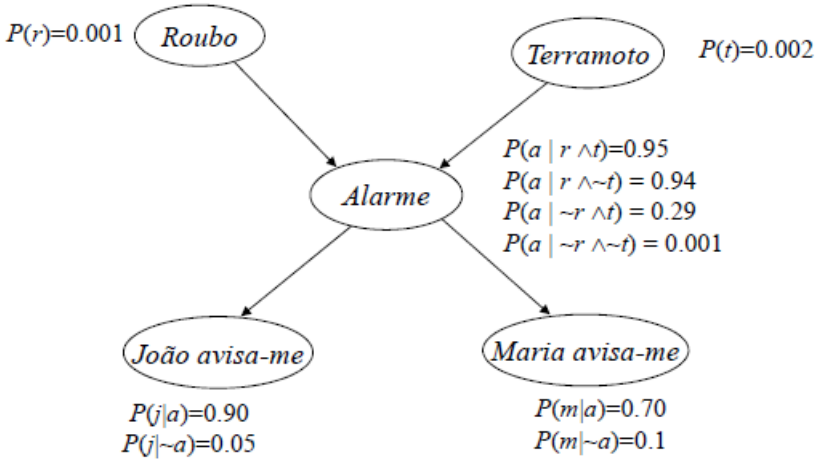
Probabilidades condicionadas

- 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:

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

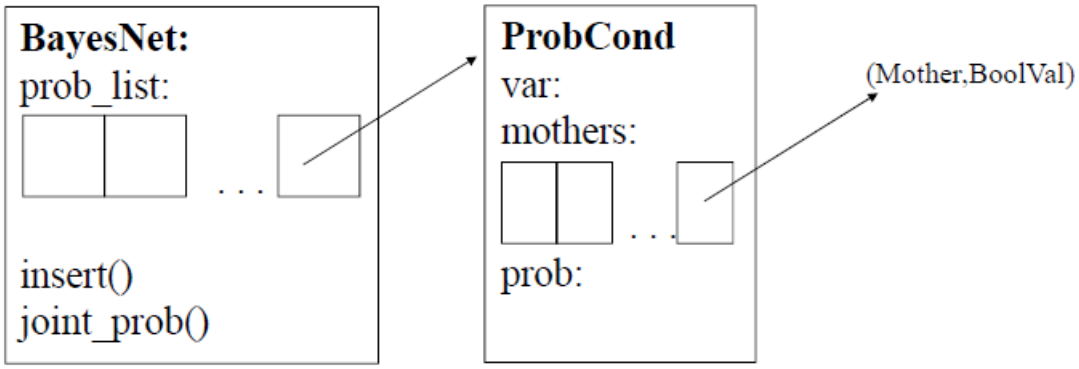
- Assim, no exemplo anterior, a probabilidade de o alarme tocar e o João e a Maria ambos avisarem num cenário em que não há roubo nem terramoto, é dada por:

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

Redes Bayesianas em Python

- Vamos criar uma rede de crença bayesianas, 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
    - join_prob - obter a probabilidade conjunta para uma dada conjunção de valores de todas as variáveis da rede

Redes de crença em Python



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 a rede bayesiana)
    - Exemplo: o conjunto dos ascendentes de "João avisa" é {"alarme", "roubo" e "terramoto"}

Redes de crença bayesianas - probabilidade individual

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

- Seja:
    - C = {x1, …, xn} - conjunto de variáveis da rede
    - xi ∈ C - uma qualquer variável da rede
    - vi ∈ {v,f} - valor de xi cuja probabilidade se pretende calcular
    - {a1, …, ak} ⊂ C - conjunto das variáveis da rede que são ascendentes de xi

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

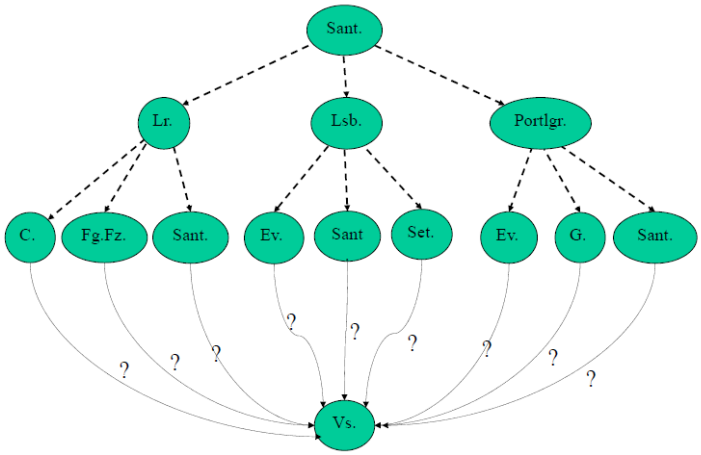
- Um problema é algo (um objetivo) cuja solução não é imediata
- Por isso, a resolução de um problema requer a pesquisa de uma solução
- Um problema é algo cuja solução é imediata
- Exemplos de problemas:
    - Dado um conjunto de axiomas, demonstrar um novo teorema
    - Dado um mapa, determinar o melhor caminho entre os dois pontos
    - Dada uma situação num jogo de xadrez, determinar uma boa jogada
    - etc

Formulação de problemas e pesquisa de soluções

- A formulação de um problema inclui:
    - Descrição do ponto de partida - o estado inicial
    - Exemplo:
        - A situação no jogo de xadrez
    - Um conjunto de transições de estados
    - Uma função que diz se um dado estado satisfaz o objetivo
    - or 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 objetivo.

Aplicação: determinar um percurso num mapa topológico

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



Estratégias de pesquisa

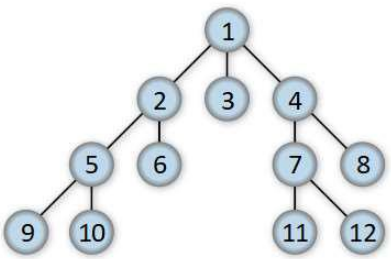
- Pesquisa em árvore
    - Estratégias de pesquisa cega (não informada):
        - Em largura
        - Em profundidade
        - Em profundidade com limite
        - Em profundidade com limite crescente
    - Estratégias de pesquisa informada
        - Pesquisa A* e suas variantes (custo uniforme, gulosa)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

Pesquisa em árvore - algoritmo genérico

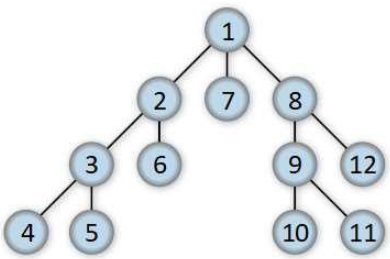
pesquisa(Problema,Estratégia) retorna a Solução, ou 'falhou'

Árvore ← árvore de pesquisa inicializada com estado inicial do Problema
Ciclo:
se não há candidatos para expansão, retornar 'falhou'
Folha ← uma folha escolhida de acordo com Estratégia
se Folha contém um estado que satisfaz o objetivo
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

Pesquisa em árvore - implementação baseada numa fila

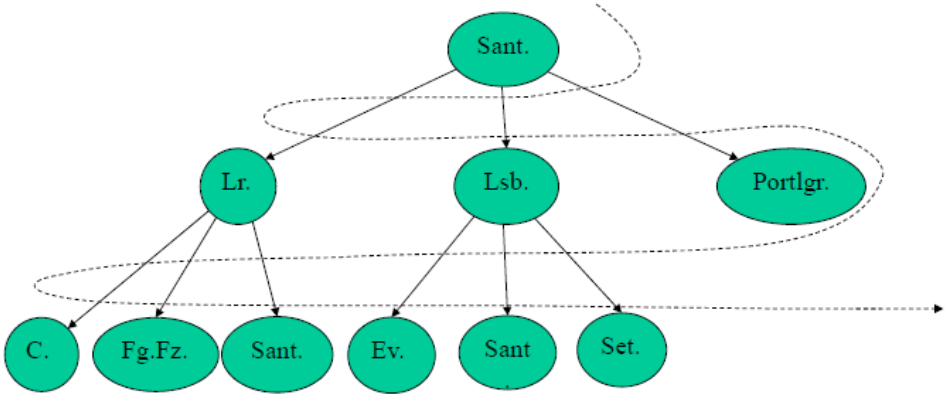
pesquisa_em_arvore(Problema,AdicionarFila) retorna a Solução, ou 'falhou'

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

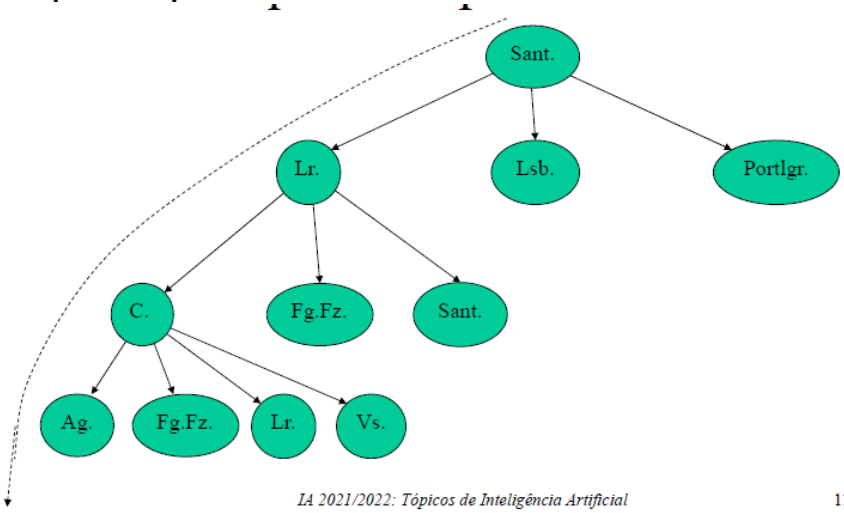
pesquisa_em_largura(Problema) retorna a Solução, ou 'falhou'
retornar pesquisa_em_arvore(Problema,juntar_no_fim)

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

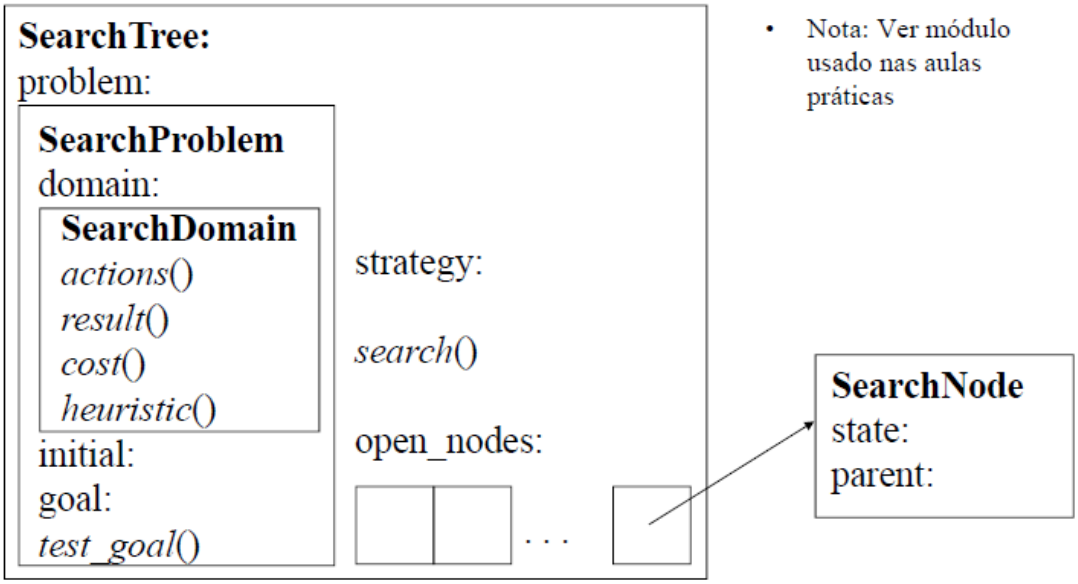
- Vamos criar um conjunto de classes para suporte à resolução de problemas por pesquisa em árvore

- Classe SearchDomain() - classe abstrata que formata a estrutura de um domínio de aplicação

- Classe SearchProblem(domain,initial,goal) - classe para especificação de problemas concretos a resolver

- Classe SearchNote(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 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 atual 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 é igual a 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,Estrategia)

Avaliação das estratégias de pesquisa

- Completude - uma estratégia é completa se é capaz de encontrar uma solução quando existe um solução

- Complexidade temporal - quanto tempo demora a encontrar a solução

- Complexidade espacial - quanto espaço de memória é necessário para encontrar a 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 h(n) é admissível, então a pesquisa A* encontra-se sempre (um)a solução ótima
- A pesquisa A* é também completa

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 a solução, a primeira solução encontrada é ótima
- Pesquisa gulosa
    - Ignora custo acumulado g(n)
    - f(n) = h(n)
    - Dado que o custo acumulado é ignorado, não é verdadeiramente um caso particular da pesquisa A*
    - Tem um comportamento que se aproxima da pesquisa em profundidade
    - Ao ignorar o custo acumulado, facilmente deixa escapar a solução ótima

Pesquisa num grafo de estados - motivação
- 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 dique 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 selecionado 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 objetivo, então retornar a solução encontrada.
5. Expandir N :
– CV ← conjunto dos vizinhos sucessores de N
– Para cada X ∈ CV - (ABERTOS ∪ FECHADOS), ligá-lo ao antecessor direto, N
– Para cada X ∈ CV ∩ (ABERTOS ∪ 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 e 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 um árvore
- Pelo contrário, a pesquisa em grafo leva em conta que o espaço de estados é normalmente um grado e garante que a árvore de pesquisa não tem mais do que um caminho para cada estado

Avaliação da pesquisa em árvore - fatores de ramificação

- Seja:
    - N - número de nós de árvore de pesquisa no momentos 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
- Fator de ramificação efetivo - número de filhos por nó, N, numa árvore com ramificação constante e com profundidade constante d. Portanto: $1 + B + B^2 + \cdots + B^d = N$ ou seja: $\frac{B^{d+1}-1}{B-1} = N$ (resolve-se por métodos numéricos).
    - O fator de ramificação efetiva é um indicador da eficiência da técnica de pesquisa utilizada

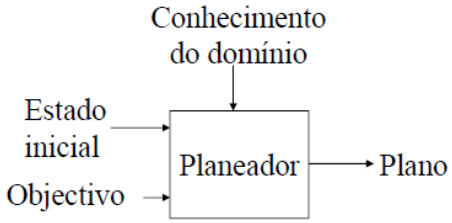
Aplicação: planear um passeio turístico

- 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 ações

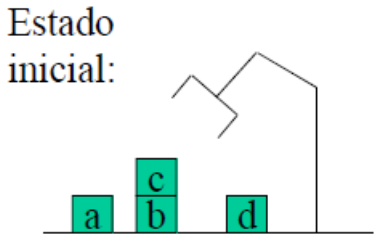
- O problema consiste em determinar uma sequência de ações a desempenhar por um agente por forma a que. partindo de um estado inicial, se atinja um dado objetivo
- O conhecimento do domínio inclui uma descrição das condições de aplicabilidade e efeitos das ações possíveis

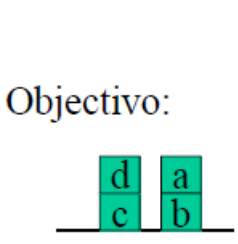


Representação de ações em problemas de planeamento

- 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





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 fator de ramificação efetiva
    - Quando melhor a heurística, mais baixo será esse fator
- Em alguns domínios, há funções de estimação de custos que naturalmente constituem heurísticas admissíveis
    - Exemplo: Distância em linha reta 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 o combinando-as numa nova heurística
- IMPORTANTE: O custo de uma solução ótima para um problema simplificado constitui um heurística admissível para o problema original

Pesquisa A* - combinação de heurísticas

- Se tivermos várias heurísticas admissíveis (h1, …,hn), podemos combiná-las numa nova heurística:
    - H(n) = max({h1(n), …,hn(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
    - Ótima
- 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 portanto a optimalidade da pesquisa

IDA*

- Semelhante à pesquisa em profundidade com aprofundamento iterativo
    - A limitação à profundidade é estabelecida indiretamente através de um limite na função de avaliação f(n) = g(n)+h(n) <= fmax

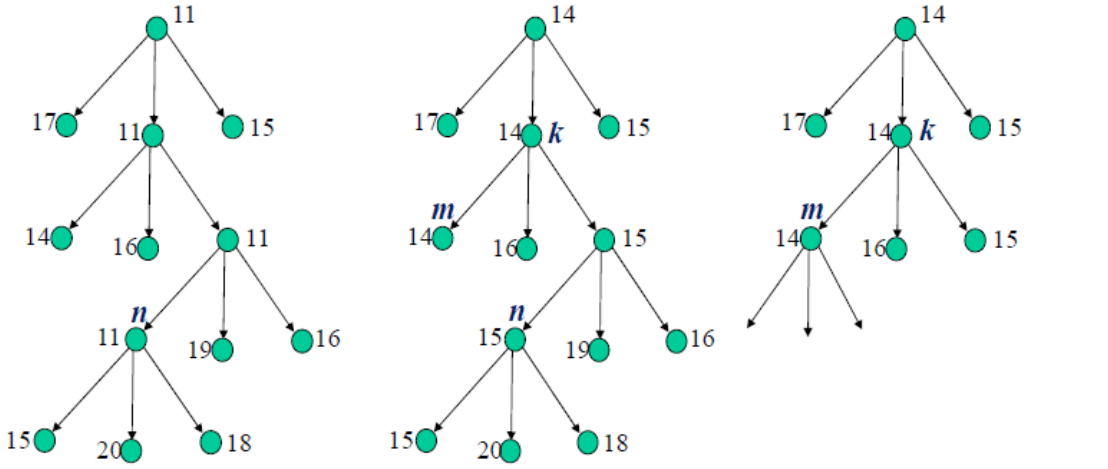
- Ou seja: Qualquer nó n com f(n)> fmax não será expandido

- Passos do algoritmo:
    1. fmax = f(raiz)
    2. Executar pesquisa em profundidade com limite fmax
    3. Se encontrou solução, retornar solução encontrada
    4. fmax ← menor f(n) que tenha sido superior a fmax 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 atualizados
- 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*

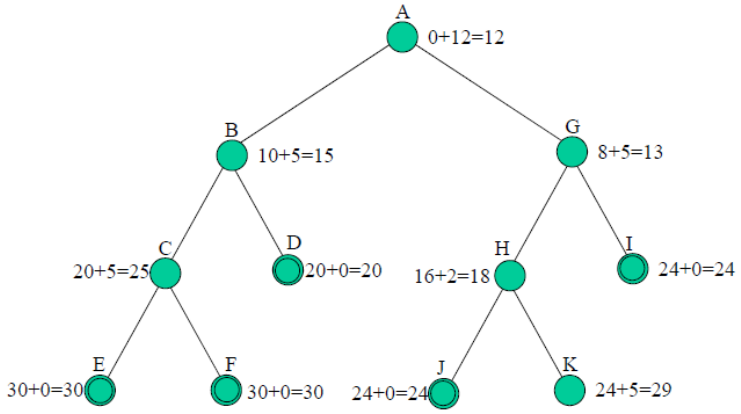
SMA*

- 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), atualizando 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 atualizado como no RBFS

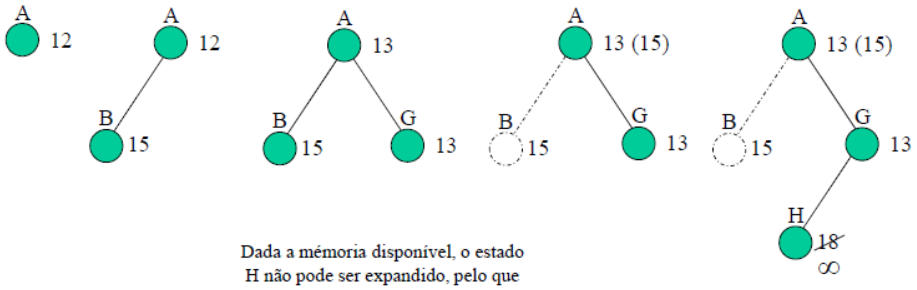
SMA* - exemplo - espaço de estados



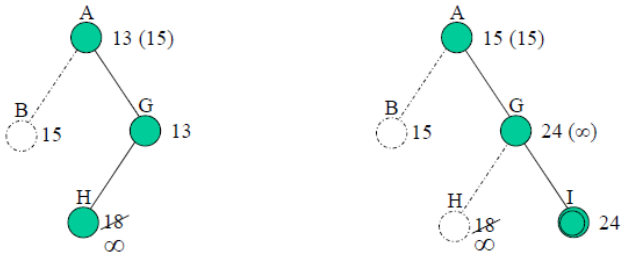
SMA* - exemplo

- 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



- 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