

Inteligência Artificial

José Mendes 107188

2023/2024



universidade
de aveiro

1 Noções de Programação Declarativa

1.1 Programação Declarativa

A Programação Declarativa abstrai-se da implementação, focando-se apenas na descrição do que se pretende fazer, enquanto faz uso de dois paradigmas:

- **Programação Funcional**, baseado em funções/cálculo-lambda, a entidade central é a função;
- **Programação em Lógica**, baseado em lógica de primeira ordem, a entidade central é o predicado;

1.2 Paradigma Imperativo

O fluxo de operações é especialmente sequenciado de operações com foco na forma como as tarefas são executadas (**instrução**). Podemos **alterar o conteúdo em memória** e ainda (instruções de afetação/atribuição) e ainda **realizar análise de casos** (if-then-else, switch/case, ...), **processamento iterativo** (while, repeat, for, ...) e ter associados **sub-programas** (procedimentos, funções).

Exemplo: SQL é uma linguagem declarativa, uma vez que nos seus comandos apenas descrevemos o que queremos obter. Assim, o programador é abstraído da forma como as operações são executadas na prática, ficando essa tarefa a cargo do compilador.

1.3 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)

A sua origem data da segunda metade do século XX, mas ainda hoje é amplamente usada e até está em crescimento em áreas como a Inteligência Artificial.

1.4 Programação Funcional

Possibilidade de definir funções localmente e sem nome. Por exemplo as funções lambda presentes em Lisp e Python.

1.5 Programação em Lógica

Um programa é uma teoria sobre um domínio. Por exemplo, temos **socrates é um homem**, $homem(socrates)$ e, um **homem é mortal**, $homem(X) : \neg mortal(X)$. Se perguntarmos se **socrates é mortal**, $mortal(socrates)$, a resposta é sim.

1.6 Atitude do programador

A programação declarativa, dada a sua elevada expressividade, é pouco compatível com aproximações empíricas (ou seja, tentativa e erro) à programação. Primeiro é preciso pensar bem na estrutura do problema antes de começar a teclar.

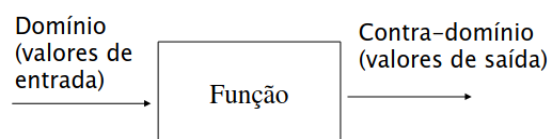
Passos a seguir: Perceber o problema \rightarrow Desenhar o programa \rightarrow Escrevê-lo \rightarrow Rever e testar

1.7 Características da Programação Funcional

- A entidade central é a função;
- A noção de função é diretamente herdada da matemática (ao contrário das linguagens imperativas, que pode ser muito diferente);
- 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 saída);
- Os elementos do domínio de entrada e saída podem ser funções;

1.8 Função

Tem valores de entrada (domínio) e valores de saída (contradomínio).



1.9 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 do problema/domínio de aplicação) e **regras** (leis gerais sobre esse problema/domínio). Implicitamente as frases estão reunidas numa grande conjunção, e cada frase está universalmente quantificada. Portanto, **programação declarativa**.

2 Programação ao estilo funcional em Python

2.1 Python

Criada no final dos anos 90, é uma linguagem de programação interpretada, iterativa, portátil, funcional, orientada a objetos e de implementação aberta. Tem como principais objetivos: simplicidade sem prejuízo da utilidade, programação modular, legibilidade, desenvolvimento rápido, facilidade de integração, nomeadamente com outras linguagens. Apresenta-se como uma linguagem **multi-paradigma**:

- **Programação funcional:** Expressões lambda, funções de ordem superior, listas com sintaxe simples, iteradores, ...
- **Programação OO:** Classes, objetos, métodos, herança, ...
- **Programação imperativa/modular:** Atribuição, sequências, condicionais, ciclos, ...

2.1.1 Python vs Java

Comparativamente ao JAVA revela-se menos consisa porque os espaços são sintaticamente relevantes e o código, não compilado para código nativo (interpretado), demora mais tempo a executar (diferença residual), no entanto apresenta uma maior facilidade de desenvolvimento e legibilidade.

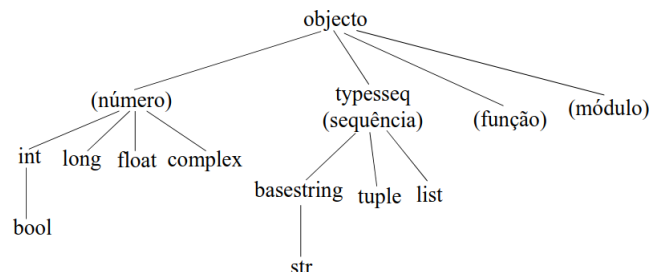
2.1.2 Áreas de aplicação

As principais áreas de aplicação são: interligação de sistemas, aplicações gráficas, aplicações para bases de dados, multimédia, internet protocol/web e robótica e inteligência artificial.

2.2 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 uma identidade/referência (identifica a posição de memória onde está armazenado), um tipo e um valor. Alguns tipos de objetos podem ter atributos e métodos, outros (classes) podem ter sub-tipos (sub-classes).



Alguns destes são imutáveis, como as **str** e **tuple**. Para determinar o tipo de um objeto utiliza-se a função pré-definida **type()**.

2.3 Variáveis

As variáveis guardam objetos, mas não são declaradas (inicializadas apenas) nem têm tipos associados (o tipo é do objeto!). Praticamente tudo pode ser atribuído a uma variável, incluindo funções, módulos e classes. Similar as linguagens imperativas, e ao contrário das funcionais, o valor de uma variável pode ser alterado. Não se pode ler o valor de uma variável se este não tiver sido previamente inicializado.

2.4 Instrução de atribuição

Tal como é habitual na programação imperativa, e ao contrário do habitual na programação declarativa, Python possui instrução de atribuição.

Exemplo:

```
n = 10
a = b = c = 0
x = 7.25
cad = "cadeia"
t = (n, x)
l = [1, 2, 'quatro', 5.0]
```

Pode user a operação de atribuição para decompor estruturas:

```
triplo = (1, 2, 3)
(i, j, k) = triplo
```

2.5 Operadores

Operadores comuns matemáticos e lógicos. É importante saber que objetos de tipos diferentes nunca são iguais (exceto os diferentes tipos de números).

2.5.1 Operadores lógicos

Na conjunção e disjunção, o segundo elemento só é avaliado se for necessário para determinar o resultado.

```
if False and a: # a nao e avaliado
    ...

if True or a: # a nao e avaliado
    ...
```

2.6 Acesso a sequências

Os elementos das sequências são acedidos através de índices inteiros consecutivos (primeiro elemento tem índice 0).

É possível extrair “fatias” das sequências. A indexação é circular, pelo que podemos aceder a índices negativos.

```
# [inf:sup] retorna a sequencia entre os indices inf e sup-1 (uma opia da lista)

# Para fazer uma copia integral da lista usa-se [:]
```

Detalhe importante: instrução de atribuição, em vez de copiar valores, limita-se a associar um dado identificador a um dado objecto. Desta forma, a atribuição da variável x a uma variável y, apenas tem como resultado associar y ao mesmo objecto que x já estava associada. No caso dos objetos mütaveis, temos de ter cuidado:

```
a = [1, 2, 3]
b = a
b[1:2] = []
print(a) # [1, 3]
```

2.7 Definição de funções

Funções sem return, retornam None, são também conhecidas como procedimentos. As funções podem ser recursivas e são objetos do tipo **function**.

Os parâmetros são passados às funções segundo um mecanismo “passagem por valor” (“call by value”). Os parâmetros são passados por referência, não cópias!

Se atribuírmos um novo objeto a uma variável passada por parâmetro, essa atribuição ocorre apenas no espaço de nomes da função (imagem à esquerda). Se modificarmos um objeto passado por parâmetro (por exemplo, apagar um elemento de uma lista), isto não altera a referência do objeto, e portanto vai permanecer após o retorno da função (imagem à direita).

<pre>>>> def incr(x): ... x=x+1 ... return x ... >>> n=10 >>> incr(n) 11 >>> n 10</pre>	<pre>>>> def acresc(l,x): ... l[0:0]=[x] ... return l ... >>> lista=[5,12] >>> acresc(lista,30) [30,5,12] >>> lista [30,5,12]</pre>
---	---

O problema anterior resolve-se trabalhando sobre uma cópia local, no caso de uma lista l, podemos usar l[:], que dá uma cópia integral.

Podemos passar parâmetros com valores por defeito, podendo se, na chamada, omitir alguns parâmetros.

```
def abc(arg1, arg2=True, arg3=None):
    ...
# Quando so queremos passar um dos args por defeito devemos indicar o seu nome
abc(1, arg3=3)
```

2.8 Funções Recursivas

```
# devolve factorial de um numero 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:])
```

Esta última função, vai funcionar da seguinte forma:

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

```
# verifica se um elemento e 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
```

2.9 Expressões lambda

São **expressões cujo valor é uma função**. Funções que recebem expressões lambda como entrada e as retornam como saída chamam-se funções de ordem superior. Exemplo:

```
lambda x: x*x

# Pode ser atribuida a uma variavel
m = lambda x,y: math.sqrt(x**2 + y**2)
m(1, 2) # 2.23606797749979
```

Como qualquer objeto, uma expressão lambda pode ser passada como:

```
# Tendo uma func h que produz f(x)*X
def h(f, x): return f(x)*x

# Para usar:
h(lambda x: x+1, 7) # 56
```

Podem ser produzidas por outras funções:

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

# Para usar:
f = faz_incrementador(1)
f(10) # 11
```

As expressões lambda são também conhecidas como expressões funcionais. As funções que recebem estas expressões como entrada e/ou produzem expressões lambda como saída chamam-se 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.

2.10 Aplicar uma função a uma lista

Para tal podemos usar a função **map()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

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

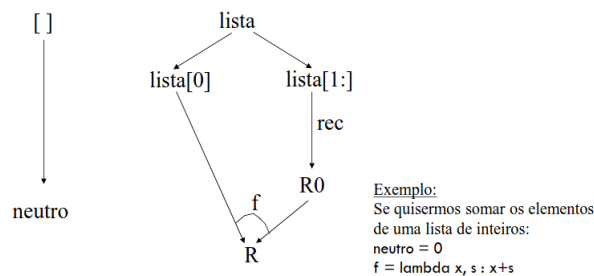
2.11 Filtrar uma lista

Para tal podemos usar a função **filter()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

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

2.12 Reduzir uma lista a um valor

Muitos dos procedimentos aplicados a listas consistem em combinar a cabeça da lista com o resultado da chamada recursiva sobre os restantes elementos, retornando um valor “neutro” pré-definido para a lista vazia. Em Python, podemos usar a função **reduce()** da biblioteca **functools**.



```
# Dada uma lista, uma func de combinacao e um elemento neutro
def reduzir(lista, fcomb, neutro):
    if lista==[]:
        return neutro
    return fcomb(lista[0], reduzir(lista[1:], fcomb, neutro))
```

2.13 Listas por compreensão (list comprehension)

São mecanismos compactos para processar os elementos de uma lista (alguns ou todos), podendo ser aplicadas não só a estas como também a tuplos ou cadeias de caracteres (str), o resultado é uma lista. Podem funcionar com a função pré-definida **map()** ou **filter()**.

$$< \textit{expr} > \textbf{for} < \textit{var} > \textbf{in} < \textit{seq} > \textbf{if} < \textit{condition} >$$

```
[x**2 for x in [2, 3, 4]] # Output: [4, 9, 16]
map(lambda x: x**2, [2, 3, 4]) # Igual

[x for x in [2, 3, 4] if x%2==0] # Output: [2, 4]
filter(lambda x: x%2==0, [2, 3, 4]) # Igual
```

Dada uma lista de listas e uma função, podemos aplicar uma função a cada um dos elementos das listas, retornando a concatenação das listas resultantes.

```
[f(y) for x in lista for y in x]

# Exemplo:
lista = [[1, 2, 3], [4, 5, 6]]
[x+1 for x in lista for y in x] # [2, 3, 4, 5, 6, 7]
```

2.14 Classes

Tal como nas restantes linguagens Orientadas a Objetos (OO), em Python uma classe define um conjunto de objetos caracterizados por diversos atributos e métodos e organizam-se numa hierarquia que permite a herança.

```
# sintaxe
class <nome-classe>:
    <decl-1>
    ...
    <decl-n>
```

Nota: Tal como acontece com as variáveis normais, também os atributos das classes não são declarados. Os atributos são acessados através da notação "objeto.atributo". Podemos, em qualquer momento, criar um novo atributo para um objeto, bastando para isso atribuir-lhe um valor.

Existe ainda classes derivadas/herança, que herdam os métodos e atributos da classe mãe, sendo possível uma classe ter várias classes mães.

2.14.1 Métodos e atributos pré-definidos

Métodos:

- `__init__` - Construtor da classe;
- `__str__` - Conversão para cadeia de caracteres;
- `__repr__` - Representação em cadeia de caracteres que aparece na consola do interpretador;

Atributos:

- `__class__` - identifica a classe de um dado objecto

2.14.2 list de Python é uma classe

Tem os seguintes métodos:

- `append(x)` - Adiciona x ao fim da lista;
- `extend(L)` - Adiciona os elementos da lista L ao fim da lista;
- `insert(i, x)` - Adiciona x na posição i da lista;
- `remove(x)` - Remove a primeira ocorrência de x na lista;
- `index(x)` - Devolve o índice da primeira ocorrência de x na lista;
- `sort()` - Ordena a lista (modifica a lista);

3 Tópicos de Inteligência Artificial

3.1 Definição de "Inteligência"

Segundo a sua definição lexical, **inteligência** é a capacidade de pensar, raciocionar, adquirir e aplicar conhecimento. É o conjunto de capacidades superiores da mente.

O seu estudo envolve várias vertentes, desde a já referida aquisição, representação e armazenamento de conhecimento, a geração de comportamento inteligente, a origem das motivações, emoções e prioridades, como as percepções dão origem a entidades simbólicas, como raciocionar sobre o passado, planear o futuro, como surge a ilusão, a crença, esperança, amor ...

3.2 Definição de "Inteligência Artificial"

É a disciplina que estuda as teorias e técnicas necessárias ao desenvolvimento de "artefactos" inteligentes. Historicamente, estas vertentes resumem-se a quatro grandes abordagens, definidas por Russel & Norvig (1995).

Pensar como o ser humano	Pensar racionalmente
Agir como o ser humano	Agir racionalmente

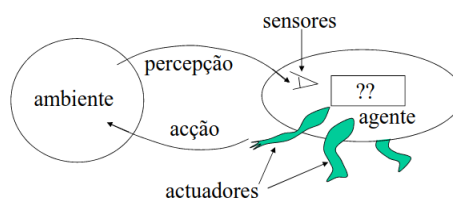
As **humanas** focam-se no ser humano, desenvolvendo uma ciência empírica, baseada na hipótese e na experiência. Por outro lado, a **abordagem racional** envolve uma combinação de matemática e engenharia.

Nota: No início estava muito focada na imitação das capacidades do ser humano, no entanto, mais recentemente tem-se voltado mais para a racionalidade como referência. Atualmente é maioritariamente utilizada para a resolução de problemas concretos e pouco abrangentes, focando-se na melhor utilização dos recursos de forma racional.

3.3 Definição de "Agente"

Segundo o dicionário, pode ser uma entidade com poder ou autoridade para agir, ou uma entidade que atua em representação de outrem.

Agente - Definido como uma entidade com capacidade de obter informação sobre o seu ambiente (sensores) e de executar (atuadores) ações em função dessa informação por Russel e Norvig.



3.3.1 Exemplo de agentes

Tipo de agente	Percepção	Ação	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 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	Propor exercícios, corrigi-los, dar sugestões	Maximizar o resultado dos alunos num teste	Conjunto de alunos

3.4 Agir como o ser humano - Teste de Turing

O **teste de Turing** (1950) pretendeu estabelecer a definição operacional para o comportamento inteligente de nível humano, que consistia em submeter o artefacto a um interrogatório através de um terminal de texto. Se o humano não conseguir concluir que está a interrogar uma máquina, então conclui-se que o artefacto é inteligente.

3.5 A "sala chinesa" de Searle

Esta teoria foi refutada por **Searle**, um filósofo americano, com o argumento da **Sala Chinesa**. Neste, descreveu o cenário de colocar um humano que não percebe chinês fechado dentro de uma sala com um livro escrito na sua língua que ensine chinês a ser interrogado por outro humano através de papéis passados por baixo da porta.

Searle defendeu que apesar de não perceber os papéis entregues por baixo da porta, o humano iria conseguir traduzi-los através do livro e elaborar respostas. Fora da sala, o interrogador iria ter a ilusão de que o interrogado sabia chinês. No entanto, nem o humano, nem a sala, nem o livro percebem chinês. Logo, deduziu que não há qualquer compreensão de chinês naquela sala. No entanto, podemos contra-argumentar que, embora individualmente, os componentes do sistema não percebam chinês (a sala, o humano, o livro, as folhas de papel), não compreendam chinês, **o sistema no seu conjunto compreende chinês**.

Foi esta a premissa para a sua argumentação de que os computadores apenas fazem uso de regras sintáticas para manipular o texto, sem ter qualquer entendimento do seu significado ou semântica. Para ele a inteligência é um processo biológico, que apenas pode ser simulado pelas máquinas, mas nunca adquirido.

3.6 Tipos e arquiteturas de agentes

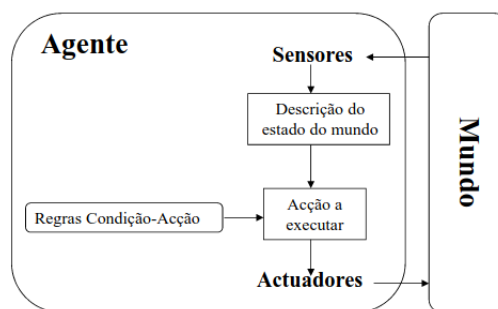
Tipos de agentes:

- Reativo simples;
- Reativo com estado;
- Deliberativos orientados a objetivos;
- Deliberativos orientados por funções de utilidades;

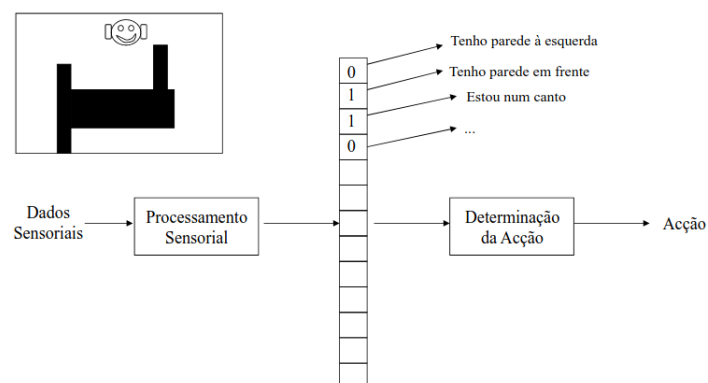
Arquiteturas de agentes:

- Subsunção;
- Três torres;
- Três camadas;
- CARL;

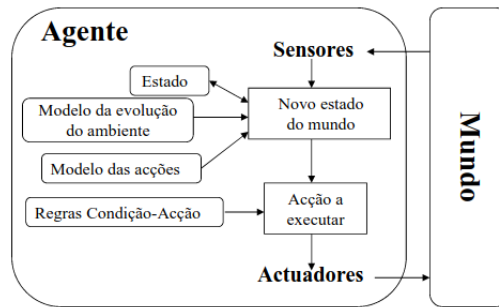
3.6.1 Agentes reativos: simples



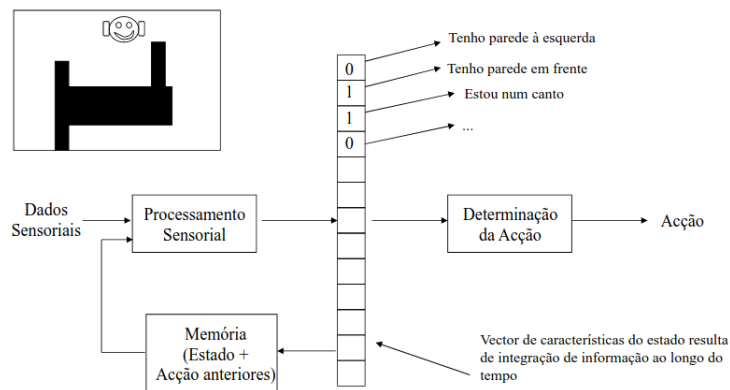
Também conhecidos por agentes de **estímulo-resposta** (ou "sistemas de produção"), este tipo de agente apresenta um conjunto de sensores, através dos quais recebe uma percepção do estado do mundo, sobre a qual aplica um conjunto de regras (regras de condição-ação, também conhecido como "regras de situação-ação" ou "regras de produção") e executa as ações correspondentes.



3.6.2 Agentes reativos: com estado interno

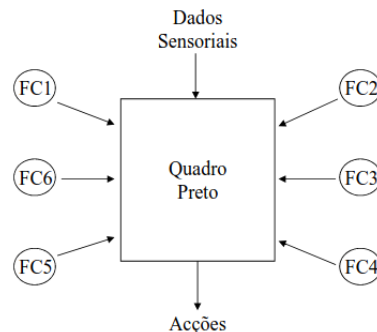


Funcionam de forma semelhante ao anterior, mas para além dos sensores, fazem uso de um **estado interno** e do histórico de ações anteriores para construir a percepção do estado do mundo.



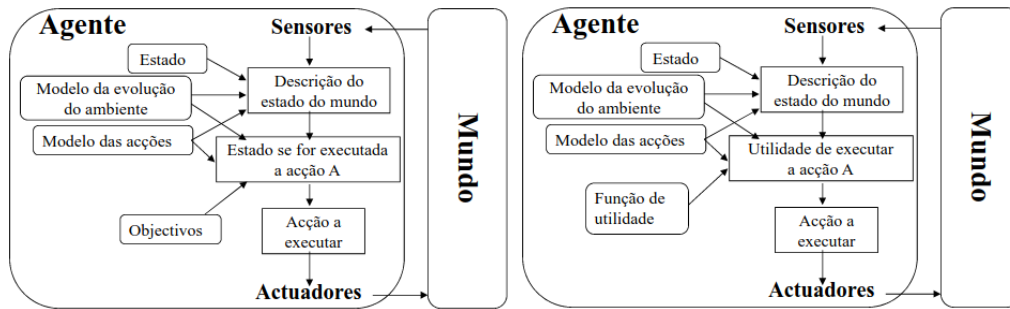
Sistemas de Quadro Preto

Podem ser vistos como uma elaboração dos sistemas reactivo 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-ação.



3.6.3 Agentes deliberativos

Contrariamente aos anteriores, executam as ações com base em **objetivos** ou **função de utilidade**.



Primeira figura: Agente deliberativo orientado a objetivos.

Segunda figura: Agente deliberativo orientado por funções de utilidade.

Nota: A função de atividade é uma função que avalia a relevância de cada estado para o agente.

3.7 Propriedades do mundo de um agente

Visto como se define um agente e os seus tipos, falta perceber como se caracteriza o seu ambiente. Este pode apresentar várias propriedades.

- **Acessibilidade** - O mundo é **acessível** se os sensores do agente tem acesso a toda a informação sobre o ambiente (descrição completa), ou pelo menos a toda a informação relevante. para o processo de escolha das ações (**efetivamente acessível**).
- **Determinismo** - Se o estado resultante da execução de uma ação é apenas e totalmente determinado pelo estado atual e pelos efeitos esperados da ação.
- **Mundo episódico** - Cada episódio de percepção-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 contínuo ou sem saltos; caso contrário o mundo diz-se **discreto**.

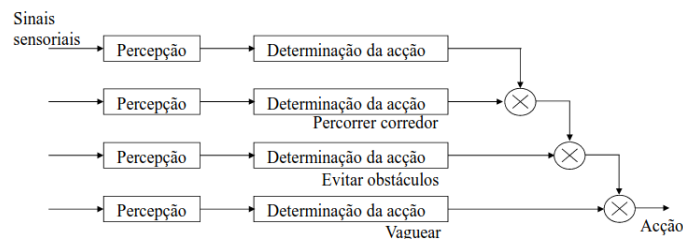
Exemplo: Um agente táxi opera num ambiente dinâmico, enquanto que um agente que jogue palavras-cruzadas opera num ambiente estático. Para um agente carro de condução autónoma, a estrada é um ambiente contínuo. Já para um agente que jogue xadrez, o tabuleiro é um ambiente discreto.

3.7.1 Mundo de um agente: Exemplos

Mundo	Acessível	Determinístico	Episódico	Dinâmico	Contínuo
Xadrês s/ relógio	Sim	Sim	Não	Não	Não
Xadrês c/ relógio	Sim	Sim	Não	Semi	Não
Poker	Não	Não	Não	Não	Não
Condução de carro	Não	Não	Não	Sim	Sim
Diagnóstico médico	Não	Não	Não	Sim	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

3.8 Arquiteturas de agentes

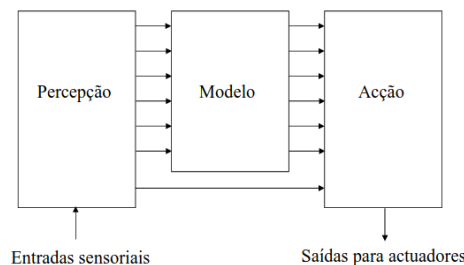
3.8.1 Subsunção



Esta arquitetura procura estabelecer a ligação entre a percepção e a ação a vários níveis, organizados em camadas, criando agentes com comportamento simultaneamente reativo e deliberativo.

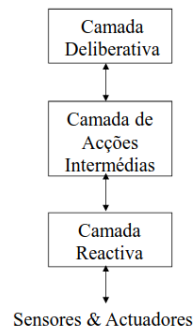
A camada mais baixa é a mais reativa, diminuindo a reatividade e aumentando o peso da componente deliberativa à medida que se sobe nas camadas. A ação do agente resulta da fusão das decisões das várias camadas

3.8.2 Três torres



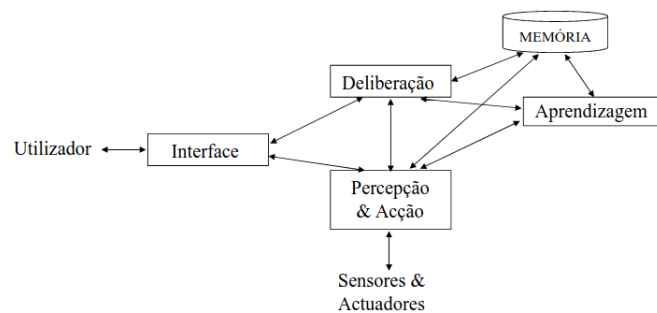
O processo de **tomada de decisão é simbólico** e feito a partir de um modelo, mas tem **algumas ações reativas**.

3.8.3 Três camadas



Para além das camadas deliberativas e reativas tem uma **camada de acções intermédias**, que é o intermediário entre as duas, produzindo uma alteração qualitativa no estado do mundo.

3.8.4 CARL



Esta arquitetura (anos 2000) veio introduzir dois novos conceitos:

- **Interfaces** - Facilitam a interação por pessoas não especializadas;
- **Aprendizagem** - Para funcionar num ambiente humano não estruturado vai ter de aprender para se ir adaptando às condições do mundo real. Está assente em **memória**.

4 Representação do conhecimento

Para poderem tomar decisões, os agentes (especialmente os deliberativos) precisam de ter algum conhecimento sobre o mundo em que se inserem.

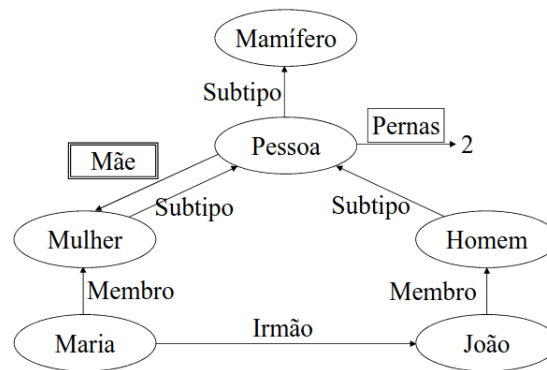
4.1 Redes Semânticas

As **redes semânticas** são representações gráficas do conhecimento que facilitam a legibilidade. As redes semânticas podem ser tão expressivas quanto a lógica de primeira ordem.

4.1.1 Tipos de Relções

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

4.1.2 Exemplo



4.1.3 Herança

A herança pode ser representada em relações de **sub-tipo**, herdando todas as propriedades dos tipos mais abstratos dos quais descende, ou de **instância**, herdando as todas as propriedades do tipo a que pertence. Aplica-se um **raciocínio não monotónico** devido ao estabelecimento dos **valores por defeito**, propriedades comuns a todos os membros desse tipo e do **cancelamento de herança**, que ocorre quando um sub-tipo altera um valor por defeito herdado de um tipo do qual descende.

4.1.4 Métodos e Demónios

Normalmente, por razões computacionais, usam-se redes semânticas bastante menos expressivas do que a lógica de primeira ordem, deixando-se de lado a **negação**, a **disjunção** e a **quantificação**.

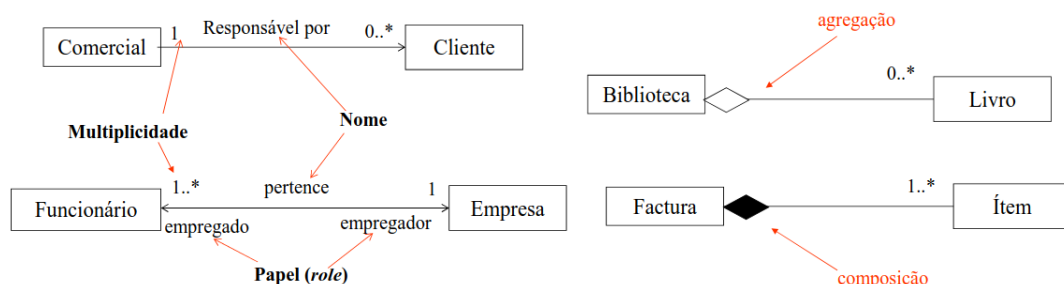
Em contra-partida, nomeadamente nos chamados sistemas de frames, usam-se métodos e demónios:

- **Métodos** - têm uma semântica similar à da programação orientada por objectos.
- **Demónios** - são procedimentos cuja execução é disparada automaticamente quando certas operações de leitura ou escrita são efetuadas.

4.2 UML / Diagramas de Classes

Na linguagem gráfica UML (Unified Modelling Language), os diagramas de classes definem as relações existentes entre as diferentes classes de objetos num dado domínio.

- **Classe** - Descrição de um conjunto de objetos que partilham os mesmos atributos, operações, relações e semântica; estes objetos podem ser físicos ou conceituais.
- **Atributo** - Uma propriedade de uma classe.
- **Operação (ou método)** - implementação de um serviço.
- **Instância** - Um objeto que pertence a uma classe.
 - 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)



4.3 Redes semânticas vs UML

<i>Redes semânticas</i>	<i>UML</i>
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

4.4 Indução versus Dedução

A **dedução** permite a partir de um conjunto de permissas consideradas verdadeiras, por mecanismos de inferências lógicas, tirar conclusões verdadeiras.

No entanto, para adquirir conhecimento novo é necessário haver **indução**, ou seja, a capacidade de inferir regras gerais a partir de observações concretas.

A dedução permite inferir casos particulares a partir de regras gerais, enquanto que a indução permite inferir regras gerais a partir de casos particulares.

4.4.1 Exemplo de Indução

Casos conhecidos:

- O gato Tareco gosta de leite;
- O gato Ozzy 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".

4.5 Lógicas

Uma lógica tem:

- **Sintaxe** - descreve o conjunto de frases ou fórmulas que é possível escrever (WFF - Well Formed Formulas);
- **Semântica** - estabelece a relação entre as frases escritas nessa linguagem e os factos que representam;
- **Regras de Inferência** - permitem manipular as frases, gerando umas a partir das outras; as regras de inferência são a base do processo de raciocínio.

4.5.1 Lógica Proposicional

A **lógica proposicional** é baseada em proposições, frases declarativas que podem ser **verdadeiras ou falsas**, que podem ser representadas por variáveis proposicionais que tomam o seu valor de verdade. As fórmulas são compostas por uma ou mais variáveis proposicionais ligadas por conectivas lógicas.

Exemplo: As proposições não dependem da interpretação, uma vez que, contrariamente aos predicados, as proposições não têm argumentos. "A neve é branca." é uma proposição, que é verdadeira.

4.5.2 Lógica de Primeira Ordem

Em contraste temos a **lógica de primeira ordem**, onde há uma distinção entre os objetos e as relações (predicados).

Além de variáveis, usadas para representar termos não especificados e constantes escalares, os objetos podem ainda tomar a forma de expressões funcionais, que se formam pela aplicação de uma função aos seus argumentos. Os objetos podem ser considerados como expressões funcionais cujo número de argumentos é zero.

O nome da relação aplica-se sempre ao seu primeiro elemento (argumento). Os seus argumentos só podem ser termos (nunca relação de relação).

Exemplo: $\text{Potencia}(4, 3)$ é uma **expressão funcional**.
 $\text{Pai}(\text{Rui}, \text{João})$ é um **predicado**.

4.6 Conectivas Lógicas (LP, LPO)

Comuns às duas lógicas abordadas, servem para combinar frases lógicas elementares por forma a obter frases mais complexas.

- **Conjunção** - $p \wedge q$
- **Disjunção** - $p \vee q$
- **Implicação** - $p \rightarrow q$
- **Negação** - $\neg p$

4.7 Variáveis, Quantificadores (LPO)

Para quantificar as variáveis da lógica de primeira ordem temos os quantificadores universais e os quantificadores existenciais.

- **Quantificador universal** - $\forall x$.
 - $\forall x A \equiv$ "Qualquer que seja x , a fórmula A é verdadeira"
- **Quantificador existencial** - $\exists x$
 - $\exists x A \equiv$ "Existe um x , para o qual a fórmula A é verdadeira"

Se A é uma fórmula bem formada (FBF), então a sua quantificação também é uma FBF.

4.8 Gramáticas (LPO)

$Fórmula \rightarrow FórmulaAtômica$
| $Fórmula\ Conectiva\ Formula$
| $Quantificador\ Variável, \dots Fórmula$
| $'\neg' Fórmula$
| $'(' Fórmula \)'$
 $FórmulaAtômica \rightarrow Predicado\ '('\ Termo\ ',' \dots \)'$ | $Termo\ '='\ Termo$
 $Termo \rightarrow Função\ '('\ Termo\ ',' \dots \)'$ | $Constante$ | $Variável$
 $Conectiva \rightarrow '\Rightarrow' \mid '\wedge' \mid '\vee' \mid '\Leftrightarrow'$
 $Quantificador \rightarrow '\exists' \mid '\forall'$
 $Constante \rightarrow A \mid X1 \mid Paula \mid \dots$
 $Variável \rightarrow a \mid x \mid s \mid \dots$
 $Predicado \rightarrow Portista \mid Cor \mid \dots$
 $Função \rightarrow Registo \mid Mãe \mid \dots$

4.8.1 Exemplo

- “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”
 - $\exists x\ \forall y\ Gosta(x, y)$

4.9 Interpretações em Lógica Proposicional

Na lógica proposicional, a interpretação de uma fórmula é a atribuição de valores de verdade ou falsidade às várias proposições que nela ocorrem. $A \wedge B$ tem quatro interpretações possíveis que resultam da combinação dos valores de verdade e falsidade de A e B.

- **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.

4.10 Interpretações em Lógica de Primeira Ordem

Na **lógica de primeira ordem**, a interpretação de uma fórmula é o estabelecimento de uma correspondência entre as constantes que ocorrem na fórmula e os objetos do mundo, funções e relações que essas constantes representam.

4.11 Regras de Substituição

Estas regras permitem substituir uma fórmula por outra equivalente. São válidas tanto para a lógica proposicional como para a lógica de primeira ordem.

- | | |
|--|---|
| • Leis de DeMorgan | • Comutação |
| $\neg(A \wedge B) \equiv \neg A \vee \neg B$ | $A \wedge B \equiv B \wedge A$ |
| $\neg(A \vee B) \equiv \neg A \wedge \neg B$ | $A \vee B \equiv B \vee A$ |
| • Dupla negação: | • Associação: |
| $\neg\neg A \equiv A$ | $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ |
| • Definição da implicação: | $(A \vee B) \vee C \equiv A \vee (B \vee C)$ |
| $A \Rightarrow B \equiv \neg A \vee B$ | • Distribuição: |
| • Transposição: | $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ |
| $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$ | $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ |

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

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

4.12 CNF e Forma Clausal

Uma fórmula na **forma normal conjuntiva** (CNF) é uma fórmula que consiste de uma conjunção de cláusulas.

Cláusula - fórmula que consiste de uma disjunção de literais.

Literal - fórmula atômica (literal positivo) ou a negação de uma fórmula atômica (literal negativo). Na LP uma fórmula atômica é uma proposição.

Forma Causal - representação de uma fórmula CNF através do conjunto das respectivas cláusulas.

Exemplo:

Literal - p ou $\neg p$

Cláusula - $I1 \vee I2 \vee \dots$

CNF - $c1 \ \& \ c2 \ \& \ \dots \text{CNF}$

Forma Causal - $[c1, c2, \dots]$

4.12.1 Conversão de uma Fórmula Proposicional para CNF e forma clausal

Através dos seguintes passos:

- Remover implicações
- Reduzir o âmbito de aplicação das negações
- Associar e distribuir até obter a forma CNF

Exemplo:

- Fórmula original: $A \Rightarrow (B \wedge C)$
- Após remoção de implicações: $\neg A \vee (B \wedge C)$
- Forma CNF: $(\neg A \vee B) \wedge (\neg A \vee C)$
- Forma clausal: $\{ \neg A \vee B, \neg A \vee C \}$

4.12.2 Conversão para forma clausal em Lógica de Primeira Ordem

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

Exemplo

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

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

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

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

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

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

IA 2023/2024: Técnicas de Inteligência Artificial

68

Exemplo (cont.)

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

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

Exemplo (cont.)

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

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

4.13 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:
 $\{ (A \Rightarrow B) \wedge (C \Rightarrow D), \neg B \vee \neg D \} \vdash \neg A \vee \neg C$

4.13.1 LPO - Regras de Inferência específicas

- *Instanciação universal:*
 $\{ \forall x P(x) \} \vdash P(A)$
- *Generalização existencial*
 $\{ P(A) \} \vdash \exists x P(x)$

4.14 Consequências lógicas e provas

A regras de inferência permitem-nos obter consequências lógicas (A) a partir de um conjunto de fórmulas ($\Delta = A_1, \dots, A_n$), representada esta noção por \models .

$$\Delta \models A$$

Esta consequência é obtida se A tomar o valor verdadeiro em **todas as interpretações** para as quais cada uma das fórmulas em Δ toma também o valor **verdadeiro**.

A **prova** de uma fórmula A_n é dada pela sequência de fórmulas $\Delta = \{A_1, \dots, A_n\}$ tal que cada A_i pode ser inferida a partir das fórmulas anteriores A_1, \dots, A_{i-1} . Neste caso, escreve-se:

$$\Delta \vdash A_n$$

4.15 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 que analisar caso a caso as várias interpretações.

4.16 Metateoremas

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

4.17 Resolução não é Completa

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

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

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

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

4.18 Refutação por Resolução

Este é um **mecanismo de inferência completo**, usa-se a resolução para provar que a negação da consequência lógica é inconsistente com a premissa (metateorema da redução ao absurdo). Para tal, basta derivar Falso.

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").

4.18.1 Passos

- Converter a premissa e a negação da consequência lógica para um conjunto de cláusulas.
- Aplicar a resolução até obter a cláusula vazia.

4.18.2 Substituições, Unificação

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

4.18.3 Resolução e Refutação na Lógica de Primeira Ordem

- Resolução:
 $\{ A \vee B, \neg C \vee D \} \vdash SUBST(A \vee D, g)$
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

4.19 Resolução com Cláusulas de Horn

A **refutação por resolução** é correcta e completa, mas apesar de melhor que avaliar todas as interpretações da inferência, continua a ser pouco eficiente.

Nota: Para várias cláusulas com muitos literais em cada uma vão haver múltiplos casos em que a resolução pode ser aplicada entre um literal e outros de outras formulas, o que vai gerar uma explosão de casos e um problema tendencialmente mais difícil de resolver com o aumento do seu tamanho.

Uma cláusula de Horn é uma cláusula que tem no máximo um literal positivo, como: A , $\neg A \vee B$, $\neg A \vee B \vee \neg C$ e $\neg A \vee \neg B$.

Existem algoritmos de dedução baseados em cláusulas de Horn cuja complexidade temporal é linear.

Exemplo: $\neg A \vee B \vee \neg C \quad (=) \quad A \wedge C \rightarrow B$

Nota: O problema da combinatoria da refutação por resolução vai ser minimizado, uma vez nas cláusulas de Horn cada cláusula vai ter menos possibilidades de resolução. O seu literal positivo pode ser resolvido com um negativo noutra fórmula e o inverso para os seus literais negativos.

Estas cláusulas permitem a aplicação de algoritmos com complexidade temporal linear.

5 Representação do conhecimento

5.1 Linguagem KIF (Knowledge Interchange Format)

É uma linguagem desenhada para representar o conhecimento trocado entre agentes. Além disso, pode ser usada para representar os modelos internos de cada agente. As características principais são: a **semântica puramente declarativa**, **tão ou mais expressiva que a LPO** e **permite a representação de meta-conhecimento**.

5.1.1 Características gerais

O mundo é concetualizado em termos de objetos e relações entre objetos (conjunto arbitrário de listas de objetos, como por 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**, podem ser **primitivos** (não decomponíveis) ou **compostos**.

5.1.2 Componentes da linguagem

- **Caracteres**
- **Lexemas**: especiais (papel pré-definidos na linguagem), palavras, código de caracteres, blocos de código de caracteres, cadeias de caracteres
- **Expressões**:
 - **Termos** - objetos primitivos ou compostos
 - **Frases** - expressões com valor lógico
 - **Definições** - frases verdadeiras por definição

Termos

Constantes, variáveis individuais, expressões funcionais, listas, termos lógicos, caracteres e citações.

Uma **expressão funcional** pode ser uma função ou uma estrutura de dados, ambas representadas por functor $\text{arg1}..n$.

As **listas** são uma expressão funcional, com functor `listof`: `listof arg1 ... argn`.

Os **termos lógicos**: `if c1 t1 ... cn tn default`

As **citações** podem ser feitas com `(quote TERMO)` ou `'TERMO`.

Frases

Constantes `true` ou `false`, (in)equações (comparação), frases relacional (predicados), lógica e quantificada.

Frases relacionais: `(relação t1 ... tn)`

Frases lógicas: `(not/and/or/i/!... t1 ... tn)`

Frases quantificadas: `(forall/exists var1 ... varn frase)`

Definições

Objetos, funções e relações (predicados).

- 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 (?) := (if (= (listof ?x @items) ?1) ?x))
- Definição de relações (=predicados)
 - (defrelation $r(v1 .. vn) := p$)
 - etc.
 - Exemplos:
 - (defrelation null (?) := (= ?1 (listof)))
 - (defrelation list (?x) :=
(exists (@1) (= ?x (listof (@1)))))

5.1.3 Meta-conhecimento

O **meta-conhecimento** formaliza conhecimento sobre o conhecimento. Para isto, recorre-se ao mecanismo da citação que permite tratar expressões como objetos, fazendo referência à sua representação em vez do seu significado (por exemplo 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). Podemos ilustrar da seguinte forma:

(quote Representação) $\xrightarrow{\text{refere-se a}}$ Representação $\xrightarrow{\text{refere-se a}}$ Objecto

5.1.4 Dimensões de conformação

O facto de ser uma linguagem **altamente expressiva** leva a que o KIF tenda a **sobrecarregar os sistemas de geração de inferência**, pelo que foi necessário definir várias dimensões de conformação, havendo para cada uma um perfil de conformação, que consiste numa seleção de níveis de conformação.

5.1.5 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

5.2 Engenharia do Conhecimento

A **base de conhecimento (BC)** é um conjunto de declarações que existem sobre aspetos de um certo domínio, que podem ser factos e regras do funcionamento, designadas genericamente por frases.

Nota: Com base na automatização e simplificação do conhecimento estudadas anteriormente, as **regras** serão cláusulas de Horn e os **factos** cláusulas sem literais negativos.

”Sócrates é um homem” é um facto, enquanto que ”Todos os homens são mortais” é uma regra. Cláusula de Horn porque $\text{Homem} \rightarrow \text{Mortal}$ que pode se representada por $\neg \text{Homem} \vee \text{Mortal}$.

A **engenharia do conhecimento**, por sua vez, é o processo de **preparar uma representação para uma determinada aplicação**, construindo para este fim bases de conhecimento. Não é mais do que a modelação de um sistema, que resulta numa base de dados que para além de factos tem também regras. Consiste nos seguintes passos:

- 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 conhecimento necessário. Neste caso usa-se *aprendizagem automática*.

5.2.1 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 \rightarrow *conceitos* (também chamados *classes* ou *tipos*)
- Substantivos próprios \rightarrow *objectos* (também chamados *instâncias*)
- Verbo “ser” \rightarrow pode indicar uma relação de *instanciação* (entre objecto e tipo) ou de *generalização* (entre subtipo e tipo)
- Verbos “ter” e “conter” \rightarrow podem indicar uma relação de composição
- Outros verbos \rightarrow podem sugerir outras relações relevantes

Convém avaliar a importância para o problema das palavras utilizadas bem como dos objetos, conceitos e relações subjacentes, ou seja:

- 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.

5.2.2 Ontologias

Uma **ontologia** é um vocabulário sobre um domínio conjugado com relações hierárquicas como membro e subtipo e eventualmente outras.

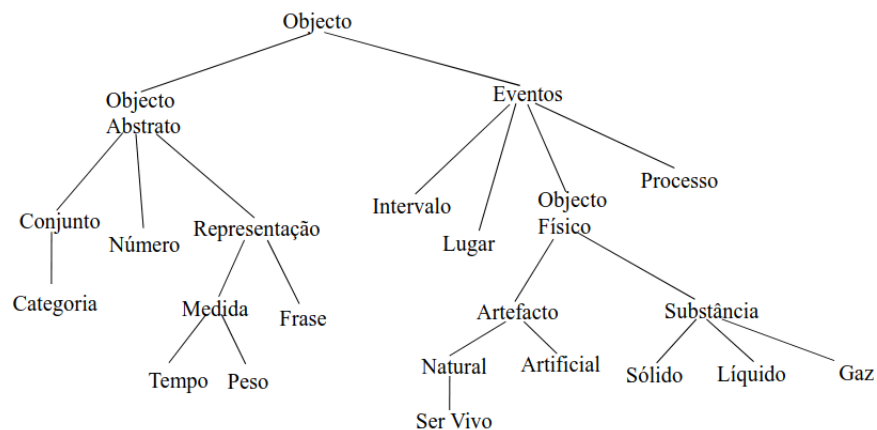
O objetivo de uma ontologia é captar a essência da organização do conhecimento num domínio.

5.2.3 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

Possível Ontologia Geral



5.3 Redes de Bayes

5.3.1 Redes de crença bayesianas

Aka "Redes de Bayes" permitem representar o **conhecimento impreciso** em termos de um conjunto de variáveis aleatórias e respetivas dependências. As dependências são expressas através de probabilidades condicionadas, sendo a rede um grafo dirigido acíclico.

5.3.2 Axiomas das probabilidades

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

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

- Proposições necessariamente verdadeiras têm probabilidade 1

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

- Proposições necessariamente falsas têm probabilidade 0

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

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

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

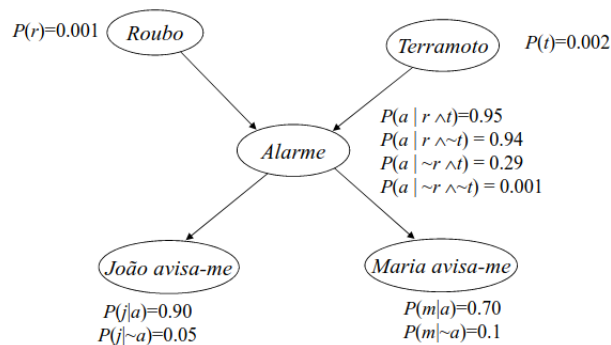
5.3.3 Probabilidades condicionadas

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

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

5.3.4 Exemplo

Por simplicidade, vamos focar em variáveis booleanas.



5.3.5 Redes de crença bayesianas - probabilidade conjunta

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

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

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

$$\begin{aligned} P(j \wedge m \wedge a | \neg t \wedge \neg r) &= P(j|a) \times P(m|a) \times P(a|\neg r \wedge \neg t) \times P(\neg r) \times P(\neg t) \\ &= 0.9 \times 0.7 \times 0.01 \times 0.99 \times 0.99 = 0.006272 \end{aligned}$$

5.3.6 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" }.

$$P(x_i = v_i) = \sum_{\substack{a_j \in \{v, f\} \\ j=1, \dots, k}} P(x_1 \wedge a_1 \wedge \dots \wedge a_k)$$

Seja:

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

6 Resolução automática de problemas

Um **problema** define-se como um objetivo cuja solução não é imediata, pelo que requer a pesquisa de uma solução (Exemplo: Dado um mapa, determinar o melhor caminho entre dois pontos, ou, Dada uma situação num jogo de xadrez, determinar uma boa jogada).

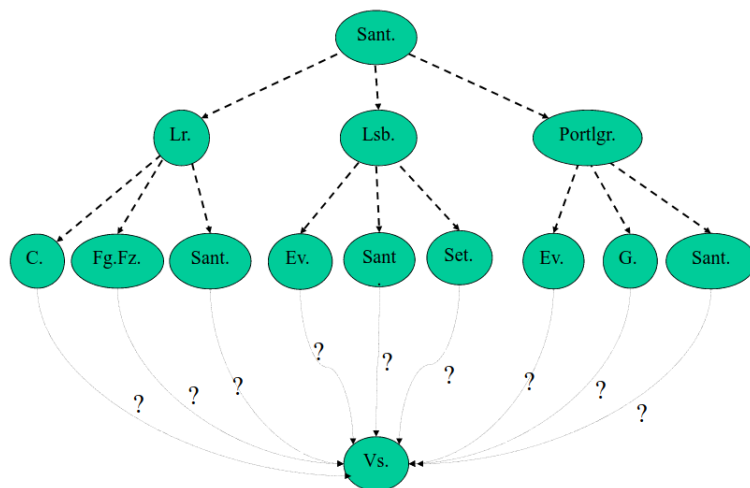
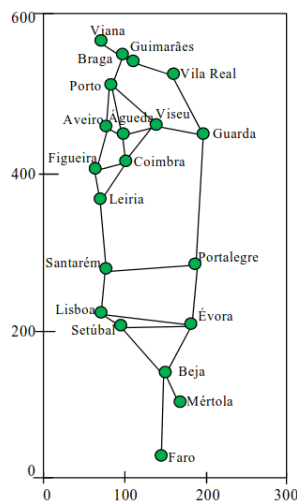
A sua **formulação** inclui a descrição de um ponto de partida, um conjunto de transições entre estados, uma função que diz se um estado satisfaz um objetivo e por vezes ainda uma função para avaliar o custo de uma solução.

A **pesquisa** é o processo que de forma recursiva ou iterativa, executa transições sucessivas entre estados até atingir um que satisfaça o objetivo.

6.1 Aplicação: determinar um percurso num mapa topológico

Dados: Distâncias por estrada entre cidades vizinhas.

Exemplo: Determinar um caminho de Santarém para a Viseu.



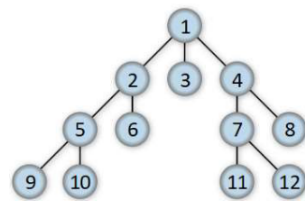
6.2 Pesquisa em árvore

Ao longo da pesquisa entre o estado inicial e o objetivo, as transições sucessivas entre estados estudadas vão originar ramificações que formam uma árvore.

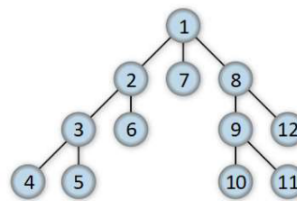
6.2.1 Algoritmo Genérico

```
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 contém um estado que satisfaz o objectivo
      então retornar a Solução correspondente
      senão expandir Folha e adicionar os nós resultantes à Árvore
  Fim do ciclo;
```

6.2.2 Percursos na árvore de pesquisa



Pesquisa em largura



Pesquisa em profundidade

6.2.3 Implementação baseada em fila

Em programação, a árvore pode ser implementada através de um fila, inicializada com o estado inicial e à qual são acrescentados os estados expandidos de acordo com a estratégia definida. A forma como a estratégia é definida distingue os vários métodos de pesquisa.

```
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)
```

6.2.4 Pesquisa em Profundidade

Nesta pesquisa não informada, a partir do nó raiz, **cada um dos seus ramos é explorado ao máximo**, antes de retroceder. Implementado através de método que adiciona nós à cabeça da lista (LIFO).

Podem ser implementadas pequenas variações, como **sem repetição de estados**, de forma a prevenir ciclos infinitos, convém garantir que os estados já visitados no caminho, não são novamente gerados, **com limite** de visitas em profundidade, ou **com limite crescente**, que consiste em resolver o problema para um limite e caso não seja encontrada solução aumentar esse limite (e assim sucessivamente).

6.3 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)
```

6.4 Avaliação de 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.

Otemalidade - primeira solução que a estratégia de pesquisa consegue encontrar é a melhor solução.

6.5 Pesquisa A*

Esta é uma pesquisa informada com uma função de avaliação bem definida que **dá primazia à exploração dos nós com melhor custo da raiz até ao objetivo** em detrimento dos restantes.

$f(n) = g(n) + h(n)$ Com $g(n)$ a dar o custo desde o nó inicial até ao atual e $h(n)$ a dar o custo estimado até ao objetivo.

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 esta função é admissível, então a pesquisa A* encontra sempre (um)a solução ótima. A pesquisa A* é também completa.

6.5.1 Variantes

Pesquisa de custo uniforme

Esta é uma variante da pesquisa A*, também conhecida por algoritmo de Dijkstra, onde a função de avaliação dá primazia aos nós com base no custo desde o nó inicial. Tem o comportamento parecido com o da **pesquisa em largura**, e caso exista a solução, a primeira solução encontrada é ótima.

$$f(n) = g(n) + 0, \text{ com } g(n) \text{ a dar o custo desde o nó inicial até ao atual.}$$

Pesquisa gulosa

Este método foca-se na proximidade ao objetivo. Tem um comportamento semelhante à **pesquisa em profundidade**, com a variante de que chega mais rápido à solução. Ao ignorar o custo acumulado, facilmente deixa escapar a solução ótima.

$$f(n) = h(n), \text{ com } h(n) \text{ a dar o custo estimado até ao objetivo.}$$

6.6 Pesquisa num grafo de estados

Numa pesquisa em árvore muito simplificada e em profundidade, facilmente se criam ciclos infinitos por repetição de estados em cada caminho estudado. Este problema pode ser resolvido com o registo dos estados estudados em cada caminho.

No entanto, não previne que transições a partir de diferentes estados levem ao mesmo estado. A **pesquisa num grafo de estados** tem como objetivo evitar este acontecimento de forma a tornar a pesquisa mais eficiente.

Assim, para além de implementar uma fila de nós abertos (não expandidos, em cada iteração, o primeiro nó em ABERTOS é selecionado para expansão), guarda também uma **lista de nós fechados** (já expandidos), para evitar a repetição de estados.

A sua implementação é semelhante ao da pesquisa em árvore, distinguindo-se apenas em algumas etapas.

6.6.1 Algoritmo

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

Tal como a pesquisa em árvore, a **pesquisa em grafo** utiliza árvore de pesquisa. No entanto, a pesquisa em árvore normal ignora a possibilidade de o espaço de estados ser um grafo (mesmo se 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.

6.7 Avaliação da pesquisa em árvore - factores de ramificação

Existem vários critérios que permitem avaliar a performance dos vários métodos de pesquisa em árvore que podemos utilizar. Os **fatores de ramificação** dão-nos uma métrica para esta avaliação.

A **ramificação média** consiste no quociente entre o número de nós resultantes da expansão (nós filho) pelo número de nós expandidos (mesmo que não tenham gerado filhos). Indica-nos a dificuldade do problema.

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

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.

O **fator de ramificação efetivo** indica a eficiência da técnica de pesquisa utilizada e tenta normalizar a ramificação, determinando o número de nós filho por nó (B), numa árvore com ramificação constante (progressão geométrica).

$$1 + B + B^2 + \dots + B^d = N$$

ou seja

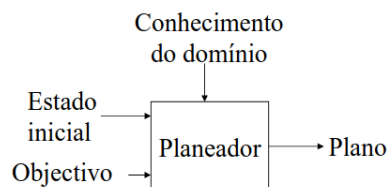
$$\frac{B^{d+1} - 1}{B - 1} = N$$

Nota: Resolve-se por métodos numéricos, ou seja, tentativa e erro.

6.8 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 dado estado inicial, se atinja um dado objetivo.

O **conhecimento do domínio** inclui uma descrição das condições de aplicabilidade e efeitos das ações possíveis.



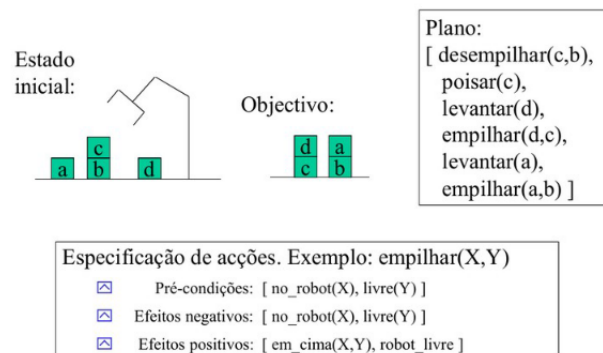
6.9 Representação de ações em problemas de planeamento

STRIPS - Uma abordagem bastante conhecida ao planeamento de soluções (Stanford Institute Problem Solver), desenvolvido nos anos 70.

A funcionalidade de um dado tipo de operação é definida pelos seus formalismos, através de uma estrutura denominada **operador**, que é composta da seguinte forma:

- **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 e que deixam de ser verdade ao executar-se a operação.
- **Efeitos positivos** (add list) - Um conjunto de fórmulas atómicas que representam propriedades do mundo e que passam a ser verdade ao executar-se uma operação.

6.9.1 Exemplo: planeamento no mundo dos blocos



6.10 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**, sendo que, quanto mais baixo for esse fator, melhor é a heurística.

Em alguns domínios, há funções de estimação de custos que naturalmente constituem heurísticas admissíveis (por exemplo, distância em linha reta no domínio dos caminhos entre cidades).

Em muitos outros domínios práticos, não há uma heurística admissível que seja óbvia (por exemplo, planeamento no mundo dos blocos).

6.11 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, pelo que 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: custo de uma solução ótima para um problema simplificado constitui uma heurística admissível para o problema original

6.12 Pesquisa A* - combinação de heurísticas

Se tivermos várias heurísticas admissíveis (h_1, h_2, \dots, h_n) , podemos combiná-las numa nova heurística:

$$H(n) = \max\{h_1(n), h_2(n), \dots, h_n(n)\}$$

Esta nova heurística tem as seguintes propriedades:

- Admissível;
- Dado que é uma melhor aproximação ao custo real, vai ser uma heurística melhor do que qualquer das outras.

6.13 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 otimalidade;
- Ou então, usar heurísticas com uma melhor aproximação média ao custo real, ainda que não sendo estritamente admissíveis, e não garantindo portanto a otimalidade da pesquisa

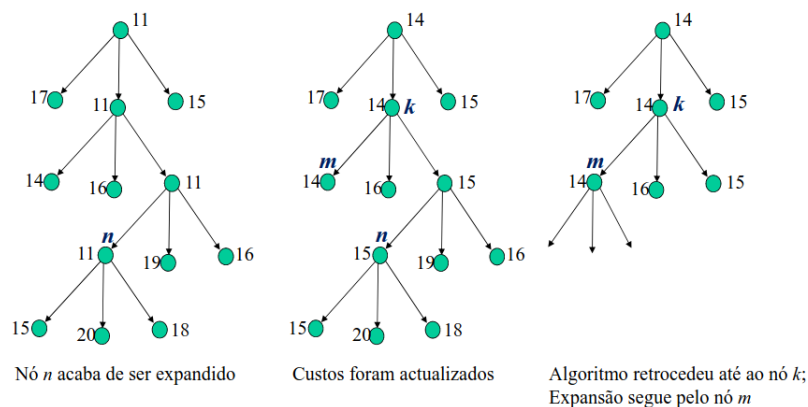
6.14 IDA*

- 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

6.15 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

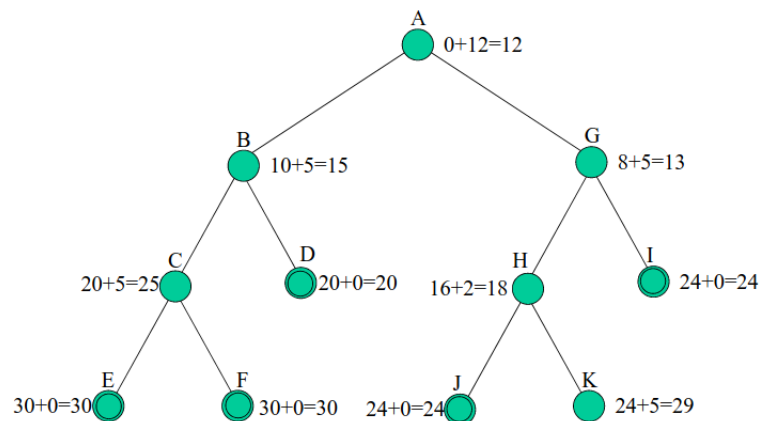
6.15.1 Exemplo



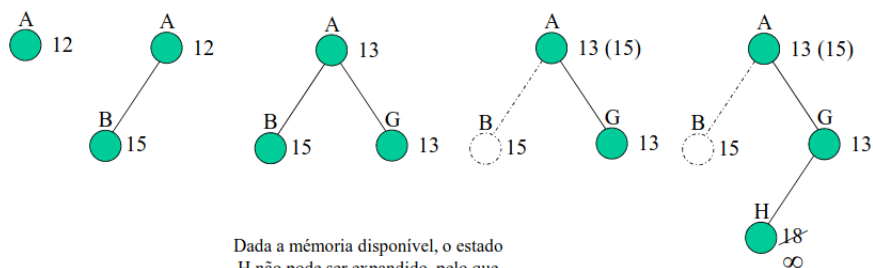
6.16 SMA*

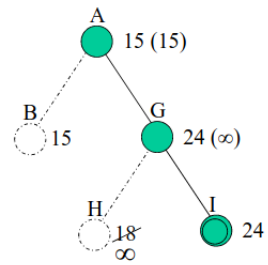
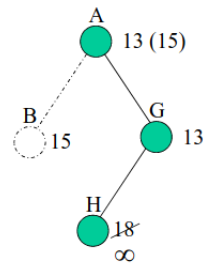
- A* com memória limitada simplificado (*simplified memory-bounded A**)
- Usa a memória disponível
 - Contraste com IDA* e RBFS: estes foram desenhados para poupar memória, independentemente de ela existir de sobra ou não
- Quando a memória chega ao limite, esquece (remove) o nó n com maior custo $f(n)=g(n)+h(n)$,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

6.16.1 Exemplo - espaço de estados



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





- 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