

# 1 Cos'è l'intelligenza artificiale

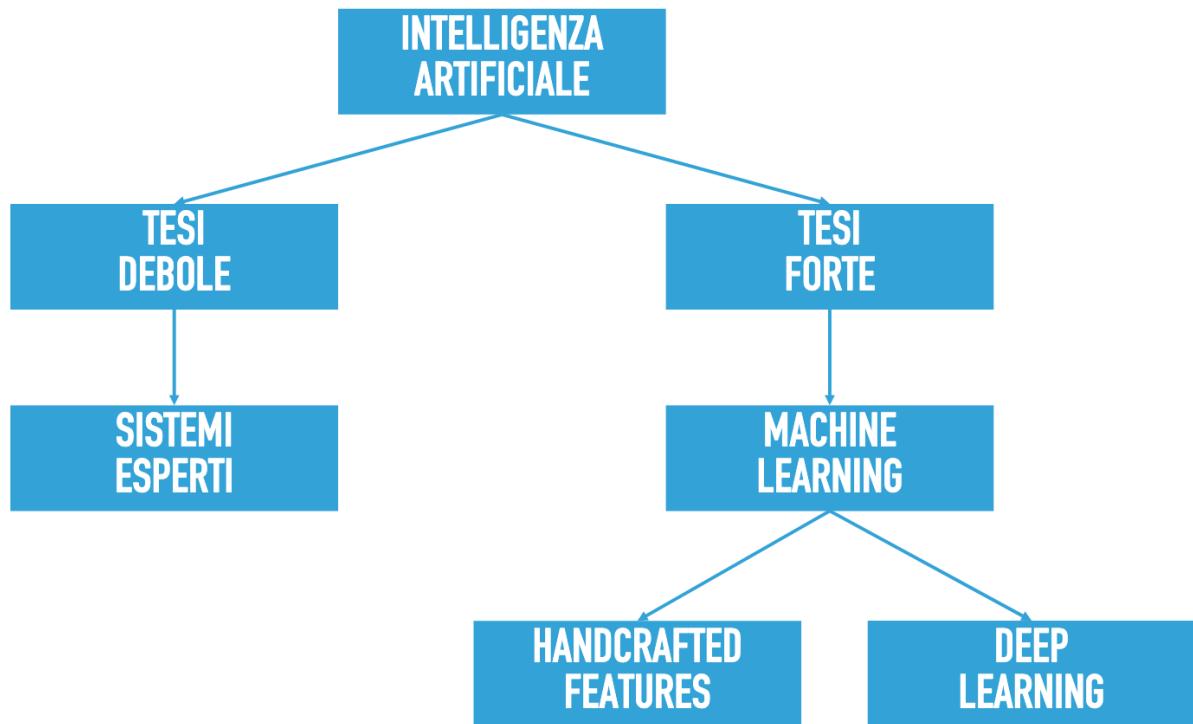
Nel 1950 Alan Turing propose un **esperimento mentale** (oggi conosciuto come **test di Turing**) col fine di dare una risposta concreta alla domanda: *"Una macchina è in grado di pensare?"*. Un computer supererà il test (quindi la macchina verrà definita **intelligente**) se un esaminatore umano, dopo aver posto alcune domande in forma scritta, non sarà in grado di capire se le risposte provengono da una persona o no.

Da ciò deriva una prima **definizione** di Intelligenza Artificiale:

## Definizione AI

Disciplina che studia tecniche e metodologie finalizzate alla realizzazione di sistemi artificiali (macchine) in grado di risolvere problemi complessi con un **comportamento «indistinguibile dall'essere umano»**.

# 2 Approcci all'intelligenza artificiale



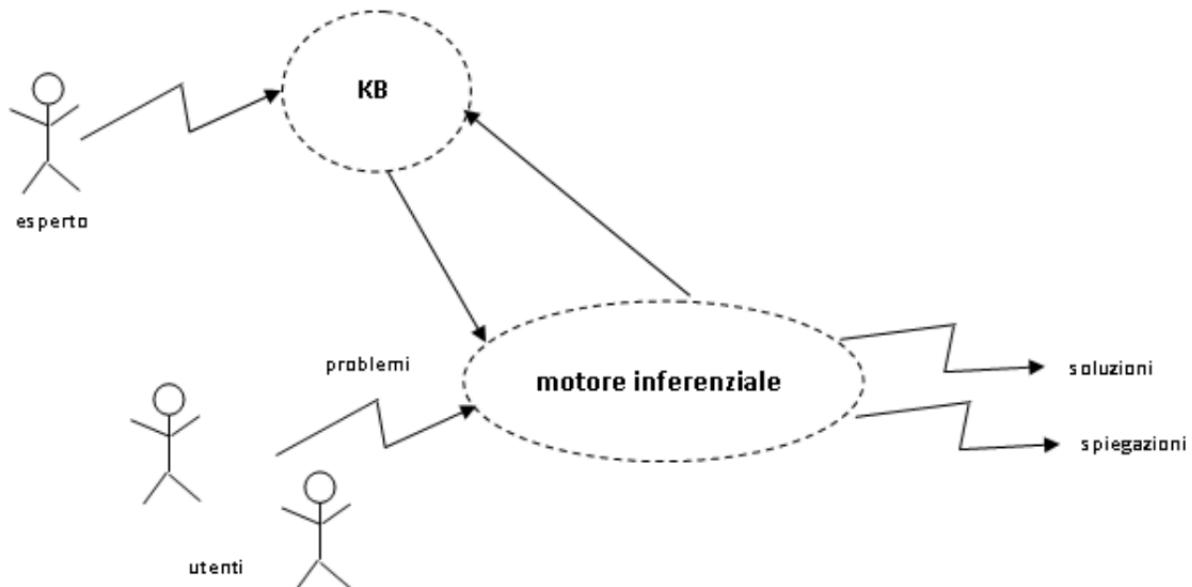
# 3 Intelligenza Artificiale Debole

L'**intelligenza artificiale debole** crea sistemi che possono **sembrare intelligenti** durante la risoluzione di problemi specifici, ma lo fanno seguendo regole e dati che hanno a disposizione, senza pensare o capire (senza ragionare) come un essere umano.

Questi sistemi usano una **"base di conoscenza"** - *KB* (cioè un archivio di fatti e regole) e un **"motore di ragionamento inferenziale"** che applica queste regole per trovare soluzioni.

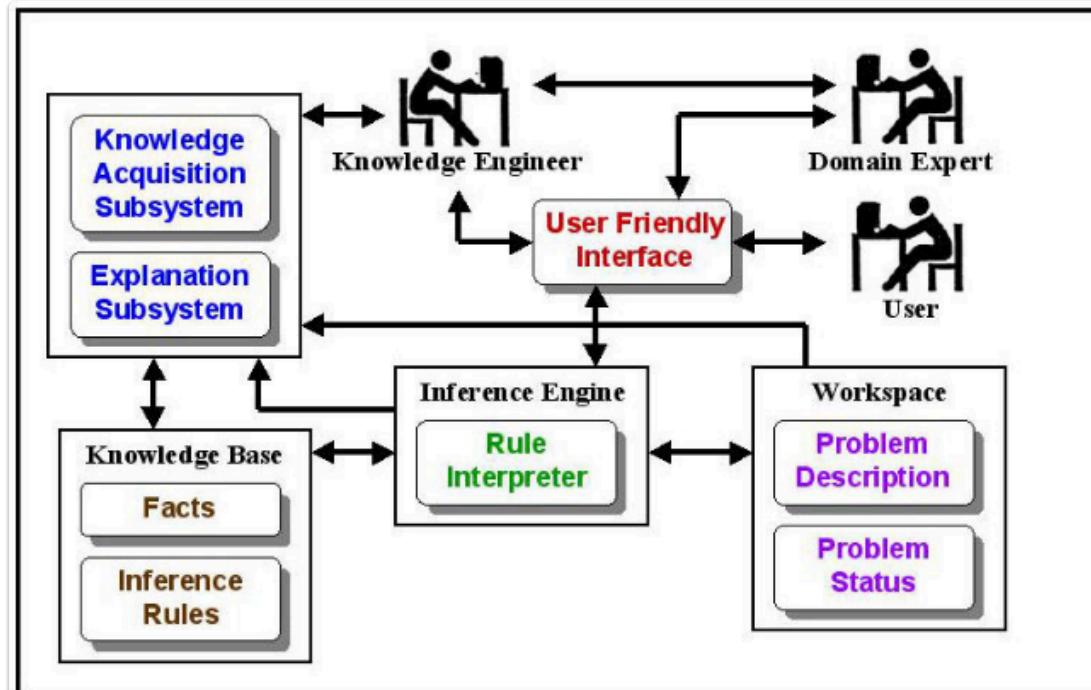
In pratica, la macchina **simula alcune capacità umane**, ma non ha coscienza né comprensione reale: agisce "come se" fosse intelligente, ma **non lo è davvero**. Il sistema infatti riesce a risolvere solamente i problemi che

coinvolgono la conoscenza presente nella KB.



### 3.1 Sistema Esperto

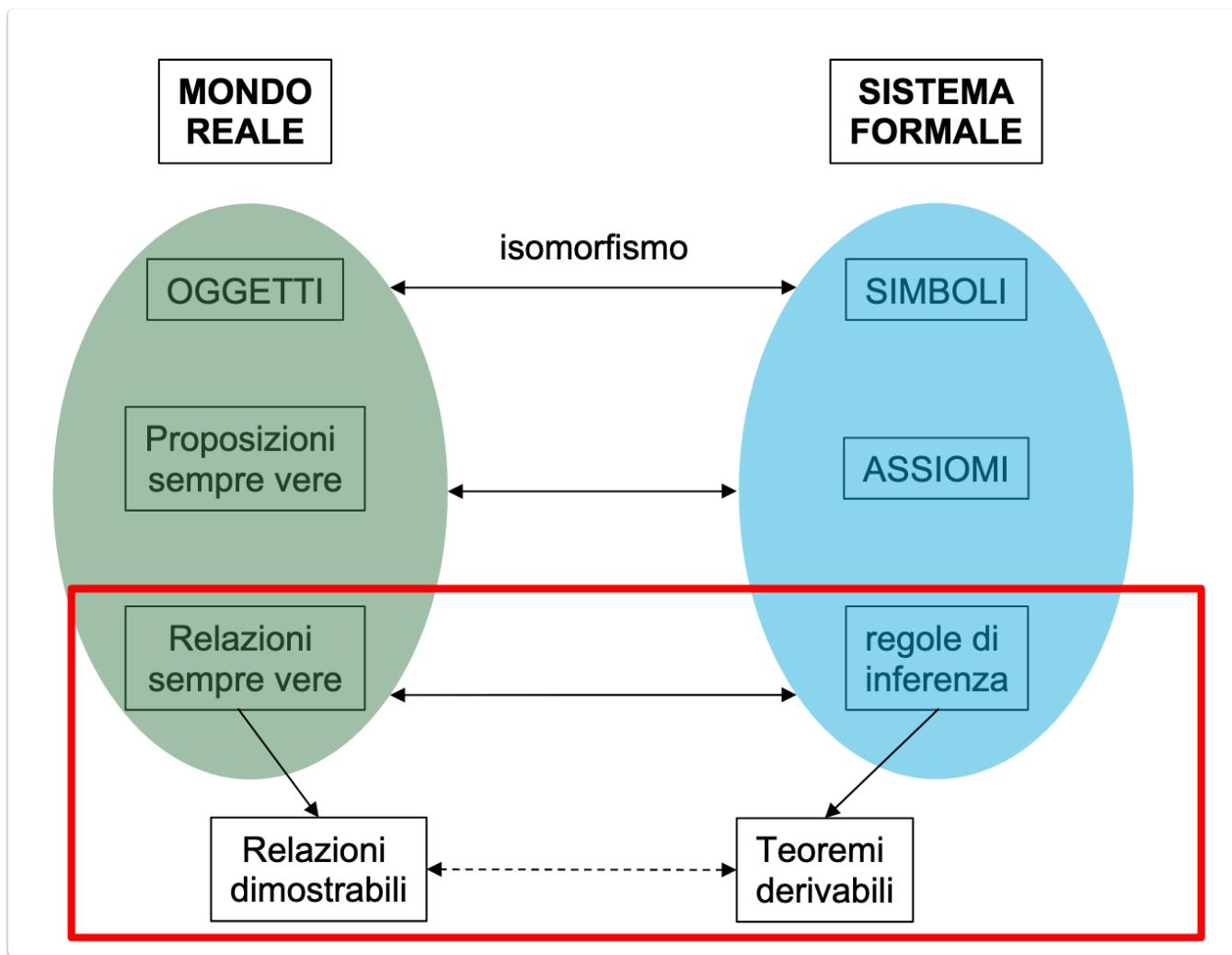
La tesi debole porta al concetto di **sistema esperto** ovvero programmi basati su una **base di conoscenza** (creata manualmente da esperti umani) e un **motore inferenziale** che applica regole predeterminate per emulare le competenze specialistiche in un dominio specifico.



### 3.2 Sistema Formale

L'elemento base della tesi debole è il **sistema formale** (SF), una struttura logico-matematica rigorosa che utilizza simboli, regole e procedure definite per generare conclusioni attraverso processi deduttivi. Il sistema formale viene utilizzato per **rappresentare la conoscenza** mediante un insieme di simboli, regole (di inferenza) e assiomi diversi.

Gli oggetti del mondo reale corrispondono a **simboli** nel sistema formale; le proposizioni (vere) corrispondono ad **assiomi**. Le relazioni tra gli oggetti reali si traducono nelle **regole di inferenza** del sistema formale.



Sul sistema formale opera il **motore inferenziale**, una sorta di algoritmo che applica le regole del sistema formale per derivare conclusioni. Fissato un **teorema** da dimostrare, ha l'obiettivo di applicare in successione le regole di inferenza in modo da derivarlo, a partire dagli assiomi.

Il motore inferenziale segue un **procedimento di derivazione di nuove teorie** (si parla anche di dimostrazione di nuovi teoremi) a partire dalle teorie messe inizialmente a disposizione (assiomi).

Il motore inferenziale si ferma quando ha "espanso" tutti i nodi e non è detto che il procedimento abbia durata finita. Se il grafo generato ha profondità **infinita**, il sistema è **non decidibile** in quanto il suo motore inferenziale non si ferma mai e pertanto non è possibile dare una risposta alla domanda. Un sistema del genere non consente di prendere **decisioni**.

### 3.2.1 Proprietà dei sistemi formali

I sistemi formali godono di alcune **proprietà**, non di tipo intrinseco ma direttamente connesse all'isomorfismo con il mondo reale:

- **Decidibilità**: per rendere decidibile in ogni caso un problema si fa in modo che lo spazio delle possibili risposte risulti **finito**. In questo modo una risposta viene comunque data ma, in caso di esito negativo, non possiamo dire con certezza che il teorema sia **indimostrabile**;
- **Coerenza**: ogni teorema da esso derivato corrisponde, alla luce dell'isomorfismo, ad una verità del mondo reale;
- **Completo**: tutti i teoremi del mondo reale sono da esso derivabili

Esse garantiscono:

1. tutto ciò che afferma il sistema formale è vero;

2. tutto ciò che è vero è rappresentabile col sistema formale;

### 3.2.2 Rappresentazione della conoscenza

Esistono 2 principali modalità di rappresentazione della conoscenza all'interno di un sistema formale:

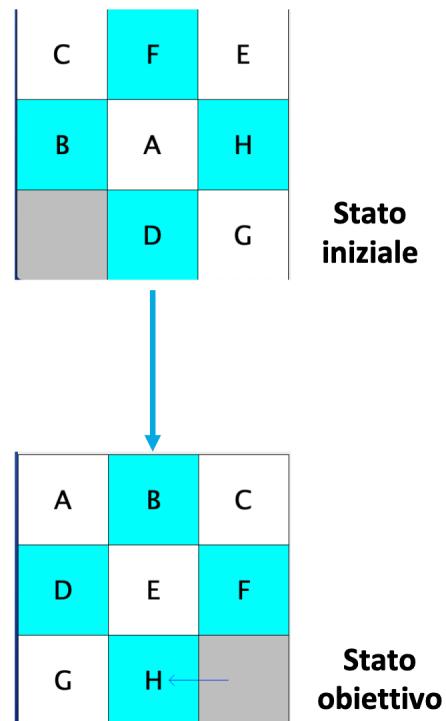
- Rappresentazione nello Spazio degli Stati (**SSR** - State Space Representation)
- Rappresentazione per **decomposizione in sottoproblemi**

#### 3.2.2.1 SSR - State Space Representation

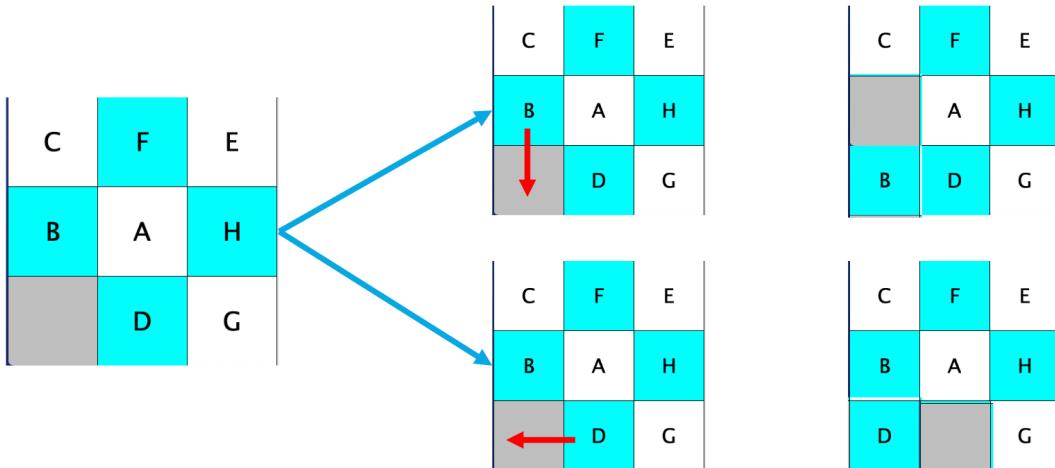
Un problema può essere descritto mediante un **insieme di stati** che caratterizzano le **soluzioni parziali** al problema. Con stato si intende una delle possibili **configurazioni** del sistema.

#### Il gioco «8-puzzle»

- Cosa si intende per stato?  
Partiamo con un esempio!
- Problema: **il gioco «8-puzzle»**
  - Il gioco prevede una **scacchiera 3x3**;
  - 8 caselle sono occupate dalle **lettere A,...,H** (inizialmente messe a caso);  
una delle caselle è vuota.
  - L'**obiettivo** è portare la scacchiera nella configurazione in cui le prime 8 caselle contengono (in **ordine**) le lettere A,...,H e la nona casella è vuota.

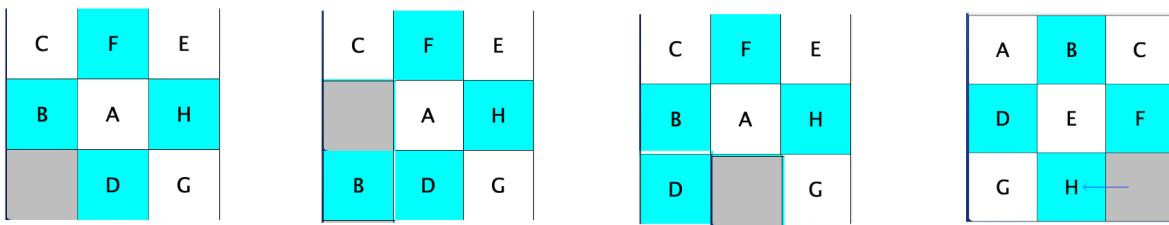


- Le mosse possibili sono lo spostamento nella casella vuota di una lettera contenuta in una casella adiacente;
  - come caselle adiacenti si considerano solo quelle nelle quattro direzioni cardinali (nord-sud-ovest-est) rispetto alla casella vuota.



- Quale è lo **stato** nel gioco 8-puzzle?
    - Ogni possibile configurazione delle tessere è uno stato.

- Alcuni esempi:



Stato iniziale ( $s_0$ )

Stato s1

Stato s2

Stato sg

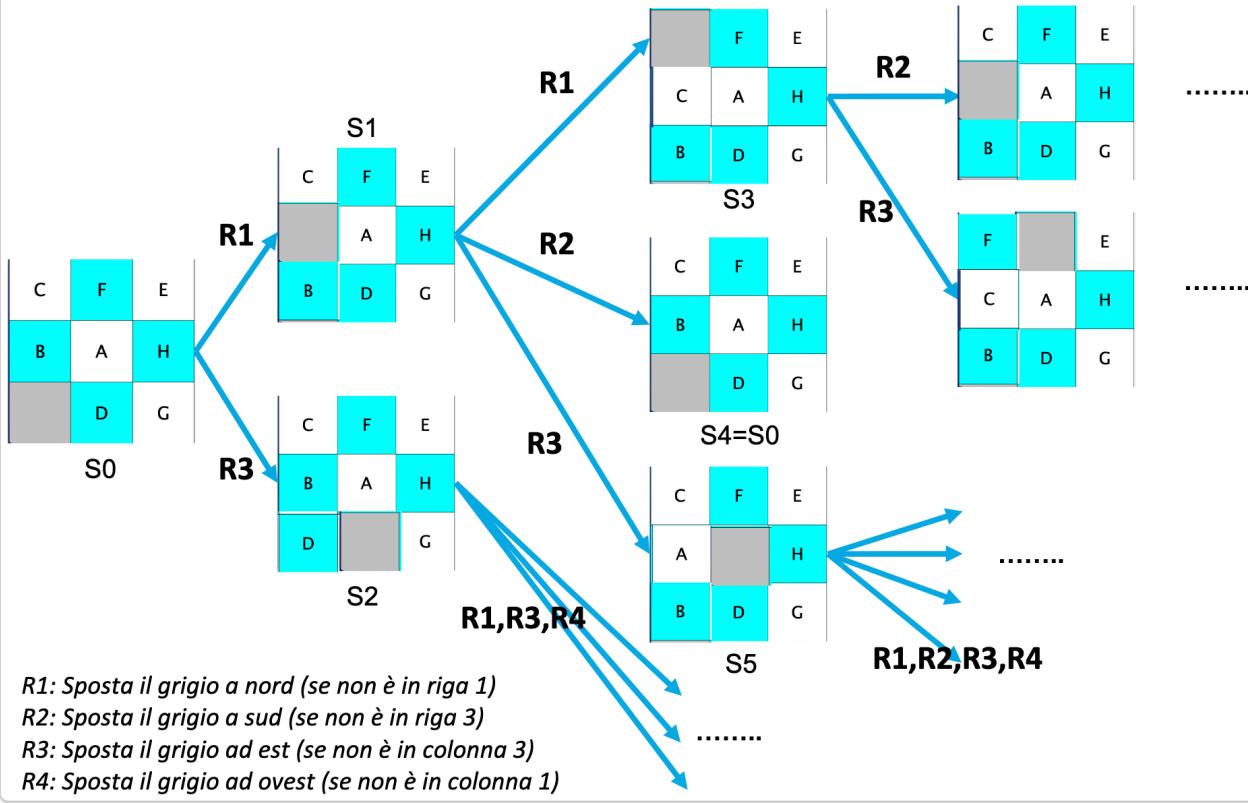
- Avremo quindi un numero molto elevato di stati «raggiungibili» a partire dallo stato iniziale.

L'SSR ci consente di risolvere il problema **senza dover definire manualmente l'algoritmo risolutivo** (che non è sempre facile da trovare) o meglio delegando la ricerca dell'algoritmo al **motore inferenziale**.

Bisogna pertanto definire il problema all'interno del sistema formale:

1. **Stato:** matrice  $3 \times 3$  composta dai caratteri (simboli) che vanno da *A* ad *H*;
  2. **Regole:** si definiscono le 4 regole di inferenza e le relative precondizioni
    1. **R1:** Sposta il grigio a nord (se non è in riga 1)
    2. **R2:** Sposta il grigio a sud (se non è in riga 3)
    3. **R3:** Sposta il grigio ad est (se non è in colonna 3)
    4. **R4:** Sposta il grigio ad ovest (se non è in colonna 1)

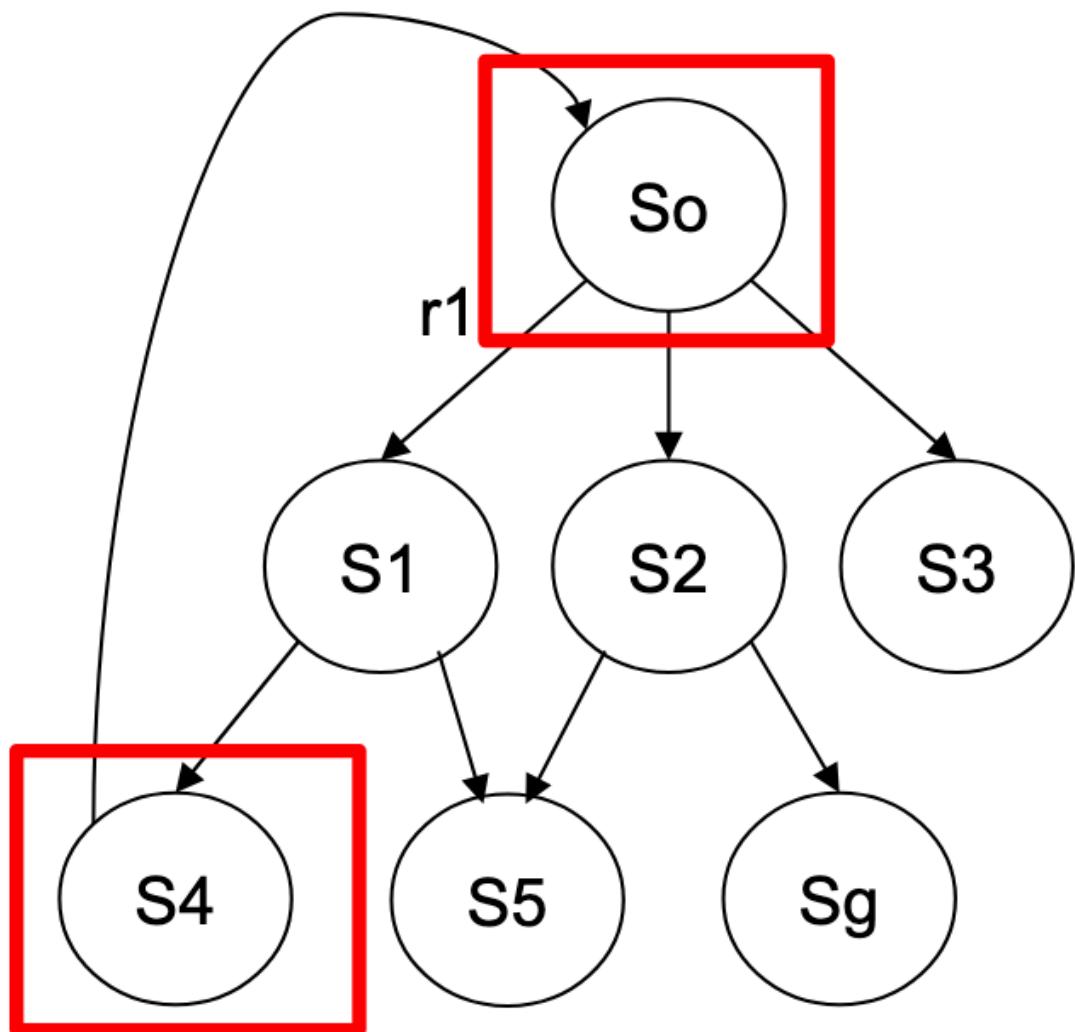
## Il gioco «8-puzzle»: esplorazione dello spazio degli stati



Lo spazio degli stati viene rappresentato mediante un grafo:

- $S_0$ : stato di partenza;
- $S_g$ : predato (stato da raggiungere). Si parla di predato e non di "stato obiettivo" in quanto un problema prevede che il predato sia soddisfatto da più stati diversi
- La transizione fra stati può essere effettuata mediante le regole di inferenza;

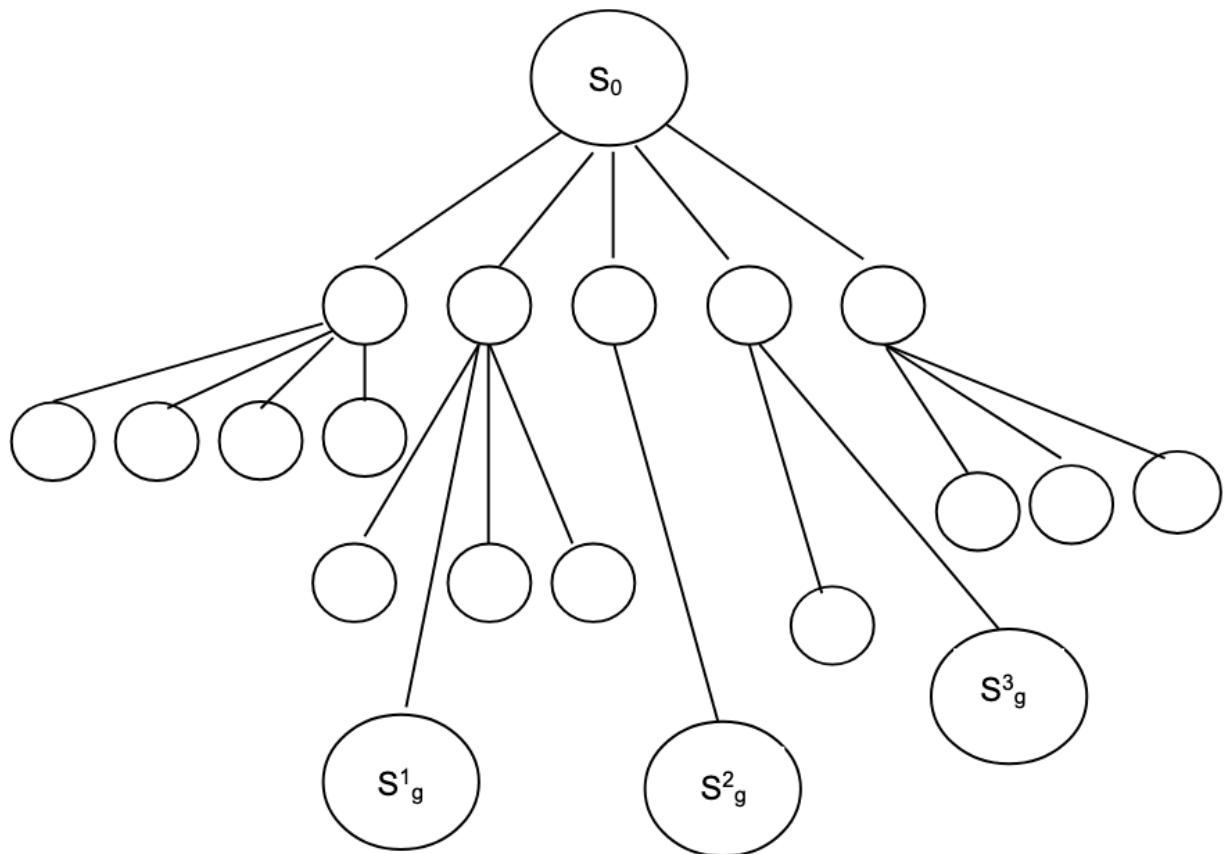
In generale, uno stato può essere descritto da un percorso che parte dallo stato iniziale  $S_0$  e si sviluppa attraverso l'applicazione di una successione di regole.



### 3.2.2.1.1 Tipologie di (algoritmi di) ricerca

#### 3.2.2.1.1.1 Ricerca in avanti e all'indietro (forward/backward search)

Una prima distinzione riguarda la **direzione** con cui l'algoritmo di ricerca analizza il grafo.



La **ricerca in avanti** (forward) consiste nel naturale modo di lettura del grafo: si ricerca, a partire da  $S_0$ , il percorso che porta ad  $S_g$ .

Questo approccio presenta tuttavia una problematica relativa al fatto che è necessario **esplorare tutte le strade alternative**, il cui numero cresce esponenzialmente con la profondità del grafo. Per ogni livello raggiunto bisogna quindi **esplorare tutti i suoi figli**.

Supponendo di avere a disposizione  $n$  regole, e che si verifichi sempre il caso peggiore, per il 1 livello le alternative sono  $n$ , per il 2 livello diventano  $n^2$ , ... la crescita è esponenziale e segue la distribuzione  $n^d$ , dove  $d$  è la profondità del grafo. Ciò può dare luogo a tempi di **calcolo inaccettabili**!

Dualmente, con la **ricerca all'indietro** (backward), si parte dal nodo finale  $S_g$  e, applicando a ritroso le regole di inferenza, si risale a ritroso la catena degli stati cercando di raggiungere  $S_0$ . In questo caso non si pone il problema dell'analisi delle strade alternative in quanto, percorrendo il grafo al contrario, è possibile prendere una **sola strada**. Questo approccio **non è tuttavia sempre percorribile** dal momento che l'applicazione di una regola in senso inverso è un problema intrattabile in certi casi oppure, più semplicemente, non si conosce a priori la soluzione finale.

### 3.2.2.1.1.2 Algoritmi di ricerca cieca

L' algoritmo di ricerca cieca **non contiene alcuna conoscenza del problema**, ma si limita a percorrere il grafo, stato per stato, finché non trova quello corrispondente alla **soluzione** (se la trova). La ricerca cieca **non sfrutta la conoscenza** delle caratteristiche del problema ma solo il relativo grafo degli stati.

#### Caratteristiche algoritmi ricerca cieca

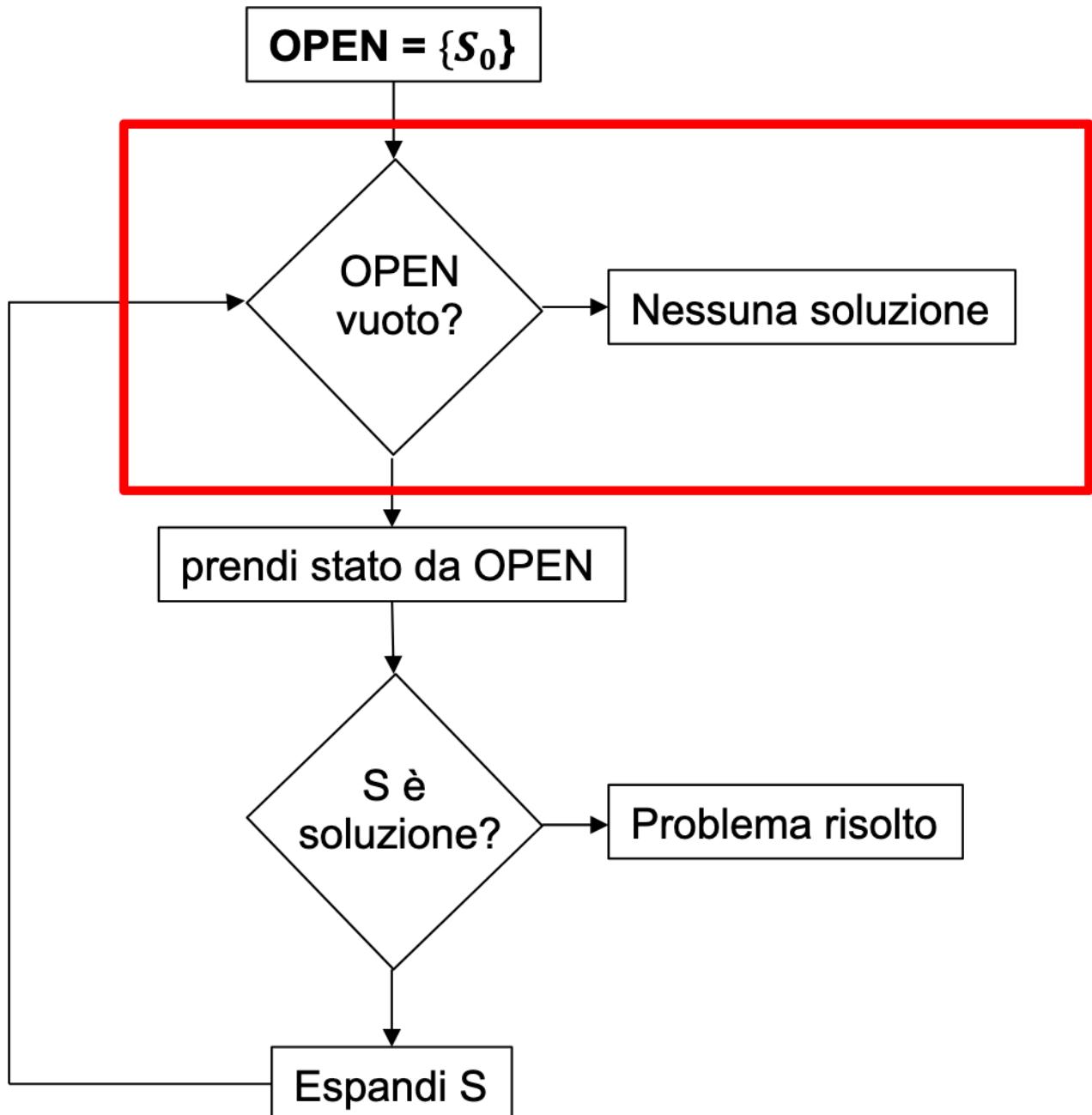
**PRO:**

- Il progettista deve semplicemente costruire la SSR

#### CONTRO:

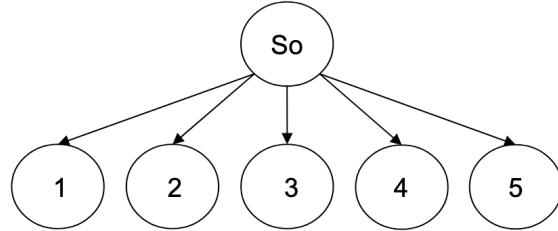
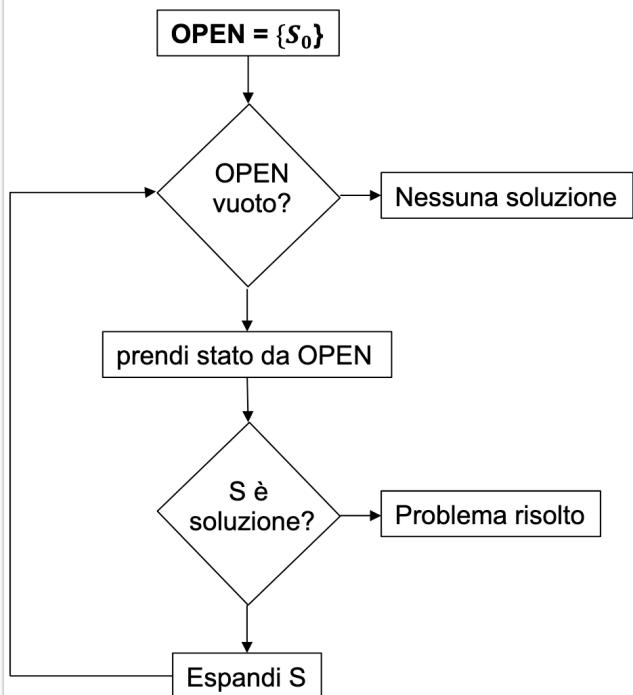
- il grafo che rappresenta la SSR può essere **molto grande o addirittura infinito**, il che renderebbe molto oneroso in termini computazionali l'algoritmo di ricerca della soluzione.

In tali algoritmi si fa uso di una struttura dati chiamata "**OPEN**" che contiene i nodi dello spazio degli stati che l'algoritmo prenderà in **considerazione** per verificare se sono o una soluzione, o parte del percorso necessario a raggiungerla. Inizialmente contiene solamente lo stato iniziale  $S_0$ , che viene detto **aperto**.



Se **OPEN** non è vuoto, l'algoritmo sceglie (rimuovendolo) uno stato  $S$  contenuto al suo interno e verifica se è una soluzione al problema. Se  $S$  è **soluzione**, l'algoritmo **termina**, altrimenti lo stato viene **espanso** applicando le regole di inferenza e costruendo in questo modo un **nuovi stati** (sostanzialmente si procede nel grafo), i quali vengono successivamente aggiunti alla struttura dati. Il ciclo si ripete finché non si raggiunge un'ipotetica soluzione.

$$OPEN = \{1 - 2 - 3 - 4 - 5\}$$



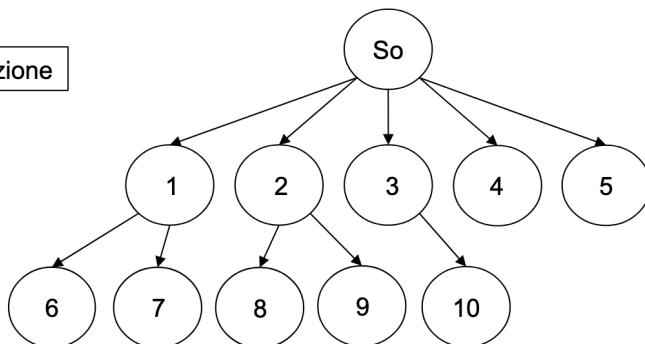
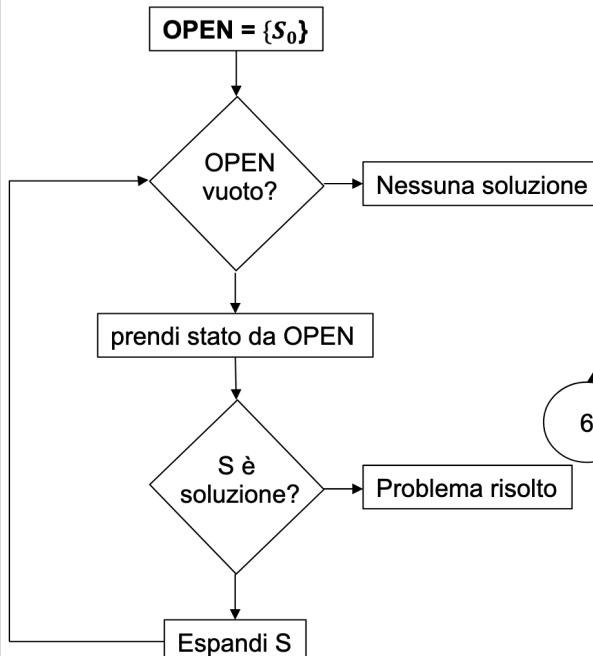
Si elimina  $S_0$  dall'insieme OPEN e lo si espande, generando TUTTI I NODI 1 – 2 – 3 – 4 – 5, che vengono inseriti in OPEN

La struttura  $OPEN$  può contenere **svariati stati** e pertanto è necessario decidere in che ordine prelevarli:

- politica **FIFO**: ricerca in **ampiezza** - BFS (Breadth First Search). Gli stati del livello  $i + 1$  vengono esplorati solamente dopo che sono stati esplorati tutti gli stati del livello  $i$ .

## Ricerca in ampiezza

$$OPEN = \{6 - 7 - 8 - 9 - 10\}$$

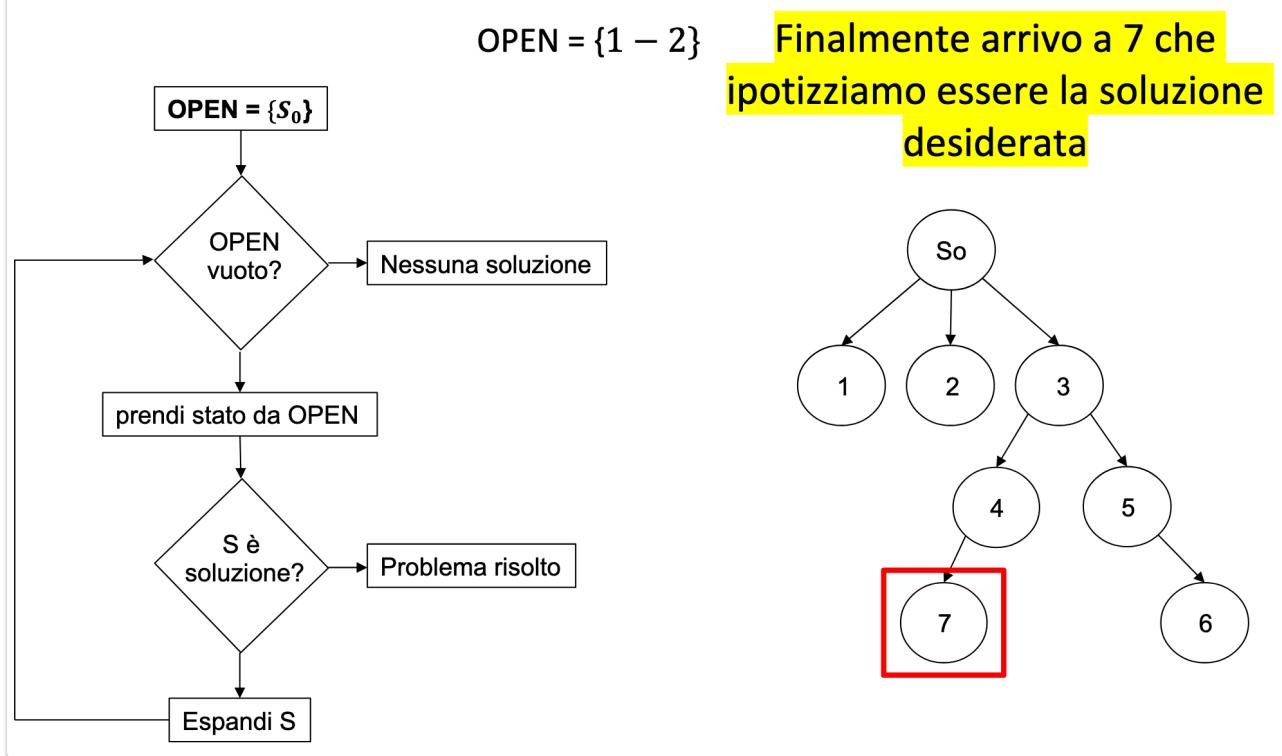


Finalmente arrivo a 6 che ipotizziamo essere la soluzione desiderata

- politica **LIFO**: ricerca in **profondità** - DFS (Depth First Search). L'algoritmo visita uno stato e, prima di considerare i suoi "fratelli" (cioè gli altri nodi sullo stesso livello), esplora ricorsivamente tutti i suoi figli e i loro discendenti. Solo dopo aver esaurito tutte le possibilità lungo un percorso, torna indietro e riprende

l'esplorazione dagli altri nodi non ancora visitati.

## Ricerca in profondità

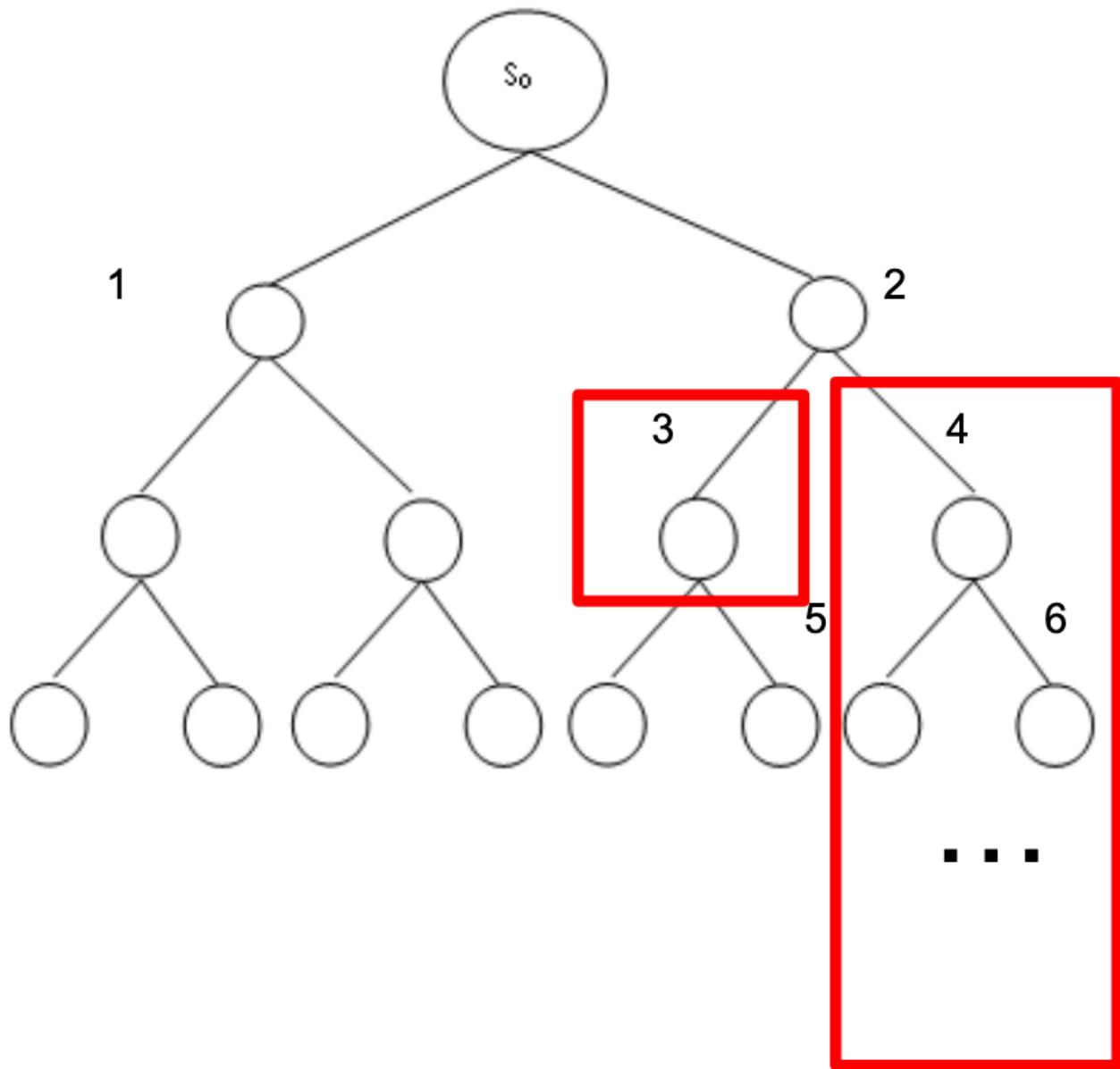


### 3.2.2.1.1.3 Ricerca in ampiezza e in profondità (BFS vs DFS)

- **BFS**
  - all'interno dell'insieme OPEN, in corrispondenza del livello k potrebbe essere necessario inserire in OPEN  $2^k$  stati.
  - L'occupazione di memoria cresce in **maniera esponenziale col numero dei nodi dell'albero**, un dato preoccupante, poiché sarebbe sufficiente un albero anche di pochi livelli per saturare la memoria del computer.
  - Ad esempio se ogni stato occupasse un solo byte (ipotesi comunque inverosimile) basterebbero 30 livelli per saturare 1 Gbyte di memoria.
- **DFS**
  - per un albero di livello massimo k con 2 regole applicabili ad ogni stato l'insieme OPEN non conterrà mai più di k stati. Anche se le regole applicabili ad ogni nodo fossero più di 2, il **numero di stati rimarrebbe comunque proporzionale alla profondità**; ad esempio per 3 regole avremmo nel caso peggiore  $2^k$  nodi aperti, per 4 regole  $3^k$  e via dicendo.

Se la soluzione non esiste, il problema è **indecidibile** e quindi **entrambi** gli algoritmi rimangono in **stallo**. Se esistesse, l'algoritmo **BFS** dato il suo funzionamento 'per livelli' **arriverebbe presto o tardi** a determinarla (è

considerato un algoritmo "completo" in quanto, se esiste, garantisce di trovare una soluzione).



L'algoritmo **DFS** potrebbe trovarla, ma, nel caso in cui fosse presente un ramo infinito, l'algoritmo potrebbe rimanere "*intrappolato*", senza fornire alcuna soluzione.

La ricerca in ampiezza (**BFS**) provoca una **occupazione di memoria** notevole, ma fornisce la garanzia di trovare sempre la soluzione, se esiste.

La ricerca in profondità (**DFS**) richiede **poca memoria** ma può rendere il sistema **indecidibile** anche quando una soluzione del problema in effetti esiste.

#### 3.2.2.1.4 Algoritmi di ricerca euristica (informata)

Negli algoritmi informati, si **sfrutta la conoscenza** che si possiede sulla natura del problema per **indirizzare** in maniera mirata l'algoritmo del motore inferenziale verso la soluzione del problema.

##### Caratteristiche algoritmi ricerca euristica

**PRO:**

- Tempi di ricerca notevolmente minori;

## CONTRO:

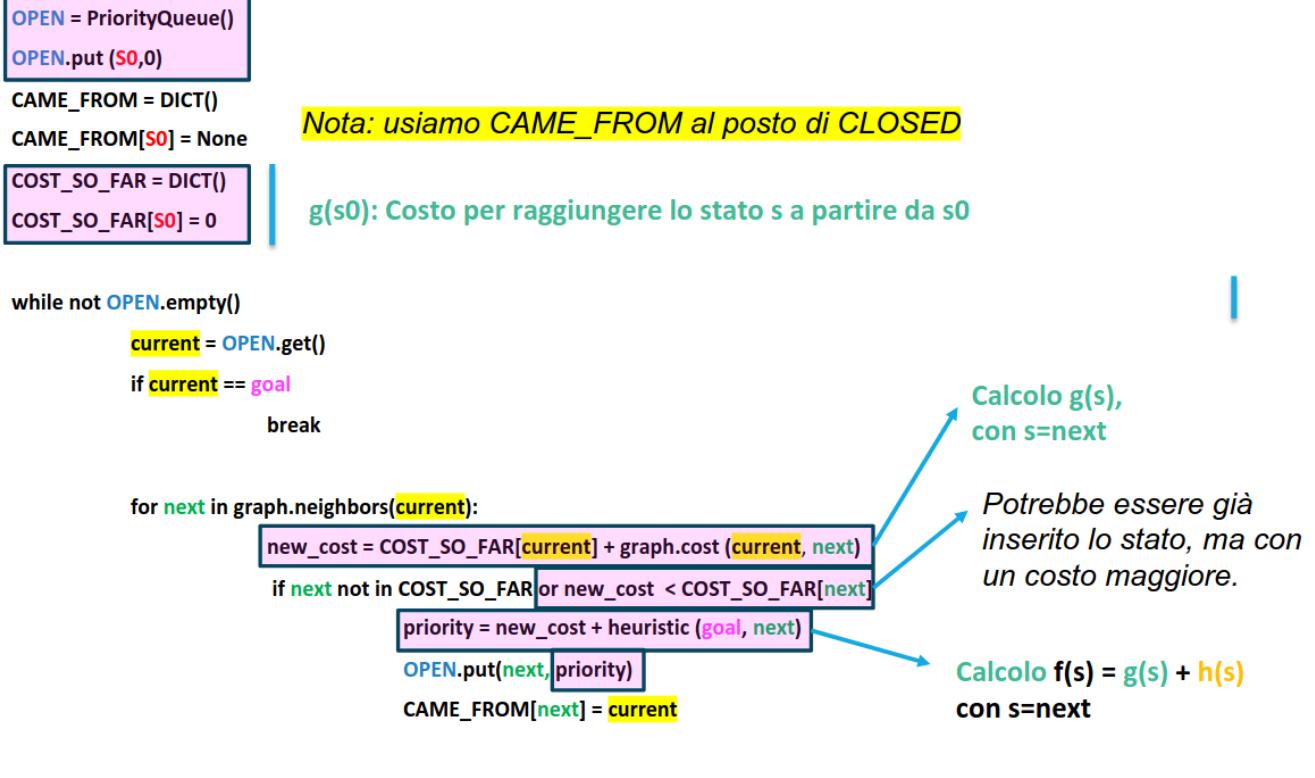
- Il progettista deve provvedere a fornire la conoscenza sufficiente a guidare l'algoritmo

Un esempio potrebbe essere quello di inserire nella struttura *OPEN* gli stati più promettenti, utilizzando una coda a priorità. In tal caso è necessario calcolare una **funzione di promettenza**:

$$f(s) = g(s) + h(s)$$

- $g(s)$  costo per raggiungere lo stato  $s$  a partire da  $s_0$  calcolato
- $h(s)$  costo per raggiungere lo stato goal da  $s$  stimato

La funzione  $h(s)$  è detta **funzione euristica**:



Generalmente non è possibile conoscere con precisione  $h(s)$  ma è possibile approssimarla. Approssimando tuttavia, non viene eseguita la potatura ottimale dell'albero e quindi il tempo necessaria a raggiungere la soluzione ottima non è quello minimo possibile.

Per trovare la soluzione ottima deve essere verificata la condizione di ammissibilità:

$$0 \leq h(s) \leq h^*(s)$$

Dove  $h^*(s)$  rappresenta il costo reale.  $h(s)$  deve quindi essere una stima per difetto della  $h^*(s)$ .

Possiamo fare di meglio con una funzione euristica «più informata»?

**Funzione euristica #2:** Per ciascuna tessera fuori posto consideriamo il numero di passi necessari per raggiungere la posizione giusta supponendo che non ci siano altre tessere ad ostacolare il cammino

E	F	C	A → 2 mosse	
B	A		B → 2 mosse	
H	D	G	C → 0 mosse	$h1(s) = 7$
		s	D → 2 mosse	
			E → 2 mosse	$h2(s) = 13$
			F → 2 mosse	
			G → 2 mosse	
			H → 1 mossaa	

Ai fini della scelta dell'**euristica** occorre considerare che al calcolo di ciascuna euristica è associato un certo **costo computazionale**, e d'altra parte ciascuna euristica assicura un diverso grado di potatura dello spazio degli stati.

Una euristica che dia luogo ad una **maggior potatura** richiede generalmente anche uno **sforzo computazionale maggiore**.

**Breadth-first search**

Stati visitati: 82047 Costo corrente: 22.0 Soluzione trovata.

**Ricerca in ampiezza**  
Stati visitati: 82047  
Costo (#livelli): 22

**Depth-first search**

Stati visitati: 13581 Costo corrente: 13280.0 Soluzione trovata.

**Ricerca in profondità**  
Stati visitati: 13581  
Costo (#livelli): 13280

**A\* search**

Stati visitati: 1095 Costo corrente: 22.0 Euristica: 0.0 Soluzione trovata.

**A\***  
Stati visitati: 1095  
Costo (#livelli): 22

**A\***

Stati visitati: 1095  
Costo (#livelli): 22

**Euristica h2**

**A\***

Stati visitati: 7307  
Costo (#livelli): 22

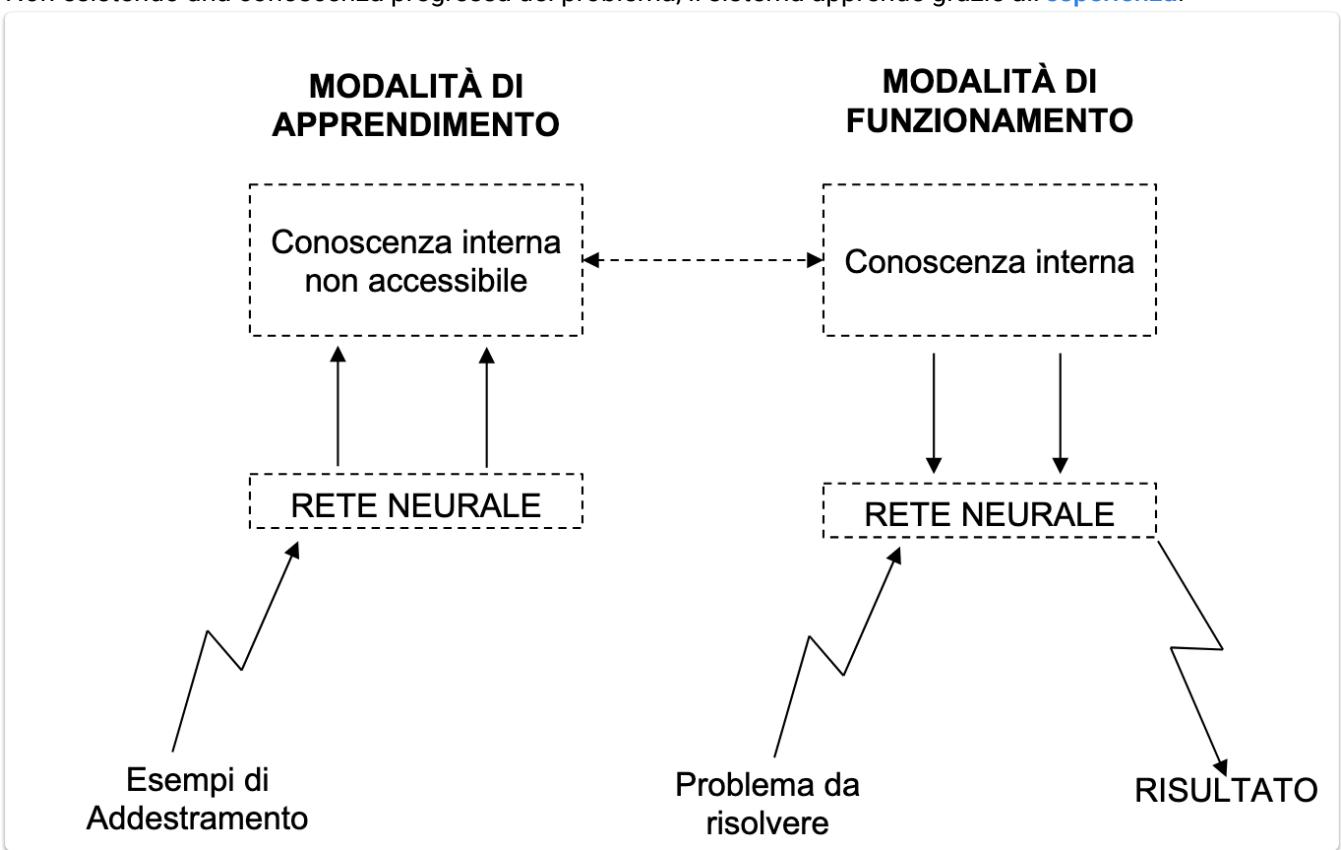
**Euristica h1**

## 4 Intelligenza Artificiale Forte

I sostenitori della **tesi forte** sostengono invece che opportune forme di intelligenza artificiale possano veramente **ragionare** e risolvere problemi, diventando **sapienti** e coscienti di sé.

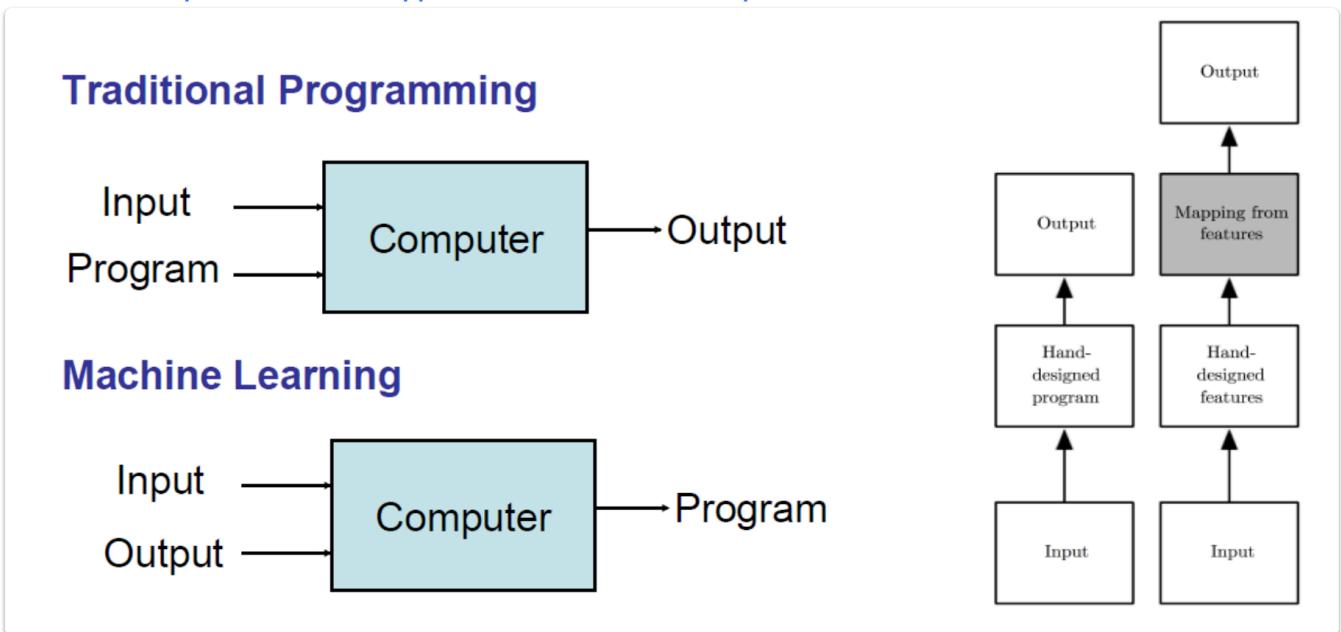
Ne sono un esempio le macchine capaci di imparare a svolgere un determinato compito utilizzando l'esperienza (**machine learning**).

Non esistendo una conoscenza pregressa del problema, il sistema apprende grazie all'[esperienza](#).



## 4.1 Machine Learning

Il [Machine Learning](#) è una disciplina che studia algoritmi capaci di imparare a svolgere un particolare compito utilizzando l'[esperienza](#), ovvero [apprendendo mediante esempi](#).



L'apprendimento (o meglio la [classificazione](#)) può avvenire tramite **2 diverse strategie**: l'estrazione manuale delle features ([handcrafted features](#)) e "*l'apprendimento automatico*" ([deep learning](#)).

### 4.1.1 Handcrafted Features

La strategia storica prevede la [manuale estrazione delle features](#) più rilevanti in base al problema da svolgere e la successiva somministrazione delle stesse ad un [algoritmo di machine learning](#). Richiede la presenza di esperti del dominio che definiscono manualmente le caratteristiche rilevanti e le regole del sistema in un linguaggio comprensibile dal sistema. Per un buon risultato è fondamentale riuscire ad individuare le [features](#)

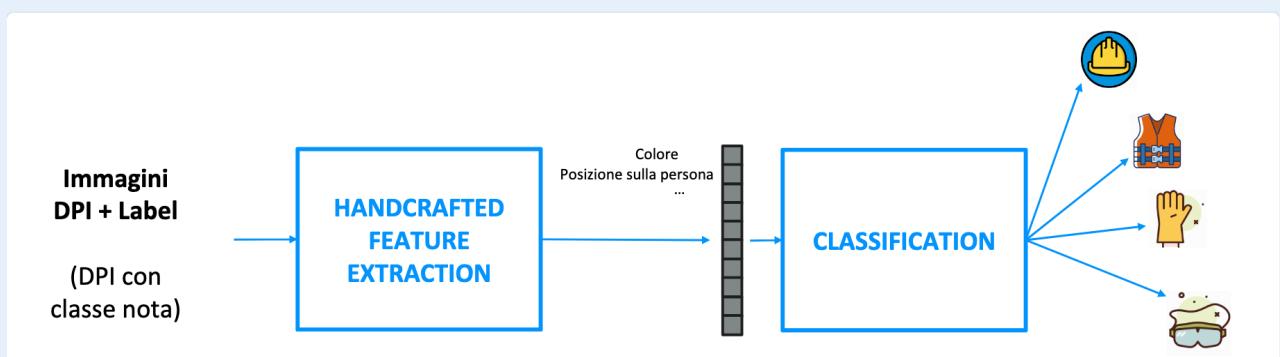
corrette, in particolare quelle che **massimizzano** le differenze tra pazienti diversi e **minimizzano** le differenze tra pazienti simili.

Dal momento che è necessario l'intervento umano, questo approccio è valido per **set ristretti di informazioni**. Un vantaggio deriva dal fatto che i risultati prodotti sono facilmente interpretabili e spiegabili, permettendo di tracciare l'intero percorso decisionale.

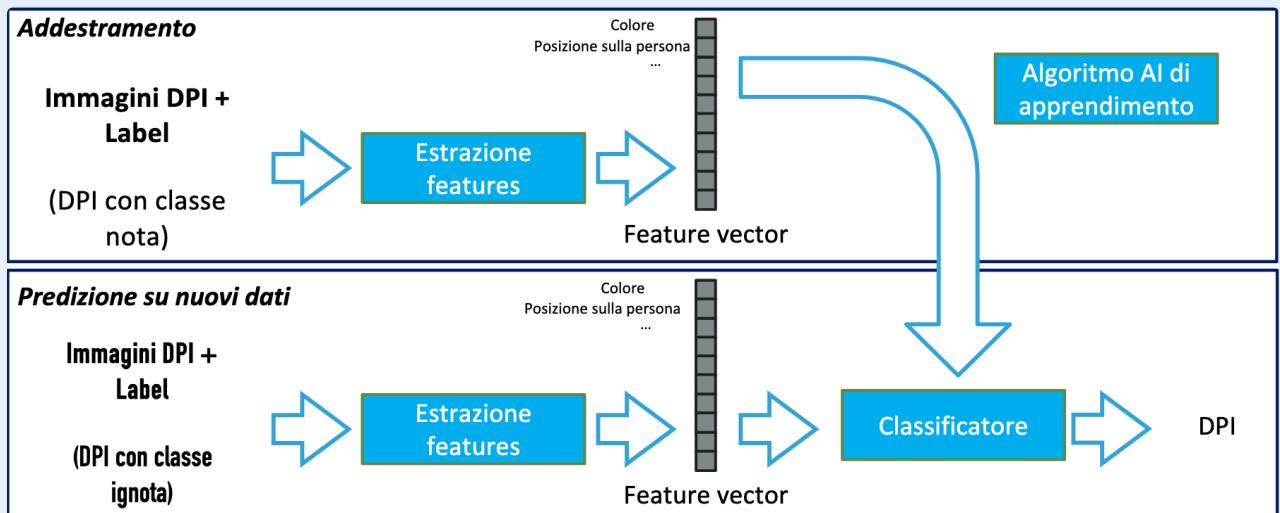
### Esempio



Consideriamo il problema di verifica della presenza di dispositivi di protezione individuale (DPI) indossati dagli operai: elmetto, maschera, gilet e guanti.



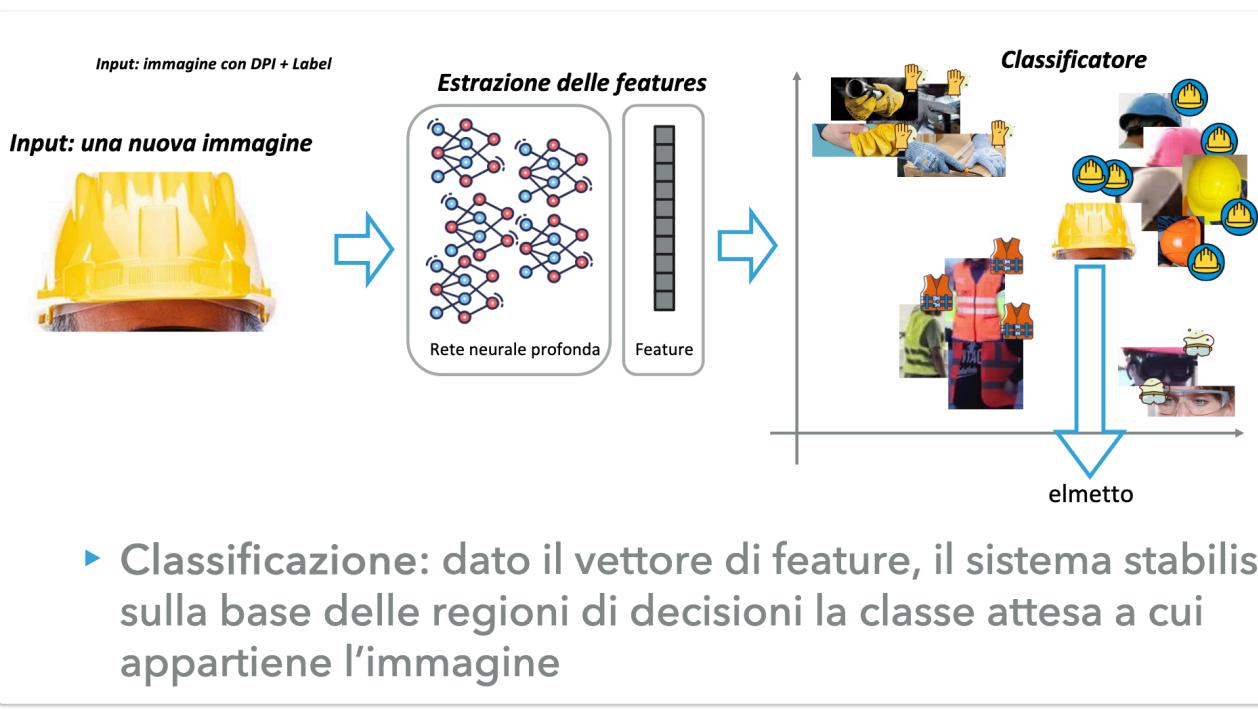
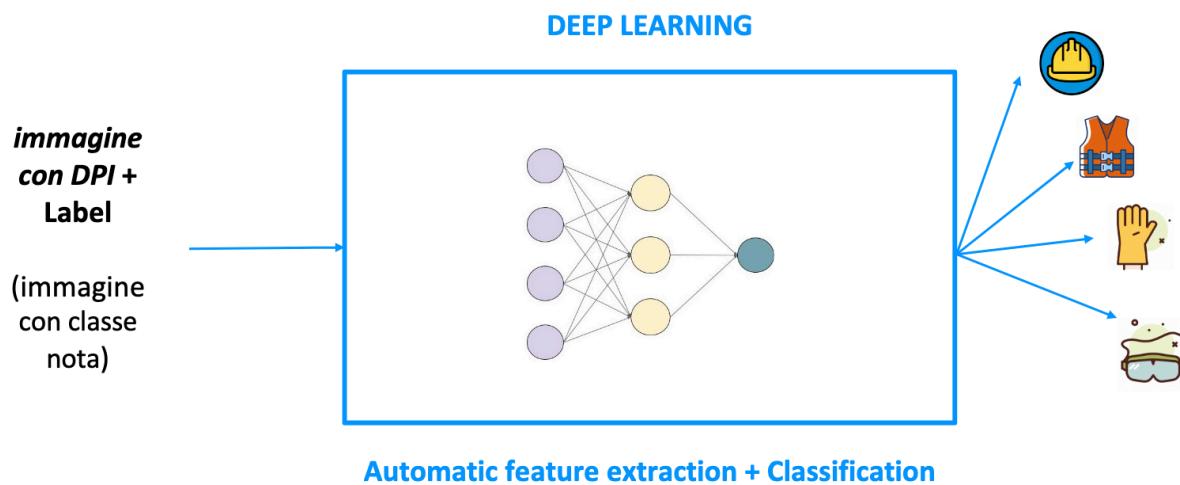
Il progettista ha il compito di individuare le **features** (come il colore del DPI, la posizione sulla persona, ecc...) che consentono poi alla macchina di identificare lo specifico dispositivo di protezione individuale, mediante la costruzione di un vettore di features.



Sostanzialmente si identifica il DPI mediante le sue caratteristiche, salvate all'interno del vettore delle features.

## 4.1.2 Deep Learning

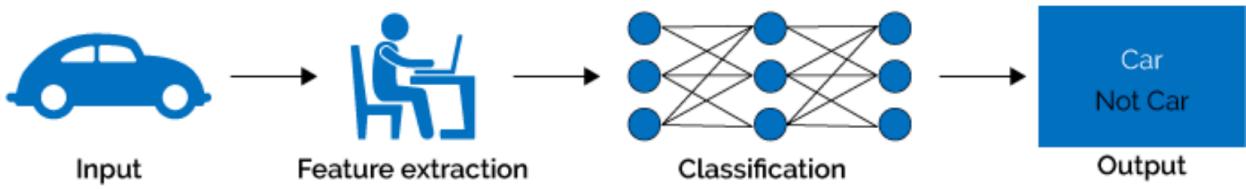
Con l'approccio moderno non è più necessario estrarre manualmente tali caratteristiche in quanto, gli algoritmi di **deep learning** imparano automaticamente quali sono le features più utili direttamente dai **dati grezzi**, durante la fase di **addestramento**.



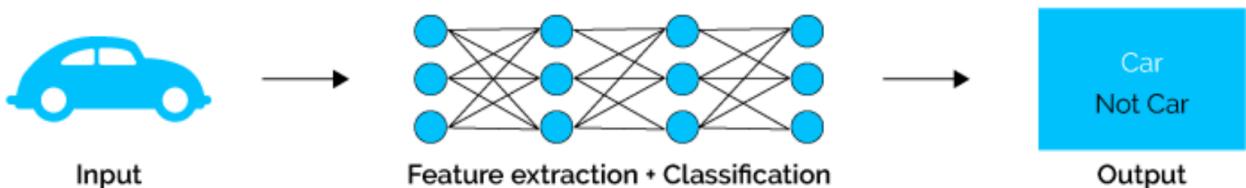
#### 4.1.2.1 Reti Neurali

Al giorno d'oggi le reti neurali artificiali sono il cuore del deep learning. Si tratta di algoritmi che effettuano l'apprendimento da esempi tentando di riprodurre il **comportamento del cervello umano**, queste reti sono composte da tanti "nodi" (neuroni artificiali) collegati tra loro e organizzati in strati.

# Machine Learning



# Deep Learning



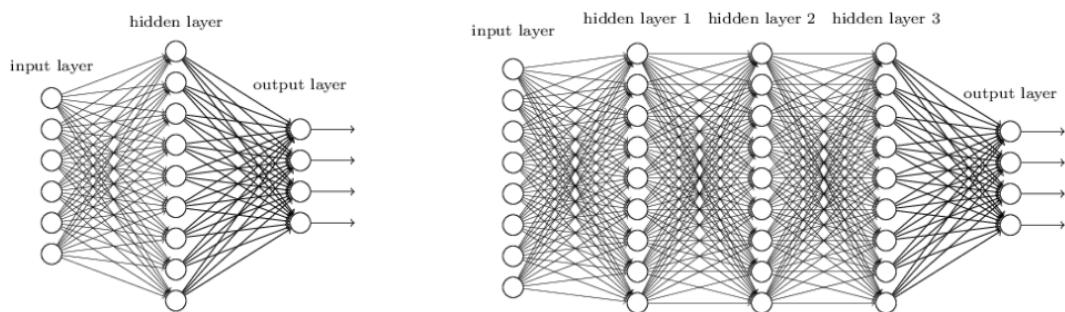
Con il deep learning le features vengono **apprese "automaticamente"** dai dati, con un processo automatico progressivo che costruisce una rappresentazione sempre più adeguata degli oggetti di interesse.

## LIMITAZIONI DELLE RETI NEURALI CLASSICHE

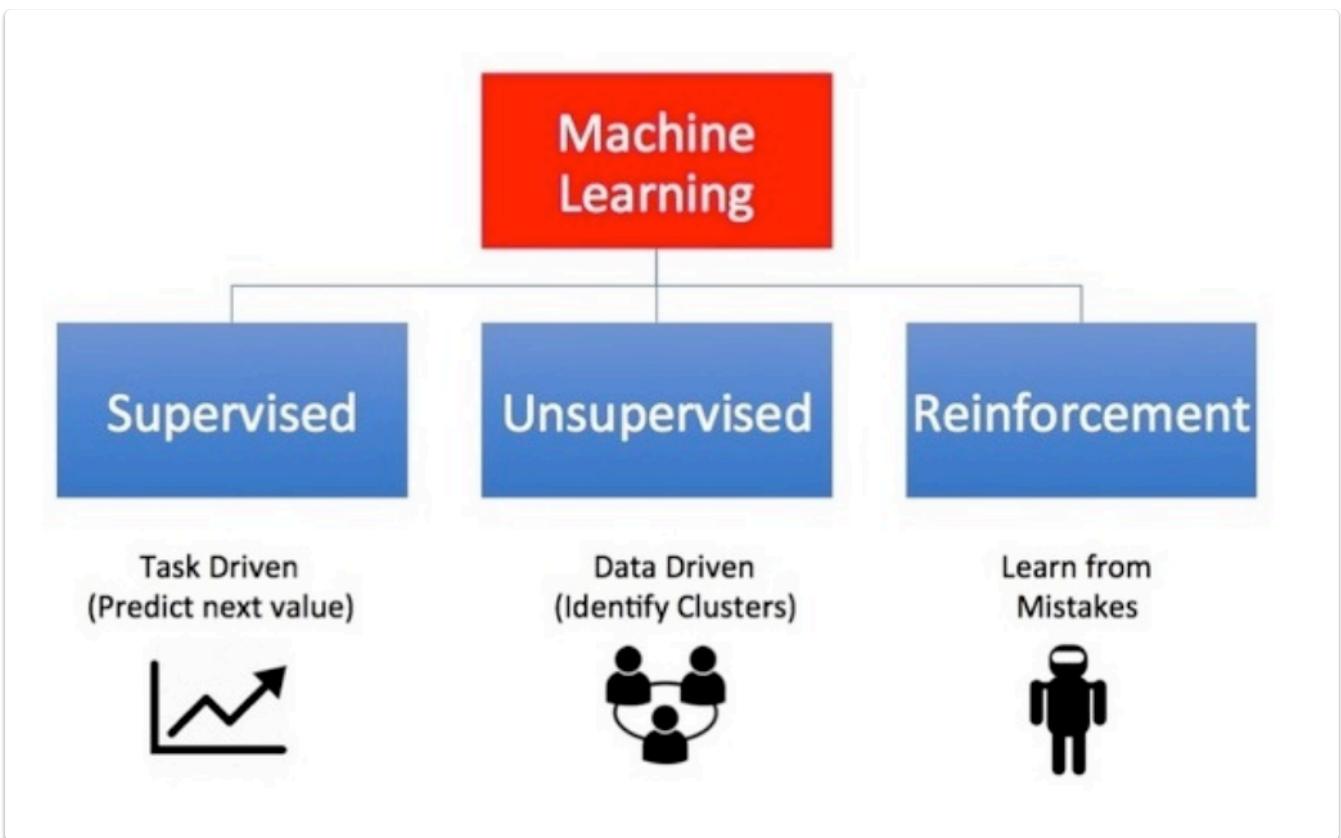
I difetti delle reti neurali usate negli anni '90 sono legati alla ridotta potenza dei calcolatori dell'epoca e a dataset di piccole dimensioni.

Reti con uno o due strati nascosti e basso numero totale di neuroni: incapacità di risolvere problemi molto complessi.

Con l'aumento della potenza di calcolo (e GPU) e con dataset di grandi dimensioni, è nato il popolarissimo deep learning.



### 4.1.3 Approcci all'apprendimento automatico



Le modalità di apprendimento sono diverse:

- **Supervised Learning:** il sistema intelligente apprende da **esempi** del problema in esame la cui **soluzione è nota**. I problemi di apprendimento supervisionato solitamente sono di 2 tipologie:
  - **Regessione:** l'output da riconoscere ha valori continui (stima del prezzo di una casa, stima dell'età di una persona, analisi finanziarie etc.);
  - **Classificazione:** l'output da riconoscere è una **classe** o categoria (riconoscimento del genere di una persona, sentiment analysis, classificazione di tumori etc.)
- **Unsupervised Learning:** il sistema intelligente impara a dividere i dati in gruppi (cluster) omogenei rispetto all'obiettivo dell'applicazione;
- **Reinforcement Learning:** sistema intelligente viene pre-addestrato per svolgere il compito, ma poi migliora nel tempo imparando dagli errori.

## 5 Architettura di un sistema intelligente

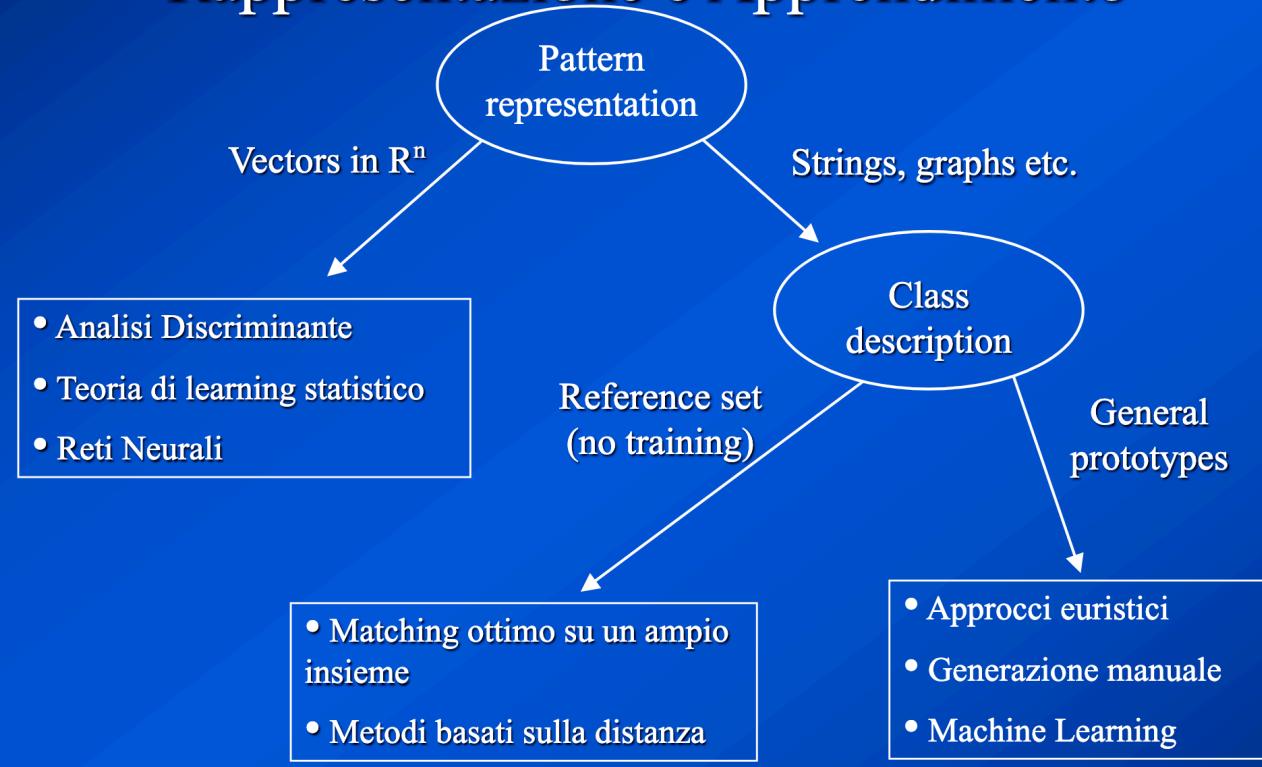


Le features rappresentano gli oggetti di interesse nel sistema intelligente e devono:

- **accumunare** tutti i campioni di una medesima classe;

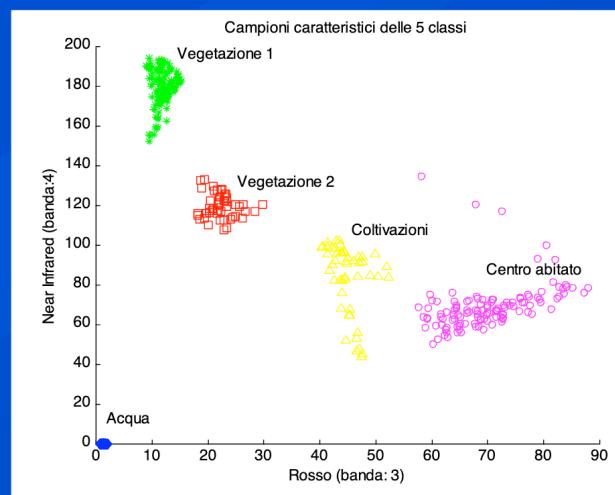
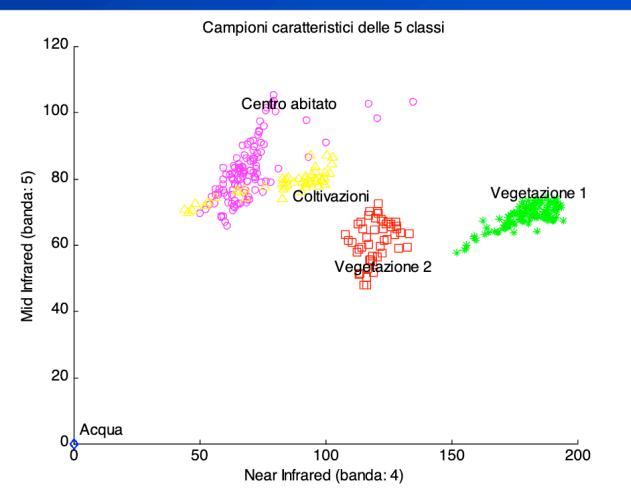
- discriminare campioni di classi diverse

# Rappresentazione e Apprendimento



Stabilite le features, un singolo oggetto del mondo reale viene rappresentato mediante un **vettore delle caratteristiche**: ad ogni oggetto corrisponde un vettore differente. A partire da tale vettore si costruisce uno **spazio vettoriale** in modo tale da poter rappresentare l'oggetto come un **punto** di tale spazio: se le features sono  $n$  si costruisce uno spazio  $n - dimensionale$ .

## Diverse rappresentazione con coppie di features



La modellizzazione di uno **spazio vettoriale** semplifica notevolmente la fase di **classificazione**: per verificare se un campione appartiene ad una determinata categoria, basta confrontarlo con campioni simili, andando a misurarne la **distanza (classificatore a minima distanza)**. Il sistema classifica quindi il campione basandosi sul campione più vicino.

Il **classificatore a minima distanza** tuttavia non basta per ottenere una **classificazione accettabile** dal momento che, anche due punti molto distanti (quindi molto probabilmente appartenenti a classi differenti) potrebbero essere comunque classificati come **simili**. È necessario pertanto introdurre delle **regioni di decisione**: se un campione non rientra nella regione di decisione stabilita per una determinata classe, allora il campione non viene classificato (**rigetto**).

## 5.1 Scelta delle features

---

La **scelta delle features** viene solitamente effettuata mediante l'ausilio dell'**analisi discriminante**, la quale consente di classificare un campione in gruppi e comprendere le differenze tra i vari gruppi. Lo scopo dell'analisi è **massimizzare la varianza tra i gruppi** e **minimizzarla** tra gli elementi dello **stesso gruppo**.

- Si parte con un training set molto **ricco** e **ridondante** di features
- L'analisi discriminante fornisce: dato un numero fissato di features  $n$ , il sottoinsieme di features che massimizza la percentuale (accuratezza di classificazione) sul training set;
- Si seleziona il più piccolo  $n$  che è vicino (1%) alla performance massima

In altre parole:

- **Partiamo dai dati di allenamento:** Abbiamo tanti esempi già etichettati, da cui il computer deve imparare.
- **Iniziamo con troppe informazioni:** Spesso abbiamo un sacco di dettagli (features), ma molti sono inutili o ridondanti.
- **Usiamo l'Analisi Discriminante per selezionare:** Questo metodo "**setaccia**" le informazioni. Per ogni numero di features ( $n$ ), trova il sottoinsieme che permette di fare le previsioni più accurate sui nostri dati di allenamento.
- **Scegliamo il gruppo più piccolo e quasi perfetto:** Alla fine, prendiamo il set di features più **piccolo** che raggiunge una performance di previsione praticamente identica a quella massima (tollerando una piccola differenza, tipo l'1%). Questo rende il modello più semplice e robusto.

## 5.2 Classificatori

---

Nel campo dell'intelligenza artificiale, in particolare nel riconoscimento di pattern (come immagini, video o audio), i **classificatori** sono strumenti fondamentali che imparano a distinguere e categorizzare i dati. Esistono diverse tipologie di classificatori, tra cui spiccano i **classificatori statistici** e i **classificatori neurali**, che adottano approcci differenti per risolvere problemi simili.

### 5.2.1 Classificatori statistici

I **Classificatori Statistici** si basano su principi della **statistica e della probabilità**. Essi cercano di modellare matematicamente le relazioni tra i dati e le diverse categorie, spesso assumendo che i dati seguano determinate **distribuzioni**.

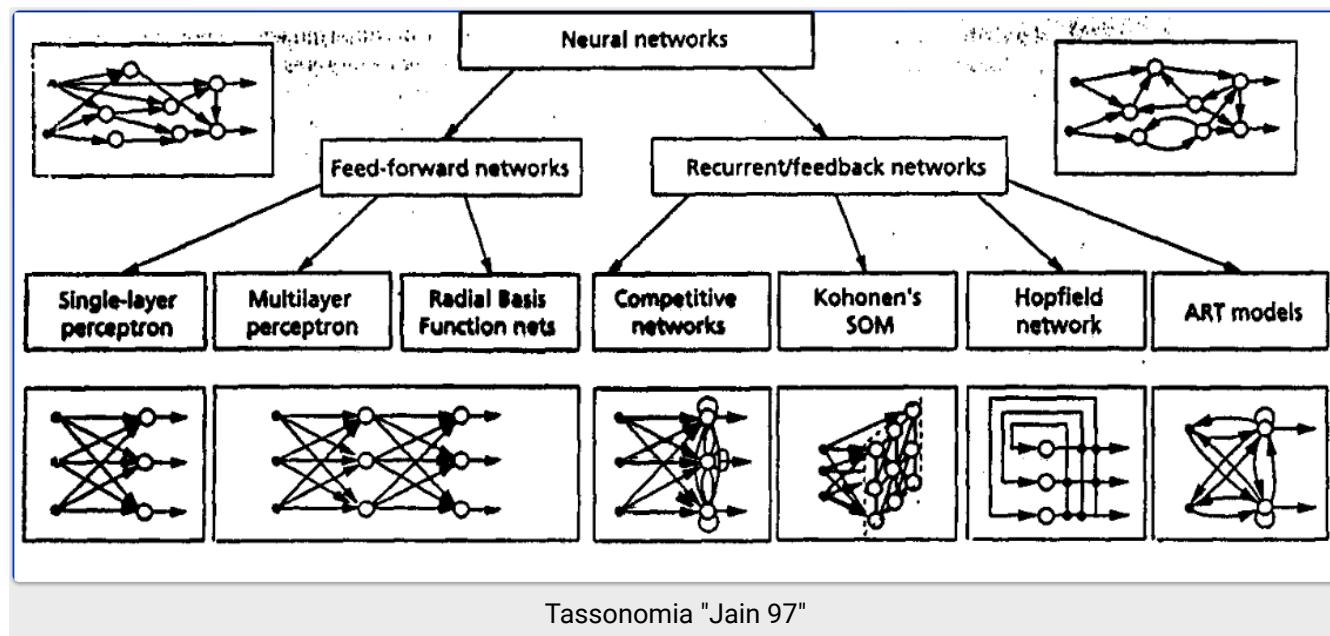
L'obiettivo è stimare la probabilità che un'osservazione appartenga a una data classe. Metodi come l'**Analisi Discriminante** (vista precedentemente per la selezione di features), la Regressione Logistica o i classificatori Bayesiani rientrano in questa categoria. Tendono ad essere più "**trasparenti**" e interpretabili, il che significa che è spesso più semplice capire perché prendono una certa decisione.

Tuttavia, possono incontrare **difficoltà** con **relazioni molto complesse** e non lineari nei dati, o richiedere un'attenta pre-elaborazione.

### 5.2.2 Classificatori neurali

Al contrario, i **Classificatori Neurali** (o Reti Neurali Artificiali) sono modelli ispirati al **funzionamento del cervello** biologico. Sono costituiti da "neuroni" interconnessi che elaborano le informazioni in strati successivi. La loro forza risiede nella capacità di apprendere e modellare **relazioni estremamente complesse e non lineari**.

all'interno dei dati, rappresentando gerarchie di caratteristiche sempre più astratte. Sebbene siano incredibilmente potenti, specialmente con grandi volumi di dati (come immagini e suoni), le reti neurali sono spesso considerate delle "scatole nere", in quanto il **processo decisionale** interno è **meno immediato** da interpretare.



Tassonomia "Jain 97"

- **Modalità di addestramento**
  - Supervised
  - Unsupervised
- **Leggi di apprendimento**
  - Coincidence learning
  - Competitive learning
  - Performance learning
  - Filter learning
  - Spatiotemporal learning

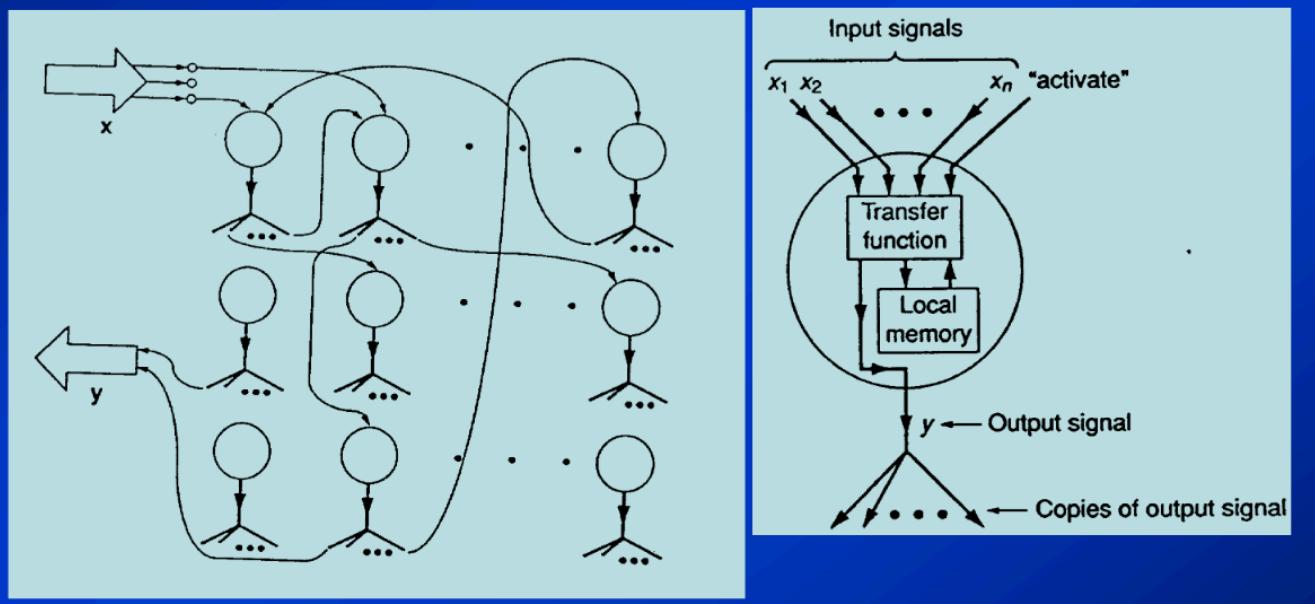
### ⌚ Definizione formale di Rete Neurale (Hecht-Nielsen)

1. Una **Rete Neurale** è una struttura parallela che **processa informazioni** distribuite;
2. Consta di **elementi di processazione** (Dotati di una memoria locale) interconnessi tramite **canali** (detti connessioni) che trasmettono segnali unidirezionali
3. Ogni neurone ha una **singola connessione** di uscita che si dirama in un certo numero di connessioni collaterali; ognuna di queste **trasporta il segnale d' uscita** del neurone
4. La computazione di ciascun neurone deve essere **completamente locale**; cioè deve dipendere solo dai valori correnti dei segnali d'ingresso che arrivano al neurone tramite opportune connessioni e dai valori immagazzinati nella memoria locale del neurone.

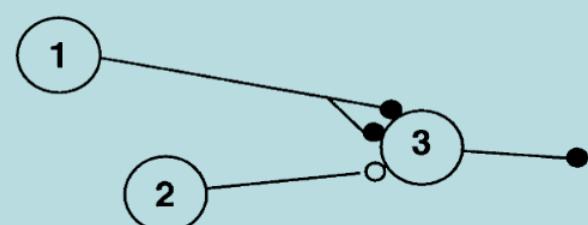
#### 5.2.2.1 Architettura di una rete neurale

# Una possibile Architettura

# Il generico neurone



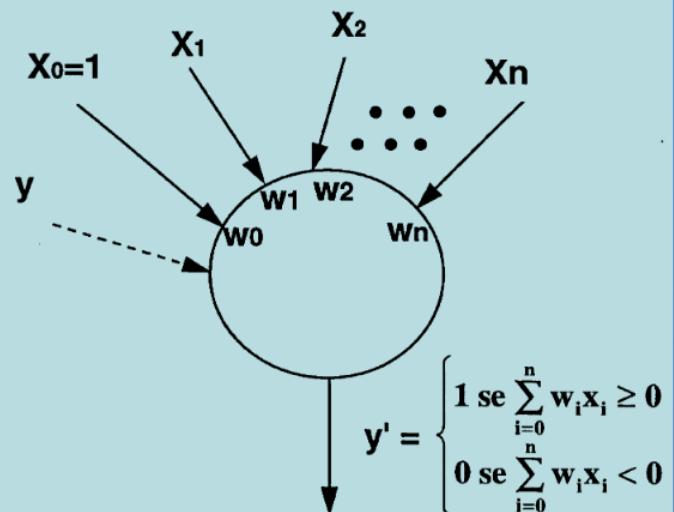
## 5.2.2.1.1 Percetroni singoli



$$N_3(t) = N_1(t-1) \& (\text{not } N_2(t-1))$$

Il modello di McCulloch-Pitts (1943)

Il Percettrone di Rosenblatt (1957)

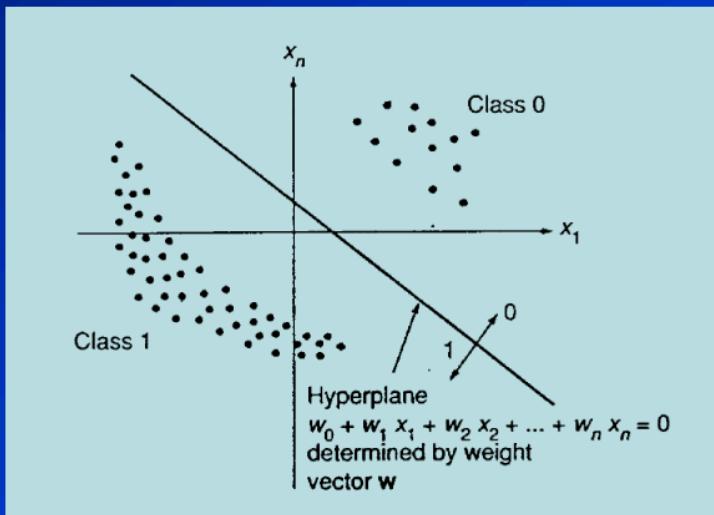


Il **percettrone di Rosembatt** è un modello di neurone artificiale singolo ed è stato il primo classificatore neurale capace di apprendere da esempi.

Un percettrone (o un generico classificatore lineare), divide lo spazio dei dati in **iperpiani** per effettuare la classificazione.

## Gli Iperpiani

Un problema semplice:  
la OR



<b>x<sub>1</sub></b>	<b>x<sub>2</sub></b>	<b>y</b>
0	0	0
0	1	1
1	0	1
1	1	1

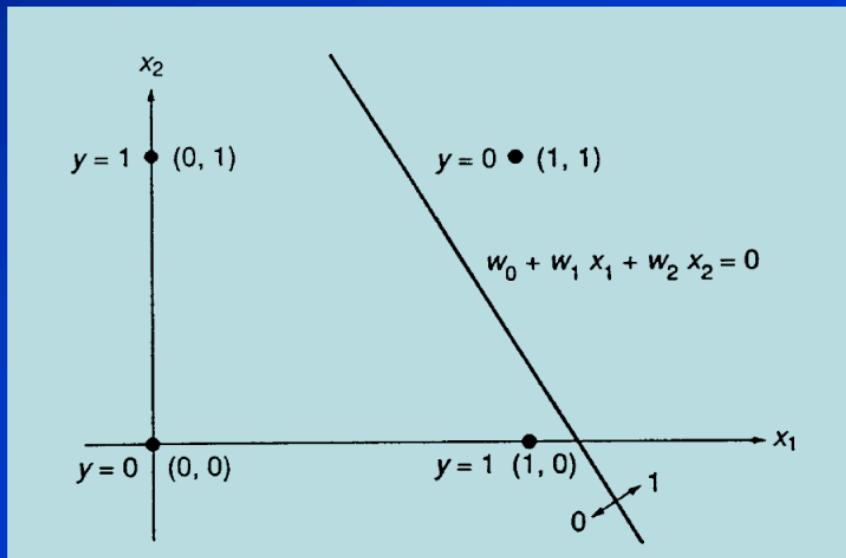
Legge di apprendimento

$$\mathbf{W}^{\text{new}} = \mathbf{W}^{\text{old}} + (\mathbf{y} - \mathbf{y}') \mathbf{x}$$

Condizione iniziale:  
 $\mathbf{W}^0 = [0.5, 0, 0]$

In questo esempio, l'iperpiano (la retta obliqua), **separa** gli elementi delle due classi. Tutti i punti che cadono da un lato della linea appartengono ad una classe, tutti quelli che cadono dall'altro lato appartengono all'altra.

## La critica di Minsky (1969): il caso della XOR



Impossibility of computing the XOR function with a perceptron. In order for the perceptron to compute XOR, a line (the hyperplane corresponding to the weights  $w_0$ ,  $w_1$  and  $w_2$ ) must be found that puts points  $(0,0)$  and  $(1, 1)$  on one side of the line and  $(0, 1)$  and  $(1, 0)$  on the other side of the line. This is obviously impossible.

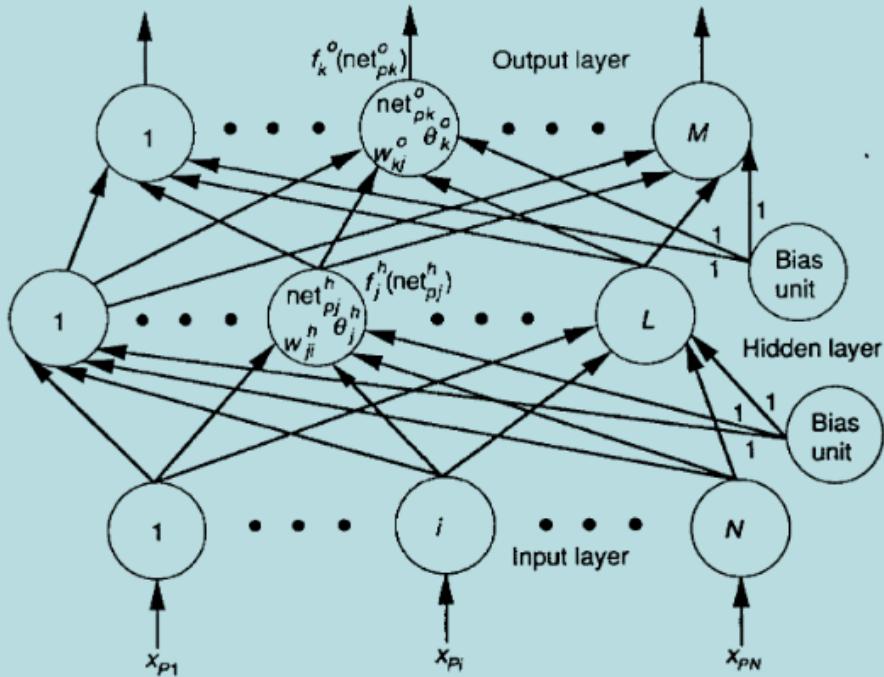
La **critica di Minsky** evidenzia i **limiti** dei perceptri singoli, i quali sono ingrado di classificare solamente **funzioni linearmente separabili** (come la funzione **OR**). La figura mostra che é impossibile tracciare una singola linea retta (un singolo **iperpiano**) che separi correttamente i punti  $(0, 1)$  e  $(1, 0)$  che danno come

risultato 1, dai punti (0, 0) e (1, 1) che danno come risultato 0.

Questo significa che un **percettrone singolo** non può risolvere problemi che non sono linearmente separabili.

#### 5.2.2.1.2 Percettroni Multilivello

## Il Percettrone Multilivello



$$\begin{aligned} \text{net}_{pj}^h &= \sum_{i=1}^N w_{ji}^h x_{pi} + \theta_j^h & \text{net}_{pk}^o &= \sum_{j=1}^L w_{kj}^o i_{pj} + \theta_k^o \\ i_{pj} &= f_j^h(\text{net}_{pj}^h) & o_{pk} &= f_k^o(\text{net}_{pk}^o) \end{aligned}$$

La figura mostra l'architettura base di un **Percettrone Multilivello (MLP)**, ovvero come è strutturata e connessa una rete neurale di questo tipo. Una MLP é composta da piú strati di neuroni artificiali organizzati gerarchicamente:

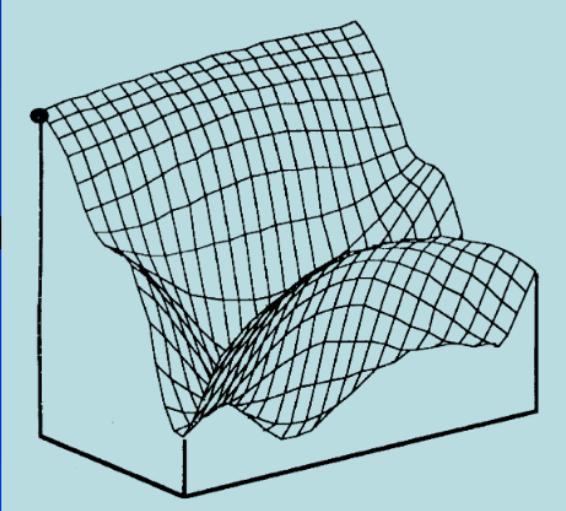
1. **Input Layer**: i dati in ingresso (le features) vengono ricevuti e presentati alla rete;
2. **Hidden Layer**: strati collocati tra quello di input e di output. Si occupano dell'elaborazione dei dati e dell'apprendimento di relazioni non lineari. Ogni nodo in uno strato nascosto riceve input dai nodi dello strato precedente e trasmette un output ai nodi dello strato successivo;
3. **Output Layer**: ultimo strato della rete dove viene riprodotto il risultato finale.

Le frecce tra i nodi rappresentano le **connessioni** tra i neuroni. Ogni connessione ha un **peso** associato, che determina l'importanza del segnale che viaggia lungo quella connessione. Questi pesi, insieme ai **bias** (nodi aggiuntivi che forniscono un valore costante, visibili come "*Bias unit*" negli strati nascosto e di output), sono i parametri che la rete impara a regolare durante l'addestramento per svolgere il suo compito.

#### 5.2.2.1.2.1 Addestramento

# Il Percettrone Multilivello: Addestramento

## Superficie d'Errore



## Aggiornamento dei pesi

$$E = \frac{1}{2} \sum_j (D_j - O_j)^2$$
$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

L'addestramento di una MLP è il processo attraverso cui la rete impara a svolgere il suo compito di classificazione o regressione, regolando i suoi **pesi** e **bias**. Questo avviene tramite l'algoritmo di **Back-Propagation**, che implementa una forma di **discesa del gradiente**.

- **Superficie d'Errore:** Concettualmente, l'addestramento può essere visualizzato come la ricerca del punto di minimo globale su una "superficie d'errore". Questa superficie rappresenta come l'errore del modello varia in funzione dei suoi pesi. L'obiettivo è trovare la configurazione di pesi che minimizza l'errore.
- **Aggiornamento dei pesi:** I pesi della rete vengono iterativamente modificati in base all'errore calcolato tra l'output predetto e quello desiderato. La formula

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial \sigma_{w_{ij}}}$$

indica che ogni peso è **aggiustato in direzione opposta** al **gradiente dell'errore** (cioè, verso la riduzione dell'errore), con un passo controllato dal tasso di apprendimento ( $\eta$ ).

### 5.2.2.1.2.2 Algoritmo Back-Propagation

# L'Algoritmo Back-Propagation

## Algoritmo supervisionato (Rumelhart, Hinton e Williams, 1986)

```
PROCEDURE Back-Propagation learning 0
BEGIN
    REPEAT
        E = 0;
        FOR r := 1 TO Ntr
        BEGIN
            forward();
            valutazione di E;
            backward();
        END;
    UNTIL (condizione di stop(E))
END;
```

L'**Algoritmo Back-Propagation** (retropropagazione dell'errore) è il metodo standard per **addestrare reti neurali artificiali feed-forward** come i Percetroni Multilivello (MLP). È un algoritmo di **apprendimento supervisionato**, il che significa che richiede un training set con input e corrispondenti output corretti.

Il suo obiettivo è **minimizzare l'errore** tra l'output generato dalla rete e l'output desiderato, regolando iterativamente i pesi e i bias delle connessioni tra i neuroni.

Il processo si svolge in cicli ripetuti (epoch), per ogni esempio del training set:

### 1. Fase Forward (Propagazione in avanti):

- L'**input** viene **presentato alla rete** e i segnali si **propagano** attraverso tutti gli strati, calcolando gli **output** di ciascun neurone fino a generare l'**output finale della rete**.

### 2. Calcolo dell'Errore:

- L'**output finale** della rete viene **confrontato** con l'**output corretto desiderato** (dal training set). Viene quindi calcolato un **valore di errore**.

### 3. Fase Backward (Retropropagazione dell'Errore):

- L'**errore** calcolato viene "**propagato all'indietro**" attraverso la rete, dagli strati di output a quelli di input. Durante questa fase, si calcola come ogni peso e bias ha contribuito all'errore totale.

### 4. Aggiornamento dei Pesi:

- Basandosi su questi calcoli (usando la discesa del gradiente) i pesi e i bias vengono leggermente modificati per ridurre l'errore nel prossimo ciclo. Il parametro  $\eta$  (tasso di apprendimento) controlla l'entità di queste modifiche.

Questo ciclo si ripete finché l'**errore** della rete non raggiunge una **soglia accettabile** o finché non viene soddisfatta un'altra condizione di **stop**. L'efficacia della Back-Propagation sta nella sua **capacità di regolare** sistematicamente tutti i **pesi** della rete per apprendere relazioni complesse dai dati.

#### 5.2.2.1.2.3 Regioni di decisione

# Il Percettrone Multilivello : Regioni di decisione

Structure	Type of Decision Regions	Exclusive-OR Problem	Classes with Meshed Regions	Most General Region Shapes
Single-layer	Half plane bounded by hyperplane			
Two-layers	Convex open or closed regions			
Three-layers	Arbitrary (Complexity limited by number of nodes)			

Lippman, 1987

Prof. Mario Vento, Università di Salerno

Le **regioni di decisione** sono le porzioni dello spazio delle feature che un classificatore associa a una specifica classe. La complessità di queste regioni dipende dall'architettura della rete:

- **Single-layer (Percettrone singolo):** È in grado di definire solo **regioni di decisione lineari** (delimitate da un singolo iperpiano). Questo significa che può separare classi solo se sono linearmente separabili, come illustrato dal fallimento nella risoluzione del problema XOR.
- **Two-layers (MLP con uno strato nascosto):** L'aggiunta di un singolo strato nascosto (e funzioni di attivazione non lineari) permette alla MLP di creare **regioni di decisione convesse o concave**. Questo abilita la soluzione di problemi non linearmente separabili (come il XOR, dove può creare due iperpiani che si intersecano per definire la regione desiderata).
- **Three-layers (MLP con due strati nascosti o più):** Con due o più strati nascosti, le MLP possono formare **regioni di decisione arbitrarie e complesse** (connesse o disconnesse). Questo le rende capaci di modellare praticamente qualsiasi tipo di confine decisionale, anche i più intricati, limitato solo dal numero di neuroni negli strati nascosti e dalla quantità di dati di addestramento.

In sintesi, l'addestramento tramite discesa del gradiente permette alla MLP di imparare a mappare input complessi a output desiderati, e l'architettura a più strati le conferisce la capacità di definire regioni di decisione non lineari sempre più elaborate, superando le limitazioni dei classificatori lineari.

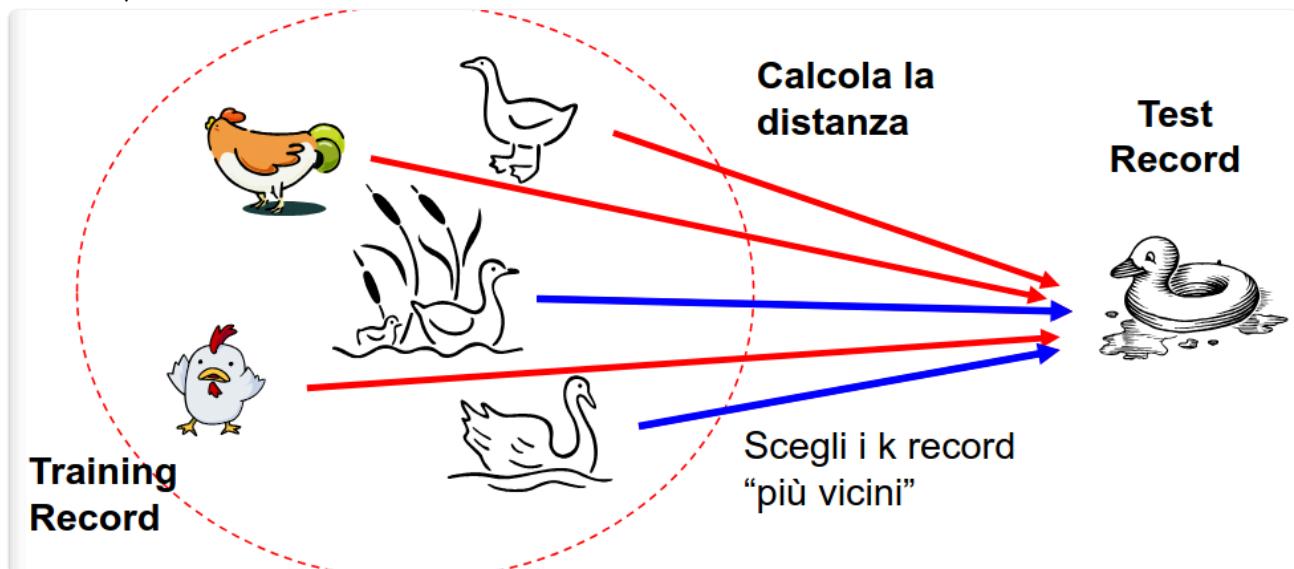
## 5.2.3 Classificatori \*NN

La classificazione viene effettuata utilizzando i **punti più vicini**.

Richiedono:

1. Un training set;
2. Una metrica per il calcolo della distanza tra i record;

3. Il valore  $k$ , ossia il numero dei vicini da considerare

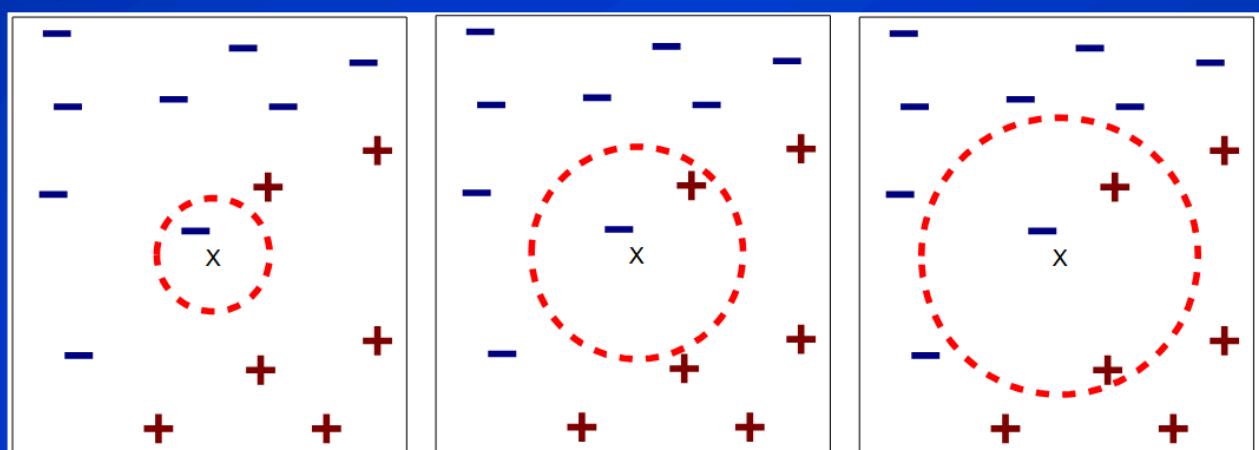


Il processo di **classificazione**:

- Calcola la **distanza** rispetto ai record nel training set;
- Identifica  $k$  nearest **neighbors**
- Utilizza le **label** delle classi dei nearest neighbor per determinare la classe del record sconosciuto (es. scegliendo quella che compare con maggiore frequenza)

## Definizione di Nearest Neighbor

• I  $k$  nearest neighbors di un record  $x$  sono i record del training set che hanno le più piccole  $k$  distance da  $x$



(a) 1-nearest neighbor

(b) 2-nearest neighbor

(c) 3-nearest neighbor

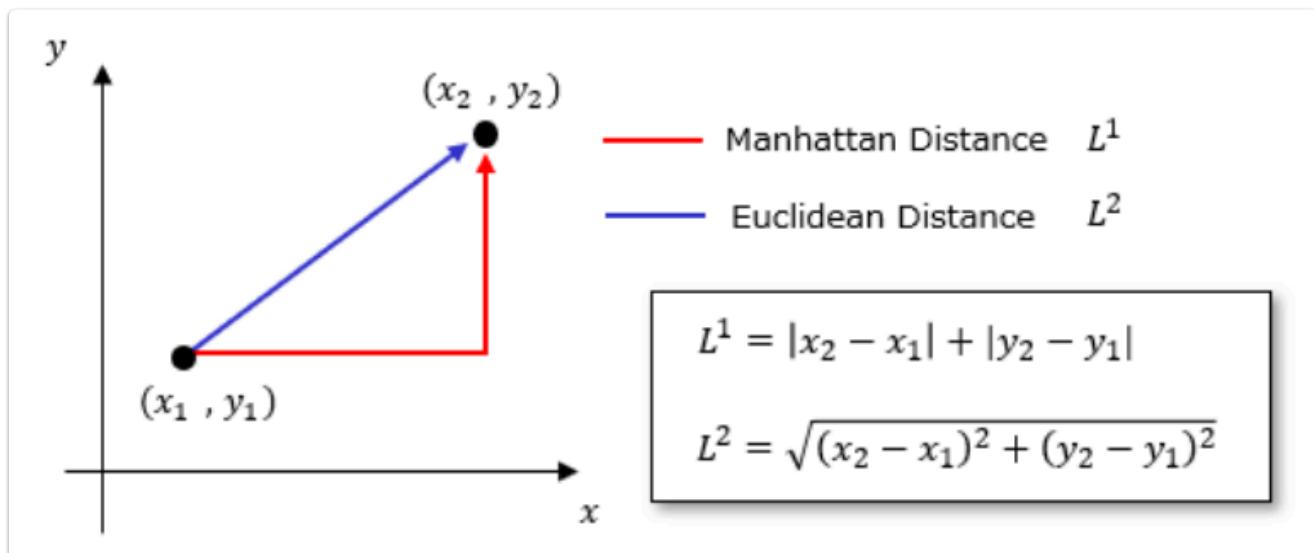
La  $X$  rappresenta il dato che si vuole classificare, + e - sono i dati già classificati.

Number of Columns: 2  
Largest Column: standard

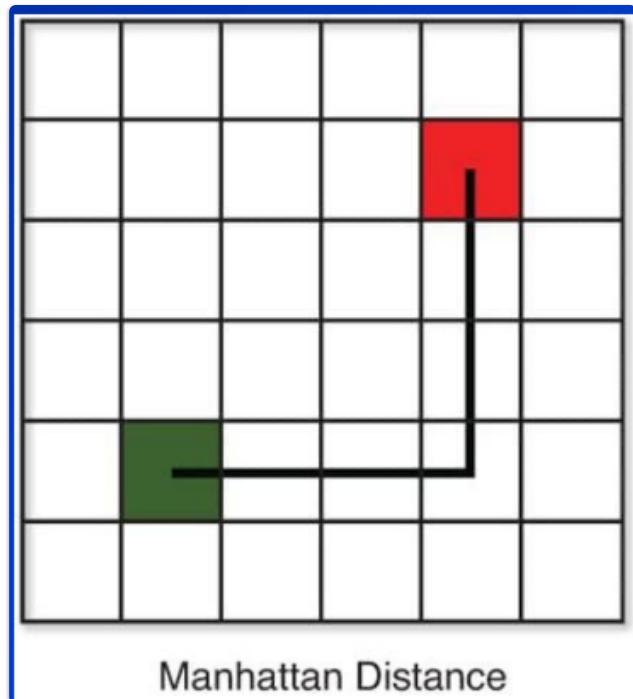
Altrimenti si può usare la distanza Manhattan:

$$L^1 = |x_2 - x_1| + |y_2 - y_1|$$

$$L^2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



-- column-break --

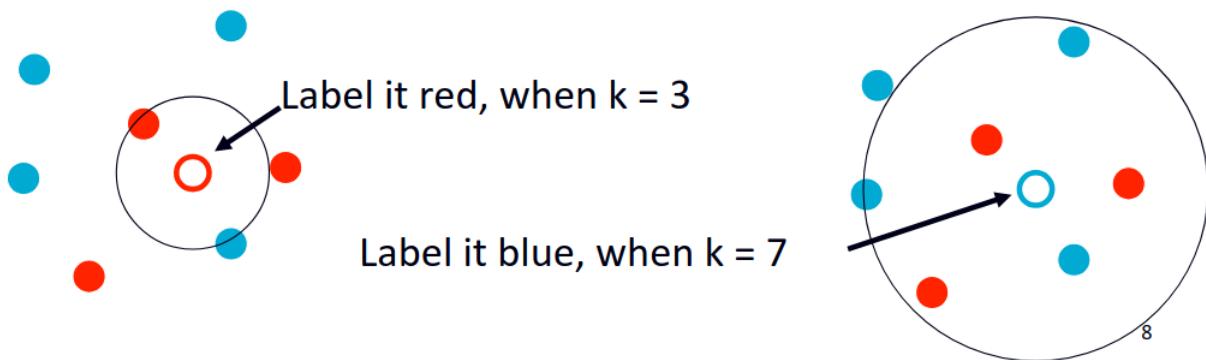


-- end-multi-column

La scelta del parametro  $k$  è molto importante in quanto:

- Se  $k$  è troppo **piccolo**, l'approccio è sensibile al **rumore**;

- Se  $k$  è troppo **grande**, l'intorno può includere **esempi** appartenenti ad **altre classi**;



## Pros & Cons

### Pro

- Non richiedono la costruzione di un modello
- Rispetto ai sistemi basati su regole permettono di costruire “contorni” (regioni di decisione) delle classi non lineari e sono quindi più flessibili

## Pros & Cons

### Cons

- Richiedono una misura di similarità o distanza per valutare la vicinanza
- Richiedono una fase di pre-processing per normalizzare il range di variazione degli attributi (Per operare correttamente anche con attributi che non hanno la stessa scala di valori )
- La classe è determinata localmente e quindi è suscettibile al rumore dei dati
- Sono molto sensibili alla presenza di attributi irrilevanti o correlati che falseranno le distanze tra gli oggetti
- Il costo di classificazione può essere elevato e dipende linearmente dalla dimensione del training set

Se  $k = 1$  si parla di classificatore **NN**, altrimenti si parla di classificatore **KNN - K-Nearest Neighbor**.

## 5.2.4 Confronto NN e KNN

# 6 Introduzione alla programmazione logica

La programmazione logica è un **paradigma di programmazione** che si basa sul **ragionamento logico** per risolvere problemi, utilizzando un modello dichiarativo in cui la conoscenza viene rappresentata tramite fatti e regole logiche. In questo approccio, il **programmatore** descrive la **struttura logica del problema**, mentre il controllo dell'esecuzione è demandato all'**interprete** del linguaggio.

I linguaggi di programmazione logica più noti sono **Prolog** e Datalog; Prolog, in particolare, è molto usato nell'intelligenza artificiale per la rappresentazione della conoscenza, la deduzione e l'elaborazione del linguaggio naturale.

In un programma logico, la computazione avviene **applicando il ragionamento logico** alle **informazioni note**: si parte da una **base di conoscenza** (fatti e regole) e si pongono delle **domande (query)** al sistema, che risponde deducendo nuove informazioni tramite **inferenza**.

La **programmazione logica** si basa su **regole logiche** invece che su sequenze di comandi. Si dichiara *cosa* è vero, non *come* fare le cose.

## 6.1 Sistemi Rule-Based

---

La conoscenza è rappresentata tramite regole del tipo *if...then* (premessa → conclusione), applicando tecniche di forward e backward chaining per inferire nuove informazioni. La base di conoscenza è composta da **fatti e regole**, e il **motore inferenziale** applica le regole attivate dai dati presenti nel sistema.

Il sistema formale è espresso come un insieme di regole nella forma *if (premessa) then(conclusione)*, la conclusione è vera se lo è anche la premessa. Le due parti sono dette **precondizione** e **postcondizione**.

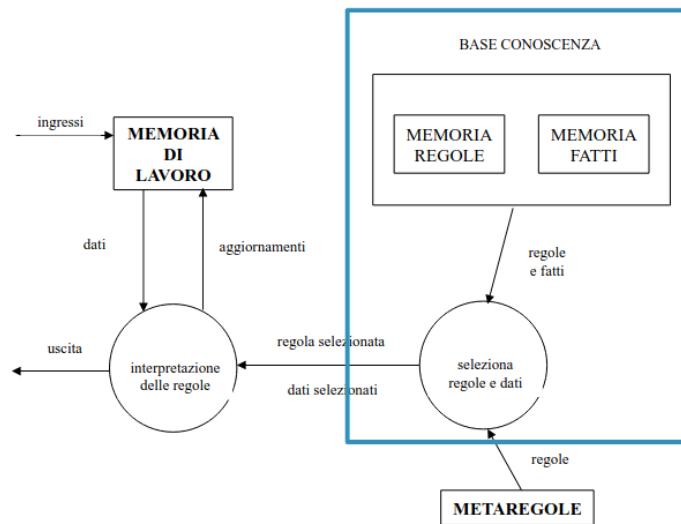
Il **motore inferenziale** esamina il database dinamico (KB) per determinare le **regole attivate** ed **applica** ciascuna di esse, aggiungendo la relativa conclusione al database. Questo approccio è detto **forward chaining**.

Altrimenti è possibile inserire nel db il goal e il motore inferenziale cercherà automaticamente in esso tutte le regole attivate quando sono vere le **postcondizioni** e metterà nella KB le relative **postcondizioni**.

**La base di conoscenza** è costituita di **fatti e regole**

**Il blocco di selezione regole e dati** percorre per intero la base di conoscenza e realizza l'**unificazione**

- Nota in un dato istante la memoria di lavoro e selezionato uno dei teoremi, determina quali regole siano ad esso applicabili.



## METAREGOLE: UN ESEMPIO

Nel caso di un sistema formale, supposto ad esempio che in ogni istante esistano 500 regole applicabili ai teoremi, possiamo immaginare che esse siano suddivise in più gruppi di regole

- Es: 2, il gruppo A e il gruppo B;

Il sistema non cerca di applicarle tutte, ma le metaregole gli suggeriscono di provare innanzitutto con A perché l'utilizzo di B non porta a risultati utili.

Una metaregola dunque inibisce un gruppo di regole a favore di un altro

## 6.2 Sistemi logic-based

Sono un'estensione dei rule-based, utilizzando la logica matematica (clausole, connettivi logici come and, or, not) per rappresentare la conoscenza.

## 6.3 Introduzione a Prolog

**Prolog** è un linguaggio di **programmazione dichiarativa** basato sulla logica, il cui nome deriva da **PROgramming in LOGic**. Nato nei primi anni '70 come strumento per l'elaborazione automatica del linguaggio naturale. Da allora si è affermato in molte aree dell'intelligenza artificiale, come la rappresentazione della conoscenza, i sistemi esperti, il ragionamento automatico, e il calcolo simbolico.

### 6.3.1 Paradigma dichiarativo

A differenza dei linguaggi imperativi (come C, Java o Python), in cui si definisce *come* deve essere eseguito un compito tramite sequenze di istruzioni, **Prolog permette di dichiarare che cosa è vero in un dominio**, attraverso fatti e regole. L'esecuzione di un programma avviene per mezzo di un **motore inferenziale** che tenta di soddisfare interrogazioni (query) poste dall'utente, tramite **unificazione e ricorsione logica**.

:≡ Example

padre (mario, bruno).

### 6.3.2 Logica come base della programmazione

Prolog si basa su un sottoinsieme della **logica del primo ordine**, ovvero su un sistema formale che permette di esprimere fatti e inferenze del tipo:

Se  $X$  è padre di  $Y$ , e  $X$  è coniuge di  $Z$ , allora  $Z$  è madre di  $Y$ .

Tradotta nel linguaggio:

```
madre(Z, Y) :- coniuge(X, Z), padre(X, Y).
```

Si legge: "la testa è vera se il corpo è vero", ovvero " $Z$  è madre di  $Y$  se  $Z$  è coniuge di  $X$  e  $X$  è padre di  $Y$ ".

### 6.3.3 Meccanismo di inferenza

Il **motore inferenziale** di Prolog funziona cercando di soddisfare le query attraverso:

1. **Unificazione**: confronto tra termini (variabili, costanti, liste) per renderli uguali.
2. **Risoluzione**: ricerca di regole e fatti che permettono di soddisfare la query.
3. **Backtracking**: ritorno indietro in caso di fallimento di una scelta, per esplorare alternative.

Questo meccanismo è particolarmente adatto per problemi di ricerca, classificazione, ragionamento, linguaggio naturale, e analisi simbolica.

### 6.3.4 Meccanismo di unificazione

L'unificazione è il **processo con cui Prolog confronta due termini** (costanti, variabili, liste, strutture) e cerca di renderli **identici**, assegnando valori alle **variabili** se necessario.

*In pratica, è il modo in cui Prolog "capisce" se una regola può essere applicata a una query.*

Quando Prolog riceve una **query**, confronta (unifica) questa con i **fatti** o la **testa delle regole** nella base di conoscenza:

1. **Se i termini sono uguali**, l'unificazione ha successo.
2. **Se una delle parti è una variabile**, assume il valore necessario per far coincidere i due lati.
3. **Se i termini non sono compatibili**, l'unificazione fallisce.

```
?- X = pippo.          % X prende valore pippo
X = pippo.

?- pippo = pippo.      % già uguali → successo
true.

?- pippo = pluto.     % costanti diverse → fallisce
false.
```

## 6.4 Sintassi Base

Un programma Prolog è composto da:

- **Fatti:** affermazioni incondizionate, es: `padre(mario, bruno)`
- **Regole:** affermazioni condizionate, es: ``genitore(X,Y) :- padre(X,Y).`
- **Query:** richieste fatte al sistema per sapere se qualcosa è vero, es: `?- genitore(mario, Y).`

Le relazioni sono espresse mediante **predicati**, che associano un nome a un numero di argomenti. Un predicato è ciò che viene prima della coppia delle parentesi tonde, es: `padre`. Il predicato mette in relazione il primo oggetto con il secondo.

```
padre(mario,bruno).
```

È un **assioma** del sistema formale: è composto dal nome del predicato, con l'indicazione degli oggetti (costanti) su cui agisce, ed è chiuso da un punto.

## I PREDICATI

**Il sistema formale è composto di predicati**

**Ogni predicato è individuato dal proprio nome, ad ogni nome è associato un predicato.**

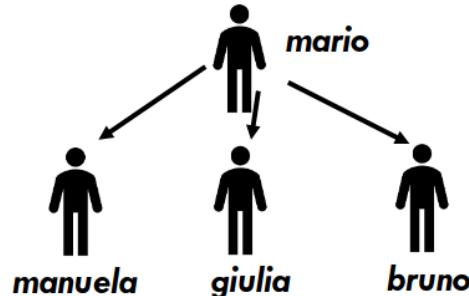
**Ogni volta che un predicato occorre nella base di conoscenza, intendiamo associare quella espressione formale ad un ‘fatto’ che è vero nel mondo reale**

Es:

**`padre (mario, manuela).`**

**`padre (mario, giulia).`**

**`padre (mario, bruno).`**



Le **variabili** sono identificate da nomi che iniziano con lettere maiuscole e sono contenitori logici che il sistema teneva di unificare con costanti o altri termini.

Ad esempio, nella query:

```
?- padre(mario, X).
```

Prolog cercherà nella base di conoscenza se c'è un fatto `padre(mario, Qualcosa)` e legherà `X` a quel valore (`X = bruno`).

Nel Prolog esistono due unità sintattiche: i fatti e le regole.

Fatti e regole sono **clausole**.

	Prolog	Logica
Implicazione	A :- B	$B \rightarrow A$
Congiunzione	A , B	$A \wedge B$
Disgiunzione	A ; B	$A \vee B$

#### 6.4.1 Fatti

La conoscenza viene rappresentata come una collezione di **fatti**. Un fatto è un'**affermazione incondizionata**, ossia vera nel nostro dominio:

```
padre(mario, giulia).  
padre(mario, bruno).
```

Qui stiamo affermando che `mario` è padre sia di `giulia` che di `bruno`. Ogni fatto termina con un punto.

#### 6.4.2 Variabili: placeholder logici

Le **variabili** sono elementi generici che possono essere associati (unificati) a costanti o ad altri termini durante il processo inferenziale.

```
?- padre(mario, X).  
X = giulia ;  
X = bruno.
```

- Le variabili iniziano con **lettera maiuscola** (es. `X`, `Y`, `Figlio`).
- Prolog tenta di **sostituire** le variabili con valori coerenti alla base di conoscenza per soddisfare una query.

#### 6.4.3 Regole: inferenza logica

Le **regole** rappresentano relazioni condizionate. Si leggono come: "*La testa è vera se il corpo è vero*".

```
genitore(X, Y) :- padre(X, Y).
```

Questa regola afferma che *X è genitore di Y se X è padre di Y*.

- `:-` si legge "se"
- La parte a **sinistra** di `:-` è detta **testa** o postcondizione (ciò che si vuole dimostrare).
- La parte a **destra** è detta **corpo** o precondizione (condizione da soddisfare).
- Più condizioni nel corpo sono unite con **virgole** (cioè congiunzione logica AND).



Warning

Le regole hanno sempre una sola postcondizione, ma possono avere una o più precondizioni legate in AND!

Si specificano prima le postcondizioni!

```
madre(X, Y) :- coniuge(Z, X), padre(Z, Y).
```

Interpretazione: *X è madre di Y se Z è coniuge di X e Z è padre di Y.*

## REGOLE

Le regole hanno sempre una sola postcondizione, ma possono avere una o più precondizioni (se le precondizioni sono più di una, si intendono legate in AND).

La postcondizione deve sempre essere specificata per prima.

**madre (X, Y) :-**

**coniuge (Z, X), padre (Z, Y).**

Se Z è coniuge di X e Z è padre di Y, allora X è madre di Y.

Nota: poiché nella base di conoscenza abbiamo espresso tutti gli oggetti 'figlio' rapportandoli al padre, possiamo dedurre, una volta noto il padre, chi sia la madre.

### 6.4.4 Liste

In Prolog, una **lista** è una struttura dati fondamentale. Si tratta di una **sequenza ordinata di elementi**, racchiusa tra **parentesi quadre** ( [ e ] ) e separati da **virgole**.

```
[1, 2, 3, 4]  
[a, b, c]  
[x, Y, 5, [a, b]]
```

Le liste contengono costanti simboliche, prive di una semantica predefinita. 1 non è un numero intero e "pippo" non è una stringa.

Il prolog distingue tra liste aperte e chiuse.

#### 6.4.4.1 Liste chiuse e aperte

Una lista è detta **chiusa** se tutti i suoi elementi sono esplicitamente definiti.

```
[a, b, c]
```

Una lista è **aperta** se non si conosce la sua coda (cioè se termina con una **variabile** che rappresenta il resto della lista).

[**a**, **b** | **X**]

Questa rappresenta una lista il cui primo elemento è **a**, il secondo è **b**, e **X** è una variabile che può essere legata ad **una qualunque altra lista**. La coda **X** rappresenta un'altra lista i cui elementi partecipano alla lista principale. Sostanzialmente il numero degli elementi non è specificato in partenza.

## LISTA APERTA

[**e1, e2, ..., ek** | **Y**]

La coda **Y** può anche non essere precisata nel momento in cui si definisce la lista aperta, oppure può essere nota solo in parte.

È proprio questo il grande vantaggio dell'uso delle liste in Prolog!

**Nota:** Nel caso più frequente, la ‘testa’ della lista (cioè che precede il simbolo | ) è composta da un solo elemento. Con la scrittura [**X** | **Y**], si intende dunque che **X** è un singolo oggetto e la coda **Y** è l’insieme di tutti gli altri elementi della lista, dal secondo in poi.

### 6.4.4.2 Struttura: testa e coda

Ogni lista può essere pensata come composta da:

- una **testa** ( Head ): il primo elemento,
- una **coda** ( Tail ): una lista contenente tutti gli elementi rimanenti.

[Head | Tail]

La notazione è letta come: una lista il cui primo elemento è **Head**, e il resto della lista è **Tail**.

[**1** | [**2, 3**]]

- Testa: 1
- Coda: [2, 3]

### 6.4.4.3 Unificazione nelle liste

È sempre possibile unificare una variabile con una lista chiusa. Non vale lo stesso per una costante!

**pippo  $\leftrightarrow$  [e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>] NO**

**X  $\leftrightarrow$  [e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>] SI**

Unificazione di una lista chiusa con un'altra lista chiusa. E' possibile sotto due condizioni:

- le due liste hanno la stessa cardinalità;
- gli elementi sono ordinatamente unificabili.

Due liste possono unificarsi solo se:

1. Hanno la stessa lunghezza (se sono chiuse);
2. Gli elementi **corrispondenti** sono unificabili tra loro;
3. Le **variabili** presenti possono essere assegnate in modo coerente;

[X, pippo, 3] **unifica con** [papa, Y, 3]

[X, pippo, 3] **non unifica con** [papa, X, 3]

poiché la variabile X dovrebbe assumere due valori differenti, cosa inammissibile. In seguito alla prima unificazione, la X diventa una variabile ‘bind’ (‘legata’ ad un valore: papa), per cui la seconda unificazione consisterebbe nell’unificare due costanti diverse: ‘papa’ e ‘pippo’.

[X, pippo, 3] **unifica con** [[1, 2], Y, 3]

dove gli elementi omologhi unificano nel seguente modo: X = [1, 2]; Y = pippo.

## UNIFICAZIONI NELLE LISTE APERTE

[pippo, 3, pluto] **unifica con** [ X | Y ]

dove X = pippo e Y = [3, pluto]. Si noti che Y è una lista.

[pippo, 3, pluto] **unifica con** [ X, Y | Z ]

dove Z è una lista contenente un solo elemento: Z = [pluto].

[ [pippo, 3], pluto] **unifica con** [ X, Y | Z ]

con X = [pippo, 3], Y = pluto e Z = [] (lista vuota).

# UNIFICAZIONI NELLE LISTE APERTE

[**pippo**] unifica con [X | Y]

dove Y è uguale alla lista vuota.

[**pippo**] non unifica con [X, Y | Z]

**È evidente che la lista chiusa deve avere una cardinalità almeno pari alla cardinalità della testa della lista aperta perché si realizzi l'unificazione.**

## 6.5 Ricorsione

Ogni predicato ricorsivo in Prolog ha due componenti fondamentali:

- **Caso base** – arresta la ricorsione
- **Caso ricorsivo** – chiama sé stesso su una struttura più semplice

Esempio: verificare se X è contenuto in una lista.

```
% Caso base: X è la testa della lista
membro(X, [X | _]).

% Caso ricorsivo: X non è la testa, cerca nella coda
membro(X, [_ | Tail]) :-  
    membro(X, Tail).
```

```
?- membro(3, [1, 2, 3, 4]).  
true.
```

```
?- membro(5, [1, 2, 3]).  
false.
```

Prolog tenta di **unificare** la lista [1, 2, 3] con [X | T]. Poi:

1. Controlla se X = 5 (cioè testa = 5)
2. Se no, richiama membro(5, T) con T = [2,3]
3. E così via, fino a trovare 5 oppure esaurire la lista

## 6.6 Eseguire un programma Prolog

1. Scrivere il programma in un file .pl
2. Avviare interprete SWI-Prolog con swipl

3. Caricare il file con `?- [nome\_file].`

4. Eseguire la query

Per verificare uno ad uno tutti i passi seguiti dal motore inferenziale per tentare di unificare il goal, si può utilizzare il comando `trace`.

```
[trace] ?- madre(sandra,X).  
  Call: (12) madre(sandra, _27512) ? creep  
  Call: (13) coniuge(_28808, sandra) ? creep  
  Exit: (13) coniuge(mario, sandra) ? creep  
  Call: (13) padre(mario, _27512) ? creep  
  Exit: (13) padre(mario, manuela) ? creep  
  Exit: (12) madre(sandra, manuela) ? creep  
X = manuela .
```

## IL COMANDO TRACE + LEASH

Attenzione! Il trace di default si ferma al primo predicato che trova.

Per non fermarsi, dobbiamo esplicitarlo con il seguente comando:

```
leash(-all)
```

```
[trace] ?- madre(sandra,X).  
  Call: (12) madre(sandra, _8988)  
  Call: (13) coniuge(_10280, sandra)  
  Exit: (13) coniuge(mario, sandra)  
  Call: (13) padre(mario, _8988)  
  Exit: (13) padre(mario, manuela)  
  Exit: (12) madre(sandra, manuela)  
X = manuela ;  
  Redo: (13) padre(mario, _8988)  
  Exit: (13) padre(mario, giulia)  
  Exit: (12) madre(sandra, giulia)  
X = giulia ;  
  Redo: (13) padre(mario, _8988)  
  Exit: (13) padre(mario, bruno)  
  Exit: (12) madre(sandra, bruno)  
X = bruno.
```

## 6.7 Prolog Cheat Sheet

### UNIFICAZIONE E CONFRONTO

=	Unificazione	X = 3	→ X = 3
\=	Non unificabile	3 \= 4	→ true
==	Uguaglianza senza unificazione	foo == foo	→ true
\==	Diversità senza unificazione	foo \== bar	→ true

### OPERATORI ARITMETICI

is	Valuta espressione	X is 2 + 3	→ X = 5
=:=	Uguaglianza numerica	3 =:= 3	→ true
=\=	Diversità numerica	4 =\= 5	→ true
<	Minore	2 < 5	→ true
>	Maggiore	5 > 2	→ true
=<	Minore o uguale	3 =< 3	→ true
=>	Maggiore o uguale	4 => 3	→ true

## CONFRONTI LESSICOGRAFICI

@<	Minore lessicografico	'a' @< 'z'	→ true
@>	Maggiore lessicografico	'z' @> 'a'	→ true
@=<	Minore o uguale (less.)	'a' @=< 'a'	→ true
@>=	Maggiore o uguale (less.)	'z' @>= 'z'	→ true

## CONTROLLO DI FLUSSO

,	AND logico	A, B	→ B solo se A è vero
;	OR logico	A ; B	→ B solo se A fallisce
->	If...Then	(A -> B)	→ se A allora B
-> ... ; ... If...Then...Else		(A -> B ; C)	→ se A allora B, altrimenti C
!	Cut	goal, !, more	→ blocca altre alternative

## LISTE

[H   T]	Testa e coda della lista	[a   [b, c]]	= [a, b, c]
[]	Lista vuota		
[X, Y]	Lista con due elementi	[1, 2]	
[_, X   _]	Secondo elemento è X		

## VARIE UTILI

writeln(X)	Stampa con newline	writeln(hello)
fail	Fallisce sempre	forza backtracking
true	Sempre vero	sempre ha successo
trace.	Debug passo passo	digita trace.
listing.	Elenca predicati	listing(mio_predicato).

☒ 1 Error in region

+

☒ 1 Error in region

+

☒ 1 Error in region

+