

UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso obbligatorio - 6 CFU

Introduzione intelligenza artificiale

Professore:

Prof. Alessio Micheli
Prof. Claudio Gallicchio

Autore:

Matteo Giuntoni

Contents

1 Introduzione	5
1.1 Obiettivi dell'IA	5
1.1.1 Modellare	5
1.1.2 Risultati	5
1.2 Storia dell'IA	5
1.3 Reti neurali	6
1.3.1 Deep Learning	6
2 Agenti intelligenti	7
2.1 Caratteristiche	7
2.1.1 Percezioni e azioni	7
2.2 Agente razionale	7
2.3 Ambienti	8
2.4 Programma agente	9
2.4.1 Tabella	9
2.4.2 Agenti reattivi	9
2.4.3 Agenti basati su modello	10
2.4.4 Agenti con obiettivo	11
2.4.5 Agenti con valutazione di utilità	11
2.4.6 Agenti che apprendono	11
2.4.7 Tipi di rappresentazione	12
3 Agenti risolutori di problemi	13
3.1 Processo di risoluzione	13
3.2 Formulazione del problema	13
3.3 Algoritmo di ricerca	13
3.4 Ricerca della soluzione	16
4 Stategia ricerca non informative	18
4.1 Ricerca in ampiezza (BF)	18
4.1.1 Analisi complessità spazio-temporale (BF)	19
4.2 Ricerca in profondità (DF)	20
4.2.1 DF ricorsiva	20
4.2.2 Ricerca in profondità limitata	21
4.3 Approfondimento iterativo (ID)	21
4.4 Direzione della ricerca	22
4.4.1 Ricerca bidirezionale	22
4.5 Ridondanze	22
4.6 Ricerca di costo uniforme (UC)	24
4.6.1 Analisi	25
4.7 Confronto strategie	25
5 Ricerca euristica	26
5.1 Algoritmo di ricerca Best-first	26
5.2 Algoritmo A	27
5.3 Algoritmo A^*	28
5.3.1 Ottimalità su A^*	29
5.4 Sotto-casi speciali: US e Greedy Best First	30
5.4.1 UC vs A^*	30
5.5 Costruire le euristiche di A^*	31
5.6 Inventare euristiche	32
5.6.1 Rillassamento problema	33
5.6.2 Massimizzazione di euristiche	33
5.6.3 Database con pattern disgiunti	33
5.6.4 Apprendimento dall'esperienza	34

5.6.5	Combinazione euristiche	34
5.7	Algoritmi evoluti basati su A^*	34
5.7.1	Beam search	34
5.7.2	IDA*	34
5.7.3	Best-first ricorsivo (BRFS)	35
5.7.4	A^* con memoria limitata (versione semplice)	36
6	Ricerca locale	37
6.1	Ricerca in salita (Hill climbing)	37
6.2	Tempra simulata	39
6.3	Ricerca local beam	40
6.4	Algoritmi generici/evolutivi	40
6.5	Spazi continui	41
6.6	Assunzioni sull'ambiente da considerare	42
6.6.1	Alberi di ricerca AND-OR	43
7	Agenti basati su coscienza	44
7.1	Introduzione	44
7.2	Agenti basati sulla conoscenza	46
7.3	Logica	46
8	Logica Proposizionale	48
8.1	Sintassi	48
8.2	Semantica	48
8.2.1	Semantica compositiva	48
8.2.2	Model checking	49
9	Calcolo proposizionale	50
9.1	Dimostrazione di teoremi	50
9.1.1	Equivalenza logica	50
9.1.2	Validità	50
9.1.3	Soddisfacenti	50
9.1.4	Dimostrazione per assurdo	51
9.1.5	Inferenza per PROP	51
10	Introduzione Machine Learning	52
10.1	Apprendimento supervisionato	53
10.2	Apprendimento non supervisionato	53
10.3	Model	53
10.4	Learning algorithms	54
11	Concept learning	56
11.1	Regole congiuntive	57
11.1.1	Rappresentazione di ipotesi	57
11.1.2	Ordinamento da generale a specifico	58
11.2	Algoritmo Find-S	59
11.3	Algoritmo List-Then eliminate	59
11.4	Algoritmo candidate elimination	60
11.5	Bias induttivo ed il suo ruolo	61
11.5.1	Unbiased learning	62
11.5.2	Bias induttivo	62

12 Modelli lineari	64
12.1 Regression	64
12.1.1 Univariate linear regression	64
12.1.2 Learning via LMS	65
12.2 Gradiant descent (ricerca locale)	66
12.3 Estensioni a l dati	67
12.4 Algoritmo iterativo di discesa del gradiante	69
12.5 Limitazioni	70
12.6 Linear basis expansion	71
12.7 Regularization	72
12.7.1 Limitazioni di una funzione a base fissa	73
12.8 Classification	74
12.9 Learning algorithm	75
13 Alberi decisionali	78
13.1 ID3 algorithm	78
13.2 Risolvere problemi ID3	81
13.2.1 Overfitting	82
13.2.2 Reduced-error pruning	83
13.2.3 Regole post-potatura	83
13.2.4 Attributi a valori continui	83
13.2.5 Gestire dati di training incompleti	84
13.2.6 Gestire attributi con costi differenti	84
14 Validation	85
14.1 Hold out	85
14.2 Un semplice meta-algoritmo	86
14.3 K-fold	87
14.4 SLT	88
15 SVM	90
15.1 Maximum margin classifier	90
16 Riassunto modelli	91

Introduzione all’Intelligenza Artificiale

Realizzato da: Matteo Giuntoni, Ghirardini Filippo

A.A. 2023-2024

1 Introduzione

1.1 Obiettivi dell'IA

1.1.1 Modellare

Modellare fedelmente l'essere umano:

- **Agire umanamente:** Test di Turing¹
- **Pensare umanamente:** modelli cognitivi per descrivere il funzionamento della mente umana

1.1.2 Risultati

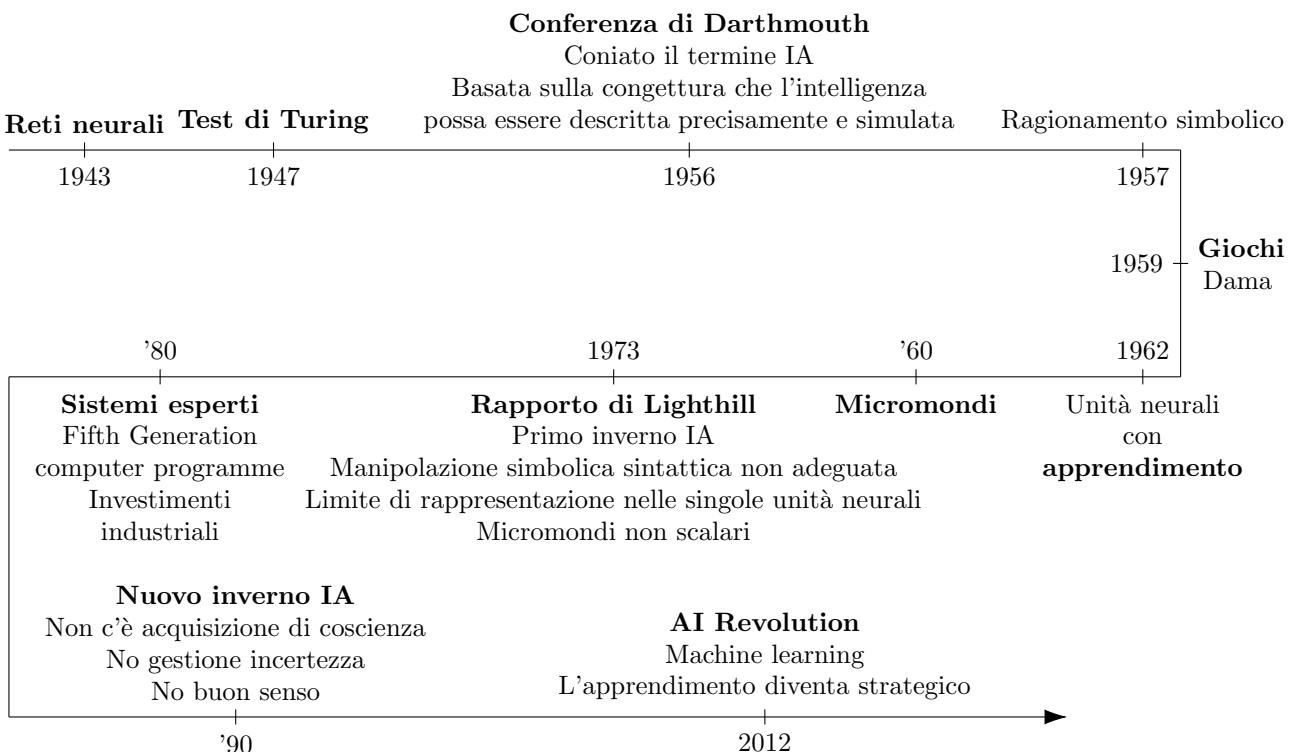
Raggiungere i risultati ottimali:

- Pensare razionalmente
- Agenti razionali: percepiscono l'ambiente, operano autonomamente e si adattano. Fanno la cosa giusta agendo in modo da ottenere il miglior risultato calcolando come agire in modo efficace e sicuro in una varietà di situazioni nuove. Ha alcuni vantaggi:
 1. Estendibilità e generalità
 2. Misurabilità dei risultati rispetto all'obiettivo

I limiti dipendono dai rischi, dall'etica e dalla complessità computazionale.

1.2 Storia dell'IA

Nasce sin dall'antichità con il desiderio dei filosofi di sollevare l'uomo dalle fatiche del lavoro. Dal 1940 c'è un'esplosione di popolarità che però si alterna tra periodi di crisi e di grandi avanzamenti.



¹Ci sono due umani e una macchina. Tutti questi conversano tramite un computer. Se l'esaminatore non riesce a distinguere l'essere umano dalla macchina allora vince quest'ultima.

Esempio 1.2.1 (Scacchi). Un esempio propedeutico è quello dell'applicazione dell'IA al gioco degli scacchi, definita **IA debole**. Negli anni '60 c'erano principalmente due opinioni al riguardo:

- Newell e Simon sostenevano che in 10 anni le macchine sarebbero state campioni negli scacchi
- Dreyfus sosteneva che una macchina non sarebbe mai stata in grado di giocare a scacchi

Nel 1997 la macchina Deep Blue sconfigge il campione mondiale di scacchi Kasparov. Viene naturale farsi alcune domande...

- Ha avuto **fortuna**?
- Ha avuto un **vantaggio psicologico**? La macchina eseguiva le mosse immediatamente e Kasparov si sentiva come l'ultimo baluardo umano.
- **Forza bruta**? La macchina calcolava 36 miliardi di posizioni ogni 3 minuti

Oggi l'Intelligenza Artificiale eccelle in tutti i giochi. L'ultimo a "cadere" è stato il Go nel 2016. Allo stesso tempo però il livello delle persone è aumentato giocando contro le macchine.

Definizione 1.2.1 (IA debole). *Al contrario dell'IA forte, non ha lo scopo di possedere abilità cognitive generali, ma piuttosto di essere in grado di risolvere esattamente un singolo problema.*

1.3 Reti neurali

Le reti neurali sono caratterizzate da:

- **Flessibilità**: capacità di acquisizione automatica di conoscenza e di adattamento automatico a contesti diversi e dinamici
- **Robustezza**: capacità di trattare incertezza e rumorosità del mondo reale
- Rappresentazione appresa dai dati in forma **sub-simbolica**
- Possibilità di usare più strati di reti neurali con diversi livelli di astrazione (**Deep Learning**)

1.3.1 Deep Learning

Abbinando alla capacità dei modelli di machine learning una grande quantità di dati e degli High Performance Computer, si è favorito molto il deep learning.

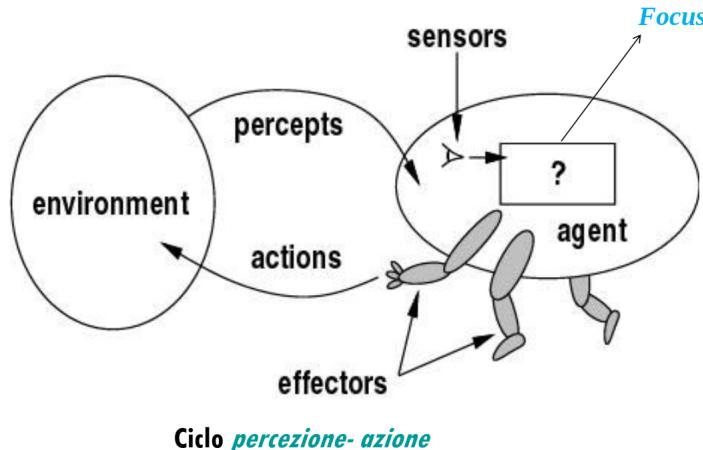
Dal 2010 le reti neurali profonde hanno iniziato a diffondersi molto nelle grandi industrie, riscuotendo successo ad esempio:

- **Computer vision**: ad esempio la classificazione del cancro della pelle
- **Natural Language Processing**: ad esempio IBM Watson o Google DeepL

Questa tecnologia ha raggiunto prestazioni a livello di quelle umane.

2 Agenti intelligenti

L'approccio moderno dell'IA (AIMA) è quello di costruire degli **agenti intelligenti**. La visione ad agenti offre un quadro di riferimento e una prospettiva più generale. È utile anche perché è **uniforme**.



Ciclo **percezione- azione**

Noi ci concentreremo sul programma che sta al centro dell'agente e che consiste in un ciclo di percezione-azione.

2.1 Caratteristiche

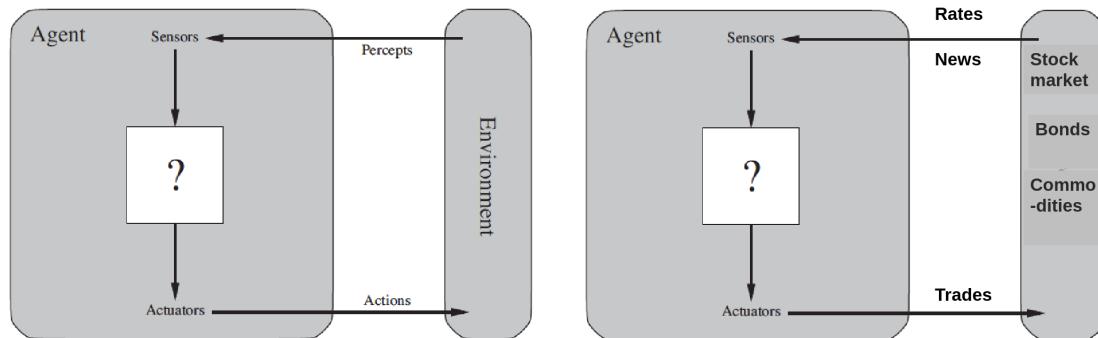
Un agente ha alcune caratteristiche:

- **Situati:** ricevono *percezioni* da un ambiente e agiscono mediante **azioni** (attuatori)

2.1.1 Percezioni e azioni

Le percezioni corrispondono agli **input** dai sensori. La **sequenza percettiva** sarà la storia completa delle percezioni.

La scelta dell'azione è *funzione* unicamente della sequenza percettiva ed è chiamata **funzione agente**. Il compito dell'IA è costruire il programma agente.



2.2 Agente razionale

Un agente razionale interagisce con il suo ambiente in maniera **efficace** (fa la cosa giusta). Si rende quindi necessario un **criterio di valutazione** oggettivo dell'effetto delle azioni dell'agente. La valutazione della prestazione deve avere le seguenti caratteristiche:

- Esterna (come vogliamo che il mondo evolva?)
- Scelta dal progettista a seconda del problema considerando l'effetto desiderato sull'ambiente.
- (possibile) Valutazione su ambienti diversi.

Definizione 2.2.1 (Agente razionale). *Per ogni sequenza di percezioni compie l'azione che massimizza il valore atteso della misura delle prestazioni, considerando le sue percezioni passate e la sua conoscenza pregressa.*

Osservazione 2.2.1. Si basa sulla razionalità e non sull'onniscienza e onnipotenza: non conosce alla perfezione il futuro ma può apprendere e ha dei limiti nelle sue azioni.

Raramente tutta la conoscenza sull'ambiente può essere fornita a priori dal programmatore. L'agente razionale deve essere in grado di modificare il proprio comportamento con l'esperienza. Può **migliorare** esplorando, apprendendo, aumentando l'autonomia per operare in ambienti differenti o mutevoli.

Definizione 2.2.2 (Agente autonomo). *Un agente è autonomo nella misura in cui il suo comportamento dipende dalla sua capacità di ottenere esperienza e non dall'aiuto del progettista.*

2.3 Ambienti

Definire un problema per un agente significa innanzitutto caratterizzare l'ambiente in cui opera. Viene utilizzata la descrizione **PEAS**:

- Performance
- Environment
- Actuators
- Sensors

Prestazione	Ambiente	Attuatori	Sensori
Arrivare alla destinazione, sicuro, veloce, ligio alla legge, viaggio confortevole, minimo consumo di benzina, profitti massimi	Strada, altri veicoli, pedoni, clienti	Sterzo, acceleratore, freni, frecce, clacson, schermo di interfaccia o sintesi vocale	Telecamere, sensori a infrarossi e sonar, tachimetro, GPS, contachilometri, accelerometro, sensori sullo stato del motore, tastiera o microfono

L'ambiente deve avere le seguenti proprietà:

- Osservabilità:
 - Se è **completamente osservabile** l'apparato percettivo è in grado di dare conoscenza completa dell'ambiente o almeno tutto ciò che è necessario per prendere l'azione
 - Se è **parzialmente osservabile** sono presenti limiti o inaccuratezze dell'apparato sensoriale
- Agente singolo o multi-agente:
 - L'ambiente ad agente **singolo** può anche cambiare per eventi, non necessariamente per azioni di agenti
 - Quello **multi-agente** può essere *competitivo* (scacchi) o *cooperativo*
- Predicibilità:
 - **Deterministico**: quando lo stato successivo è completamente determinato dallo stato corrente e dall'azione (e.g. scacchi)
 - **Stocastico**: quando esistono elementi di incertezza con associata probabilità (e.g. guida)
 - **Non deterministico**: quando si tiene traccia di più stati possibili risultato dell'azione ma non in base ad una probabilità

- Episodico o sequenziale:
 - **Episodico:** quando l'esperienza dell'agente è divisa in episodi atomici indipendenti in cui non c'è bisogno di pianificare (e.g. partite diverse)
 - **Sequenziale:** quando ogni decisione influenza le successive (e.g. mosse di scacchi)
- Statico o dinamico:
 - **Statico:** il mondo non cambia mentre l'agente decide l'azione (e.g cruciverba)
 - **Dinamico:** cambia nel tempo, va osservata la contingenza e tardare equivale a non agire (e.g. taxi)
 - **Semi-dinamico:** l'ambiente non cambia ma la valutazione dell'agente sì (e.g. scacchi con timer)
- Valori come lo stato, il tempo, le percezioni e le azioni possono assumere valori **discreti** o **continui**. Il problema è combinatoriale nel discreto o infinito nel continuo.
- **Noto o ignoto:** una distinzione riferita alla conoscenza dell'agente sulle leggi fisiche dell'ambiente (le regole del gioco). È diverso da osservabile.

Definizione 2.3.1 (Simulatore). *Un simulatore è uno strumento software che si occupa di:*

- Generare stimoli
- Raccogliere le azioni in risposta
- Aggiornare lo stato
- Attivare altri processi che influenzano l'ambiente
- Valutare la prestazione degli agenti (media su più istanze)

Gli esperimenti su classi di ambienti con condizioni variabili sono essenziali per **generalizzare**.

2.4 Programma agente

L'agente sarà quindi composto da un'architettura e da un programma. Il programma dell'agente implementa la funzione agente $Ag : Percezioni \rightarrow Azioni$.

```
function Skeleton-Agent (percept) returns action
  static: memory, agent memory of the world
  memory <- UpdateMemory(memory, percept)
  action <- Choose-Best-Action(memory)
  memory <- UpdateMemory(memory, action)
  return action
```

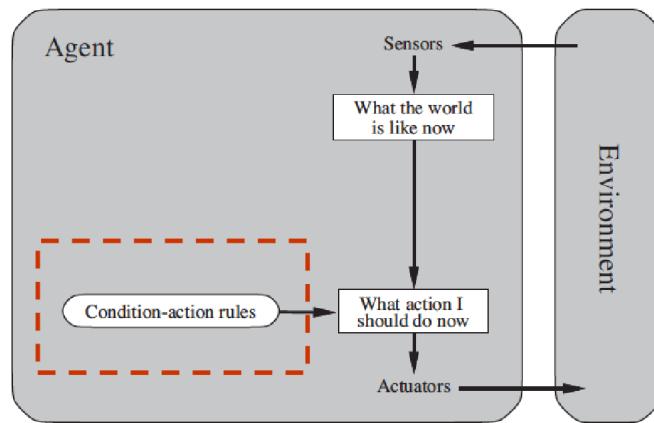
2.4.1 Tabella

Un agente basato su tabella esegue una scelta come un accesso ad una tabella che associa un'azione ad ogni possibile sequenza di percezioni.

Ha una **dimensione ingestibile**, è difficile da costruire, non è autonomo ed è di difficile aggiornamento (apprendimento complesso).

2.4.2 Agenti reattivi

L'agente agisce in base a quello che percepisce senza salvare nulla in memoria.



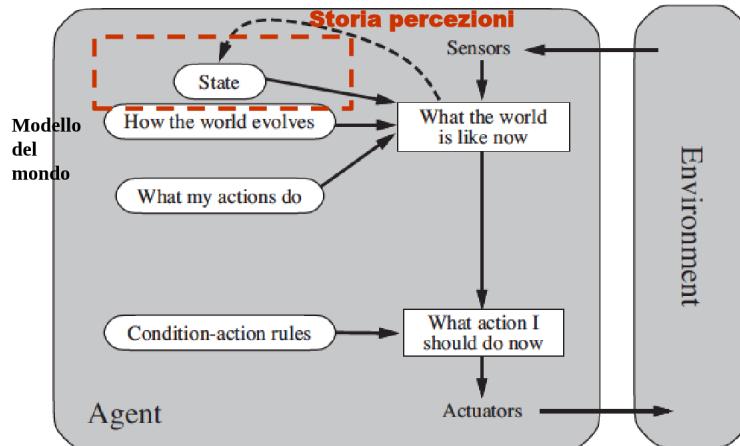
```

function Agente-Reattivo-Semplice (percezione)
    returns azione
    persistent: regole, un insieme di regole
    condizione-azione (if-then)
    stato <- Interpreta-Input(percezione)
    regola <- Regola-Corrispondente(stato, regole)
    azione <- regola.Azione
    return azione

```

2.4.3 Agenti basati su modello

L'agente ha uno stato che mantiene la storia delle percezioni e influenza il modello del mondo.



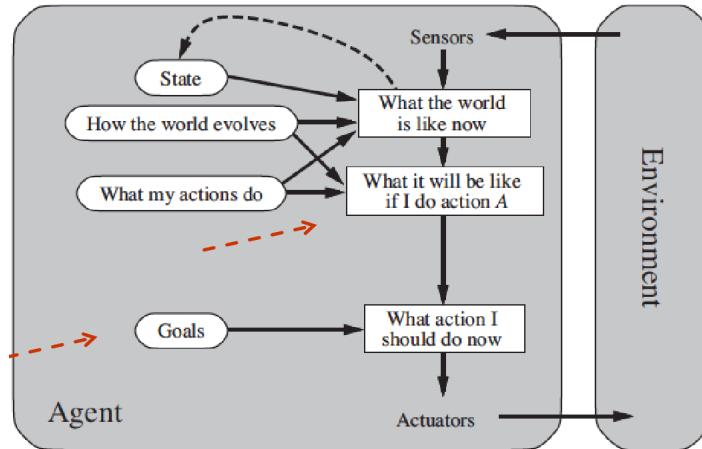
```

function Agente-Basato-su-Modello (percezione)
    returns azione
    persistent: stato, una descrizione dello stato corrente
    modello, conoscenza del mondo
    regole, un insieme di regole condizione-azione
    azione, azione più recente
    stato <- Aggiorna-Stato(stato, azione, percep., modello)
    regola <- Regola-Corrispondente(stato, regole)
    azione <- regola.Azione
    return azione

```

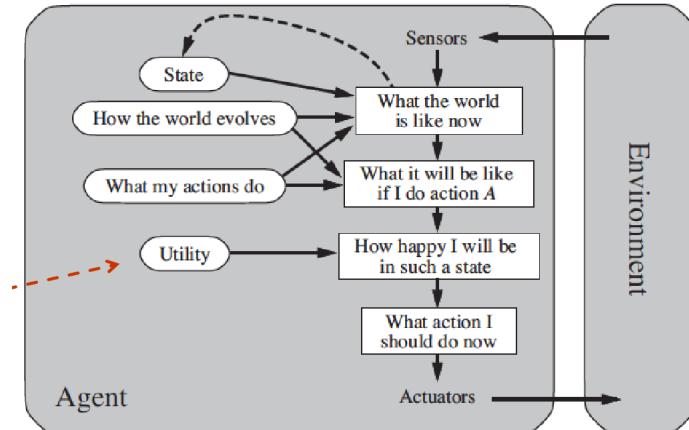
2.4.4 Agenti con obiettivo

Fin'ora l'agente aveva un obiettivo predeterminato dal programma. In questo caso invece viene specificato anche il **goal** che influenza le azioni. Abbiamo quindi più **flessibilità** ma meno efficienza.



2.4.5 Agenti con valutazione di utilità

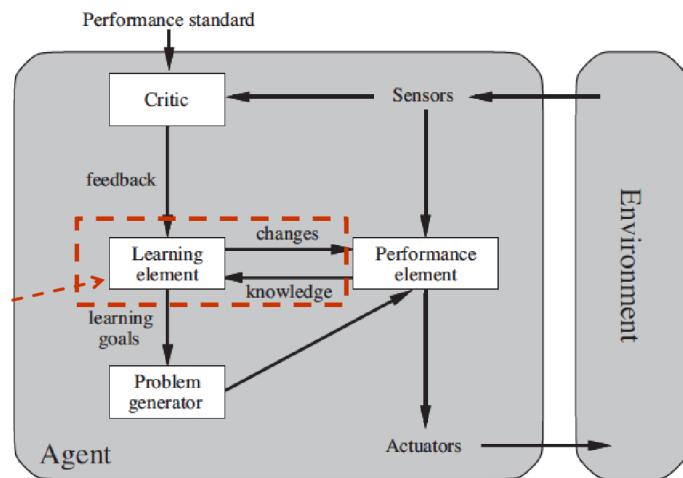
In questo caso ci sono **obiettivi alternativi** o più modi per raggiungerlo. L'agente deve quindi decidere verso dove muoversi e si rende necessaria una **funzione utilità** che associa ad un obiettivo un numero reale. La funzione terrà anche conto della probabilità di successo (**utilità attesa**).



2.4.6 Agenti che apprendono

Questo tipo di agente include la capacità di **apprendimento** che produce cambiamenti al programma e ne migliora le prestazioni, adattando i comportamenti.

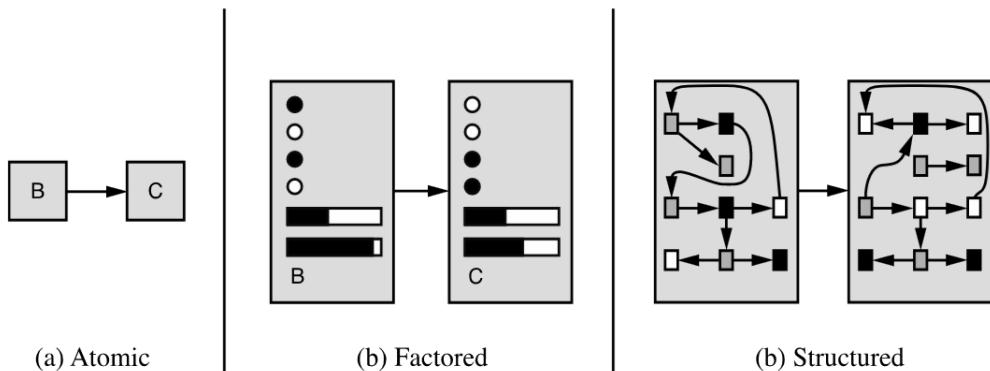
L'elemento **esecutivo** è il programma stesso, quello **critico** osserva e dà feedback ed infine c'è un generatore di problemi per suggerire nuove situazioni da esplorare.



2.4.7 Tipi di rappresentazione

Gli stati e le transizioni possono essere rappresentati in tre modi:

- **Atomica:** solo con gli stati
- **Fattorizzata:** con più variabili e attributi
- **Strutturata:** con l'aggiunta delle relazioni



3 Agenti risolutori di problemi

Gli agenti risolutori di problemi adottano il paradigma della risoluzione di problemi come **ricerca** in uno **spazio di stati**. Sono agenti con **modello** (storia percezioni e stati) che adottano una rappresentazione **atomica** degli stati.

Sono particolari gli agenti con **obiettivo** che pianificano l'intera sequenza di mosse prima di agire.

3.1 Processo di risoluzione

I passi da seguire sono i seguenti:

1. **Determinazione di un obiettivo**, ovvero un insieme di stati in cui l'obiettivo è soddisfatto
2. **Formulazione** del problema tramite la rappresentazione degli stati e delle azioni
3. Determinazione della **soluzione** mediante la ricerca
4. **Esecuzione** del piano

Esempio 3.1.1 (Viaggio con mappa). Supponiamo di voler fare un viaggio. Il processo di risoluzione sarebbe il seguente:

1. Raggiungere Bucarest
2.
 - Azioni: guidare da una città all'altra
 - Stato: città su mappa

Assumiamo che l'ambiente in questione sia **statico**, **osservabile**, **discreto** e **deterministico** (assumiamo un mondo ideale).

3.2 Formulazione del problema

Un problema può essere definito formalmente mediante 5 componenti:

1. **Stato iniziale**
2. **Azioni** possibili
3. **Modello di transizione**:

$$\text{risultato} : \text{stato} \times \text{azione} \rightarrow \text{stato} \quad \text{Risultato}(s, a) = s' \text{ (uno stato successore)}$$

4. **Test obiettivo** per capire tramite un insieme di stati obiettivo se il goal è raggiunto

$$\text{test} : \text{stato} \rightarrow \{\text{true}, \text{false}\}$$

5. **Costo del cammino**: composto dalla somma dei costi delle azioni, dove un passo ha costo $c(s, a, s')$. Un passo non ha mai costo negativo.

I punti 1, 2 e 3 definiscono implicitamente lo **spazio degli stati**. Definirlo esplicitamente può essere molto costoso, come in quasi tutti i problemi di IA, questo sarà rilevante nel seguito, come vedremo nelle prossime lezioni

3.3 Algoritmo di ricerca

Gli algoritmi di ricerca prendono in input un problema e restituiscono un **cammino soluzione**. Dobbiamo misurare le **prestazioni**: trova una soluzione? Quanto costa trovarla? Quanto è efficiente?

$$\text{costo_totale} = \text{costo_ricerca} + \text{costo_cammino_sol}$$

Esempio 3.3.1 (Arrivare a Bucarest). Partiamo con la formulazione del problema:

1. **Stato iniziale:** la città di partenza, ovvero Arad

2. **Azioni:** spostarsi in una città collegata vicina

```
Azioni(In(Arad))={Go(Sibiu), Go(Zerind), ...}
```

3. **Modello di transizione:**

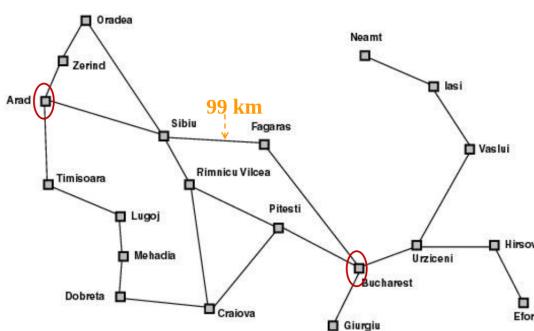
```
Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)
```

4. **Test obiettivo:**

```
{In(Bucarest)}
```

5. **Costo del cammino:** somma delle lunghezze delle strade

In questo esempio lo spazio degli stati coincide con la rete dei collegamenti tra le città.

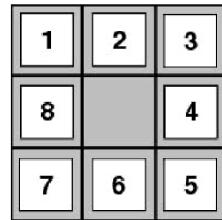


Esempio 3.3.2 (Puzzle dell'8). Partiamo con la formulazione del problema:

1. **Stati:** tutte le possibili configurazioni della scacchiera

2. **Stato iniziale:** una configurazione tra quelle possibili

3. **Obiettivo:** una configurazione del tipo



4. **Azioni:** le mosse della casella vuota

5. **Costo cammino:** ogni passo costa 1

In questo esempio lo spazio degli stati è un grafo con possibili cicli (ci possiamo ritrovare in configurazioni già viste). Il problema è NP-completo: per 8 tasselli ci sono $\frac{9!}{2} = 181.000$ stati.

Esempio 3.3.3 (8 regine). Supponiamo di dover collocare 8 regine su una scacchiera in modo tale che nessuna regina sia attaccata da altre.

1. **Stati:** tutte le possibili configurazioni della scacchiera con 0-8 regine

2. **Goal test:** avere 8 regine sulla scacchiera, di cui nessuna è attaccata

3. Azioni: aggiungi una regina

In questo esempio lo spazio degli stati sono le possibili scacchiere, ovvero $64 \times 63 \times \dots \times 57 \simeq 1.8 \times 10^{14}$. Proviamo ad utilizzare una formulazione diversa:

1. **Stati:** tutte le possibili configurazioni della scacchiera in cui *nessuna regina è minacciata*
2. **Goal test:** avere 8 regine sulla scacchiera, di cui nessuna è attaccata
3. **Azioni:** aggiungere una regina nella colonna vuota più a destra ancora libera in modo che non sia minacciata

Lo spazio degli stati passa a 2057, anche se comunque rimane esponenziale per k regine. Vediamo infine un'ultima formulazione:

1. **Stati:** scacchiere con 8 regine, una per colonna
2. **Goal test:** nessuna delle regine già presenti è attaccata
3. **Azioni:** sposta una regina nella colonna se minacciata
4. **Costo cammino:** zero

Qui lo spazio degli stati è di qualche decina di milione.

Capiamo quindi che formulazioni diverse del problema portano a spazi di stati di dimensioni diverse.

Esempio 3.3.4 (Dimostrazione di teoremi). Dato un insieme di premesse:

$$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v\} \quad (1)$$

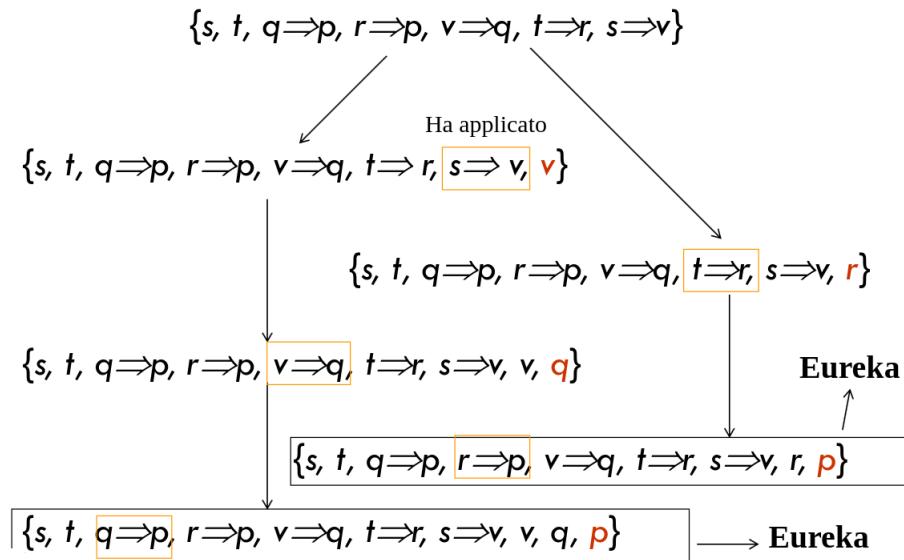
dimostrare una proposizione p utilizzando solamente la regola di inferenza *Modus Ponens*:

$$(p \wedge p \Rightarrow q) \Rightarrow q$$

Scriviamo la formulazione del problema:

- **Stati:** insieme di proposizioni
- **Stato iniziale:** le premesse
- **Stato obiettivo:** un insieme di proposizioni contenente il teorema da dimostrare
- **Operatori:** l'applicazione del Modus Ponens

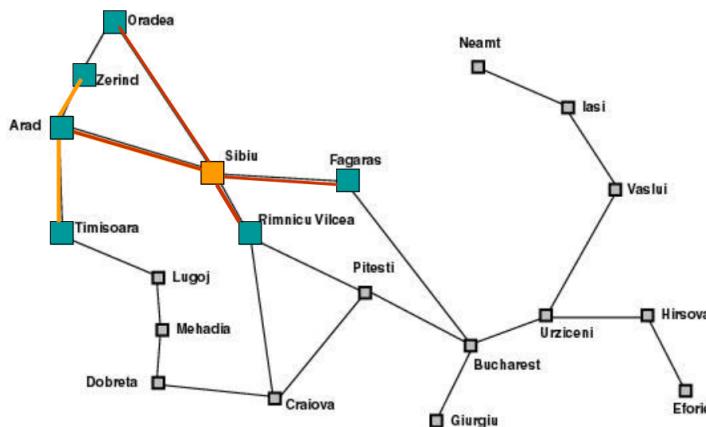
Lo spazio degli stati è quindi il seguente:



3.4 Ricerca della soluzione

La ricerca della soluzione consiste nella generazione di un **albero di ricerca** a partire dalle possibili sequenze di azioni che si sovrappone allo spazio degli stati.

Ad esempio per il caso di Bucarest:



Espandiamo ogni nodo con i suoi possibili successori (frontiera).

Osservazione 3.4.1. Si noti che un nodo dell'albero è diverso da uno stato. Infatti possono esistere nodi dell'albero di ricerca con lo stesso stato (si può tornare indietro).

La generazione di un albero di ricerca sovrapposto allo spazio degli stati:

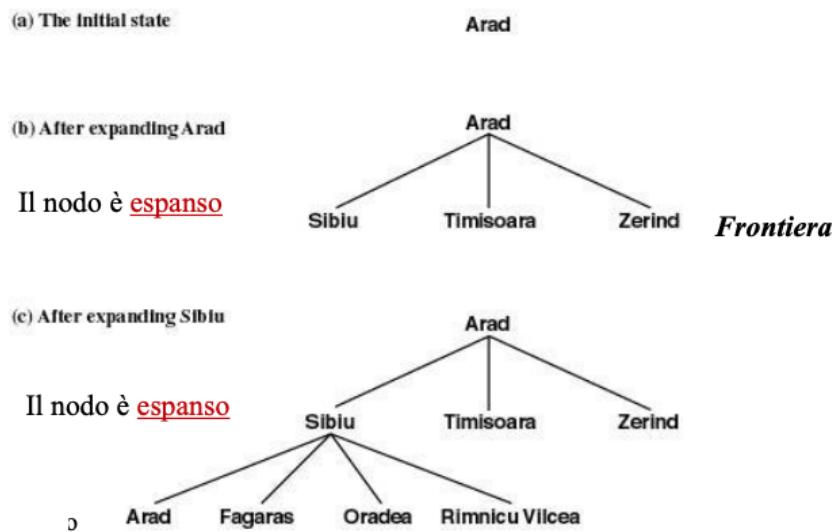


Figure 1: generazione di un albero di ricerca

```

function Ricerca-Albero (problema)
    returns soluzione oppure fallimento
    Inizializza la frontiera con stato iniziale del problema
    loop do
        if la frontiera è vuota then return fallimento
        Scegli* un nodo foglia da espandere e rimuovilo dalla frontiera
        if il nodo contiene uno stato obiettivo
            then return la soluzione corrispondente
        Espandi il nodo e aggiungi i successori alla frontiera

```

Un nodo n in un albero di ricerca è una struttura dati con quattro componenti:

1. Uno stato: n.stato
2. Il nodo padre: n.padre
3. L'azione effettuata per generarlo: n.azione
4. Il costo del cammino dal nodo iniziale al nodo: n.costo-cammino indicato come $g(n)$.
(=padre.costo-cammino + costo-passo ultimo)

Abbiamo poi la struttura dati per la **frontiera** che è una lista dei nodi in attesa di essere espandi (le foglie dell'albero di ricerca). La frontiera è implementata come una coda con operazioni:

- Vuoto?(coda) = controlla se la coda è vuota
- POP(coda) = estraie il primo elemento.
- Inserisci(elemento, coda) = inserisce un elemento nella coda.
- Diversi tipi di coda hanno diverse funzioni di inserimento e implementano strategie diverse.
 - **FIFO** - First in First out \Rightarrow usato nella **BF (Breadth-first)**.
Viene estratto l'elemento più vecchio (in attesa da più tempo); i nuovi nodi sono aggiunti alla fine.
 - **LIFO** - Last in First Out \Rightarrow usato nella **DF (depth-first)**.
Viene estratto il più recentemente inserito; i nuovi nodi sono inseriti all'inizio (pila).
 - **Coda con priorità** \Rightarrow usato nella **UC**, ed altri successivi.
Viene estratto quello con priorità più alta in base a una funzione di ordinamento; dopo l'inserimento dei nuovi nodi si riordina.

4 Stategia ricerca non informative

Ora andremo a vedere diverse **strategie non informative**: Ricerca in ampiezza (BF), Ricerca in profondità (DF) Ricerca in profondità limitata (DL), Ricerca con approfondimento iterativo (ID), Ricerca di costo uniforme (UC). Successivamente le metteremo a confronto con strategie di **ricerca euristica (o informativa)** che fanno uso di informazio riguardo alla distanza stimata della soluzione.

La valutazione di una strategia verrà fatta andando a seguire i seguenti parametri:

- **Completezza**: se la soluzione esiste viene trovata.
- **Ottimalità (ammissibilità)**: trovare la soluzione migliore con costo minore (per il costo del cammino soluzione).
- **Complessità in tempo**: tempo richiesto per trovare la soluzione (per il costo della ricerca)
- **Complessità in spazio**: memoria richiesta (per il costo della ricerca).

4.1 Ricerca in ampiezza (BF)

O come esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità. Viene inoltre implementata con una coda che inserisce alla fine (FIFO).

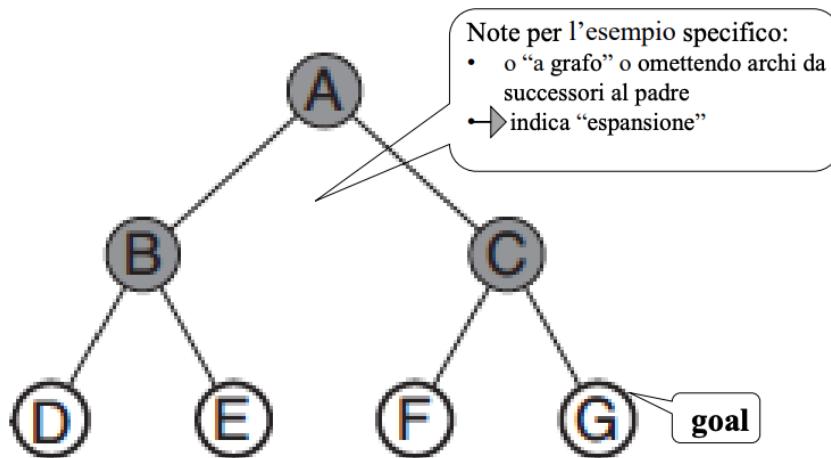


Figure 2: Ricerca in ampiezza

```

function Ricerca-Ampiezza (problema)
    return soluzione oppure fallimento
    nodo = un nodo con stato il problema.stati-iniziale e costo-di-cammino=0
    if problema.Test-Obiettivo(nodo.Stato) then return Soluzion(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    loop do
        if(Vuota?(frontiera)) then return fallimento
        nodo = POP(frontiera)
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione) [costruttore: vedi AIMA]
            if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
            frontiera = Inserisci(figlio, frontiera) /* frontiera gestita come coda FIFO

```

Note 4.1.1. Nota che in questa versione i nodo.stato sono goal-tested al momento in cui sono generati, anticipato→ più efficiente, si ferma appena trova goal prima di espandere.

Una versione aggiornata dove evitiamo di espandere (nodi con) stati già esplorati:

```

function Ricerca-Ampiezza (problema)
    return soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    if problema.Test-Obiettivo(nodo.Stato) then return Soluzion(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto //aggiunto per gestire gli stati ripetuti
    loop do
        if(Vuota(frontiera)) then return fallimento
        nodo = POP(frontiera) // aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione) [costruttore: vedi AIMA]
            if figlio.Stato non e in esplorati e non e in frontiera then // aggiunto check per
                vedere se e in frontiera
            frontiera = Inserisci(figlio, frontiera) /* frontiera gestita come coda FIFO

```

Abbiamo aggiunto **esplorati = insieme vuoto** e **if figlio.Stato non è in esplorati e non è in frontiera then** per gestire gli stati ripetuti.

Ora sempre lo stesso codice in uno script python:

```

def breadth_first_search(problem): # """Ricerca-grafo in ampiezza"""
    explored = [] # insieme degli stati già visitati (implementato come una lista)
    node = Node(problem.initial_state) # il costo del cammino è inizializzato nel costruttore del nodo
    if problem.goal_test(node.state):
        return node.solution(explored_set = explored)
    frontier = FIFOQueue() # la frontiera è una coda FIFO
    frontier.insert(node)
    while not frontier.isempty(): # seleziona il nodo per l'espansione
        node = frontier.pop()
        explored.append(node.state) # inserisce il nodo nell'insieme dei nodi esplorati
        for action in problem.actions(node.state):
            child_node = node.child_node(problem,action)
            if (child_node.state not in explored) and (not frontier.contains_state(child_node.state)):
                if problem.goal_test(child_node.state):
                    return child_node.solution(explored_set = explored)
                # se lo stato non è uno stato obiettivo allora inserisci il nodo nella frontiera
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento

```

4.1.1 Analisi complessità spazio-temporale (BF)

Inanzitutto assumiamo che:

- **b** = fatto di ramificazione (branching)
- **d** = profondità del nodo obiettivo più superficiale (depth)
- **m** = lunghezza massima dei cammini nello spazio degli stati (max)

La **strategia ottimale** è se gli operatori hanno tutti lo stesso costo k cioè $g(n) = k \cdot \text{depth}(n)$ dove $g(n)$ è il costo del cammino per arrivare a n.

La **complessità nel tempo** (nodi generati):

$$T(b, d) = 1 + b + b^2 + \dots + b^d \rightarrow O(b^d) \text{ b figli per ogni nodo}$$

Note 4.1.2. Riflettere che il numero nodi cresce exp., non assumiamo di conoscere già il grafo né una notazione di linearità nel numero nodi. Questo è tipico dei problemi in AI (pensate a quelli generati per le configurazioni dei giochi, con rappresentazione implicita dello spazio stati, non esplicitamente/staticamente in spazi enormi).

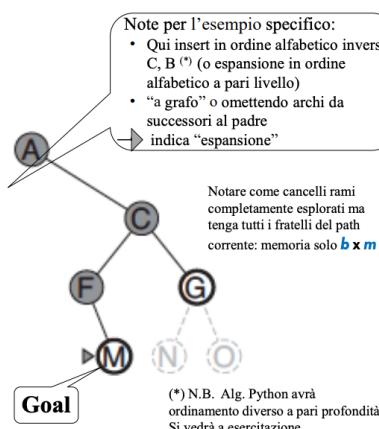
La **complessità nello spazio** (nodi in memoria): $O(b^d)$

Esempio 4.1.1. $b=10$, 1 milione nodi al sec generati; 1 nodo occupa 1000 byte

Profondità	Nodi	Tempo	Memoria	Piu incisivo!
2	110	0,11 ms	107 kilobyte	
4	11.100	11 ms	10,6 megabyte	
6	10^6	1.1 sec	1 gigabyte	
8	10^8	2 min	103 gigabyte	
10	10^{10}	3 ore	10 terabyte	
12	10^{12}	13 giorni	1 petabyte	
14	10^{14}	3,5 anni	1 esabyte	

Scala male: solo istanze piccole!

4.2 Ricerca in profondità (DF)



Viene implementata da una coda che mette i successori in testa alla lista (LIFO, pila o stack). L'algoritmo è generale e può essere usato sia con alberi che con grafì. Notare come cancelli rami completamente esplorati ma tenga tutti i fratelli del path corrente: memoria solo $b \times m$

Analisi - Versione su albero.

Se $m \rightarrow$ lunghezza massima dei cammini nello spazio degli stati e $b \rightarrow$ fattore di diramazione.

Abbiamo che tempo: $O(b^m)$ [che può essere $> O(b^d)$].

Occupazione memoria: bm [frontiera sul cammino].

Analisi - Versione su grafo

In caso di DF con visita grafo si perderebbe i **vantaggi di memoria**: la memoria torna da bm a tutti i possibili stati (potenzialmente, caso pessimo, esponenziale come BF*) (per mantenere la lista dei visitati/esplorati), ma così DF diviene **completa** in spazi degli stati finiti (tutti i nodi verranno espansi nel caso pessimo).

In ogni caso resta non completa in spazi infiniti. È possibile controllare anche solo i nuovi nodi rispetto al cammino radice-nodo corrente senza aggravio di memoria (evitando però così i cicli in spazi finiti ma non i cammini ridondanti).

4.2.1 DF ricorsiva

Ancora più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (solo m nodi nel caso pessimo). Realizzata da un algoritmo ricorsivo “con backtracking” che non necessita di tenere in memoria b nodi per ogni livello, ma salva lo stato su uno stack a cui torna in caso di fallimento per fare altri tentativi (generando i nodi fratelli al momento del backtracking).

```

function Ricerca-DF-A (problema)
    returns soluzione oppure fallimento
    return Ricerca-DF-ricorsiva(CreaNodo(problema.Stato-iniziale), problema)

function Ricerca-DF-ricorsiva(nodo, problema)
    returns soluzione oppure fallimento
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    else
        for each azione in problema.Azioni(nodo.Stato) do

```

```

figlio = Nodo-Figlio(problema, nodo, azione)
risultato = Ricerca-DF-ricorsiva(figlio, problema)
if risultato != fallimento then return risultato
return fallimento

```

```

def recursive_depth_first_search(problem, node):
    """Ricerca in profondità ricorsiva"""
    # controlla se lo stato del nodo è uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    # in caso contrario continua
    for action in problem.actions(node.state):
        child_node = node.child_node(problem, action)
        result = recursive_depth_first_search(problem, child_node)
        if result is not None:
            return result
    return None #con fallimento

```

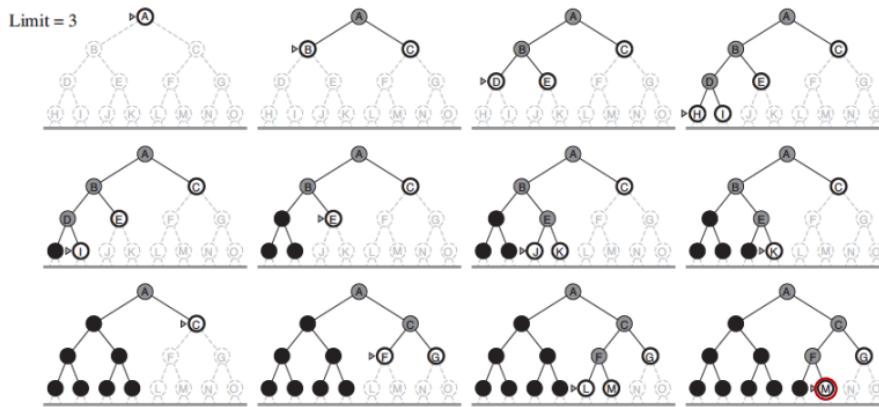
4.2.2 Ricerca in profondità limitata

Si va in profondità fino ad un certo livello predefinito l . È una soluzione definibile **completa** per problemi in cui si conosce un limite superiore per la profondità della soluzione (e.s Route-finding limitata dal numero di città - 1). È però completa se $d < l$ (d profondità nodo obiettivo superficiale). Questa soluzione non è ottimale.

- Complessità tempo: $O(b^l)$
- Complessità spazio: $O(bl)$

4.3 Approfondimento iterativo (ID)

Si prova DF (DL) con limite di profondità 0, poi 1, poi 2, poi 3 ... fino a trovare la soluzione.



Miglior compromesso tra BF e DF.

$$BF : b + b^2 + \dots + b^{d-1} + b^d \text{ con } b = 10 \text{ e } d = 5$$

$$10 + 100 + 1000 + 10.000 + 100.000 = 111.110$$

ID: i nodi dell'ultimo livello generati una volta, quelli del penultimo 2, quelli del terzultimo 3 ... quelli del primo d volte.

$$\begin{aligned} ID &: (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d \\ &= 50 + 400 + 3000 + 20.000 + 100.000 = 123.450 \end{aligned}$$

Complessità tempo: $O(b^d)$ (se esiste soluzione)

Spazio: $O(bd)$ (se esiste soluzione) versus $O(b^d)$ della BF.

Ergo: Vantaggi della BF (completo, ottimale se costo fisso oper. K), con tempi analoghi ma costo memoria analogo a quello di DF.

4.4 Direzione della ricerca

Un problema ortogonale alla strateiga è la direzione della ricerca:

- Ricerca **in avanti** o **guidata dai dati**: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo.
- Ricerca **all'indietro** o **guidata dall'obiettivo**: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

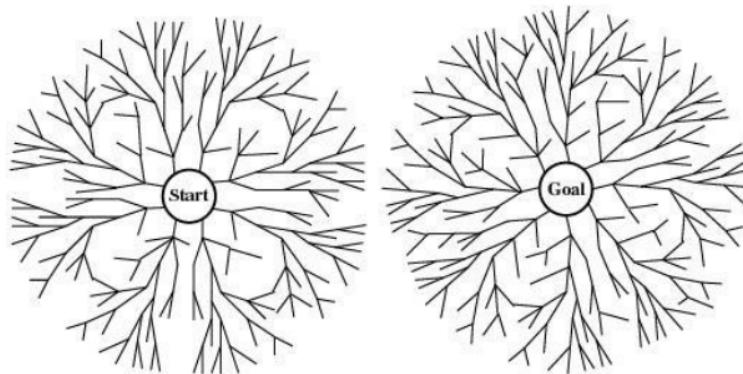
A questo punto bisogna capire quale direzione scegliere?

Conviene procedere nella direzione in cui il fattore di diramazione è minore.

- Si preferisce ricerca all'indietro quando, e.g l'obiettivo è chiaramente definito (e.g theorem proving) o si possono formulare una serie limitata di ipotesi.
- Si preferisce ricerca in avanti quando e.g gli obiettivi possibili sono molti (design).

4.4.1 Ricerca bidirezionale

Si procede nelle due direzioni fino ad incontrarsi.



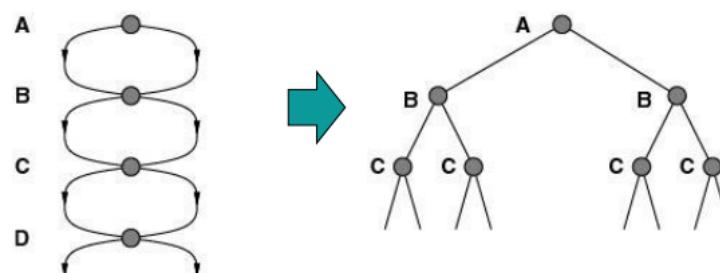
Complessità tempo: $O(b^{d/2})$ (assumendo test intersezione in tempo costante, es. hash table).

Complessità spazio: $O(b^{d/2})$ (almeno tutti i nodi una direzione in memoria, es usando BF).

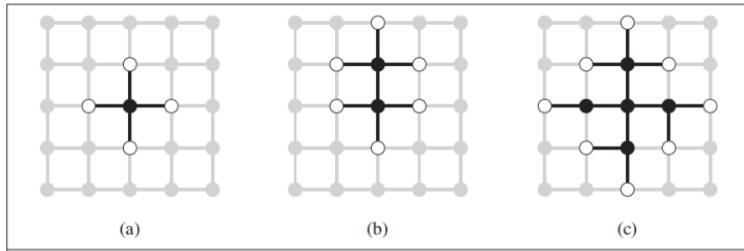
Note 4.4.1. Non sempre applicabile, es. predecessori non definiti, troppi stati obiettivo, ...

4.5 Ridondanze

Su spazi di stati a grafo si possono generare più volte gli stesso nodi (o meglio nodi con stesso stato) nella ricerca, **anche in assenza di cicli** (cammini ridondanti).



Se vediamo per esempio il caso di ridondanza nelle griglie spesso si vanno a visitare stati già visitati questa operazione fa compiere lavoro inutile. Come evitarlo?



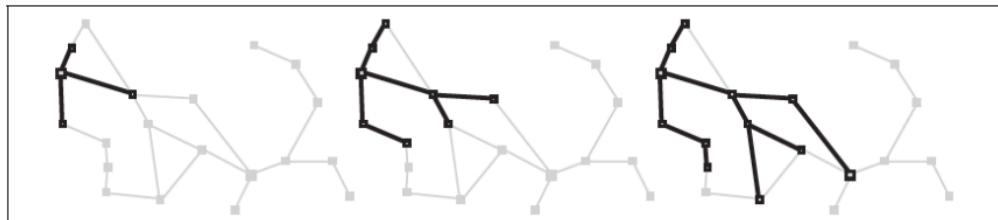
Ricordare gli stati già visitati occupa spazio (es. lista **esplorati** in visita a grafo) ma ci consente di evitare di visitarli di nuovo. Gli algoritmi che dimenticano la propria storia sono destinati a ripeterlo! Abbiamo tre soluzioni, in ordine crescente di costo e di efficacia:

- Non tornare nello stato da cui si proviene: si elimina il genitore dai nodi successori (non evita i cammini ridondanti).
- Non creare cammini con cicli: si controlla che i successori non siano antenati del nodo corrente (detto per la DF).
- Non generare nodi con stati già visitati/esplorati: ogni nodo visitato deve essere tenuto in memoria per una complessità $O(s)$ dove s è il numero di stati possibili (e.g. hash table per accesso efficiente).

Ricordare che il costo può essere alto: in caso di DF (profon.) la memoria torna da bm a tutti gli stati, ma diviene una ricerca completa (per spazi finiti). Ma in molti casi gli stati crescono exp. (gioco otto, scacchi, ...)

Note 4.5.1. Ricorda che in una ricerca su un grafo:

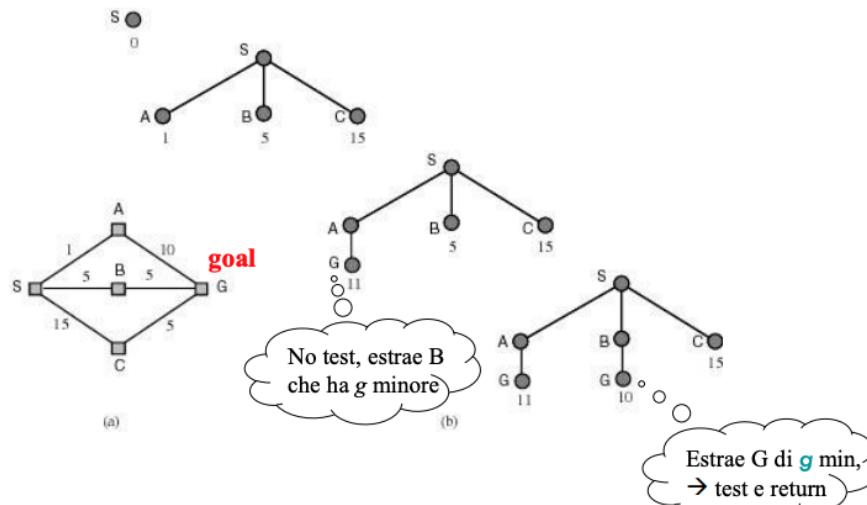
- Si mantiene una lista dei nodi (stati) visitati/esplorati (anche detta **lista chiusa**) ²
- Prima di espandere un nodo si controlla se lo stato era stato già incontrato prima o è già nella frontiera.
- Se questo succede, il nodo appena trovato non viene espanso.
- Ottimale solo se abbiamo la garanzia che il costo del nuovo cammino sia maggiore o uguale (cioè che il nuovo cammino non conviene)



La ricerca su grafo esplora uno stato al più una volta. Una proprietà è che: la **frontiera** separa i nodi esplorati da quelli non-esplorati (ogni cammino dallo stato iniziale a inesplorati deve attraversare uno stato della frontiera).

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): si sceglie il nodo di costo minore sulla frontiera (si intende il costo $g(n)$ del cammino), si espande sui contorni di **uguale (o meglio uniforme) costo (e.g. in km)** invece che sui contorni di uguale profondità (BF).

²Ed. IV AIMA: introduce il termine di insieme di stati “raggiunti” che include sia (gli stati del)la frontiera che la lista degli esplorati



4.6 Ricerca di costo uniforme (UC)

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): si sceglie il nodo di costo minore sulla frontiera (si intende il costo $g(n)$ del cammino), si espande sui contorni di **uguale (o meglio uniforme) costo (e.g. in km)** invece che sui contorni di uguale profondità (BF). Implementata da una coda ordinata per costo cammino crescente (in cima i nodi di costo minore). Codice di ricerca UC su albero:

```
function Ricerca-UC-A (problema)
    returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    frontiera = una coda con priorita con nodo come unico elemento
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera)
        // Posticipato* per vedere il costo minore su g (diverso da BF, ma tipico per coda
        // priorita)
        if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            frontiera = Inserisci(figlio, frontiera) /* in coda con priorita */
    end
```

Codice di ricerca UC su grafo:

```
function Ricerca-UC-G (problema)
    returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    frontiera = una coda con priorita con nodo come unico elemento
    esplorati = insieme vuoto
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera);
        // posticipato per vedere il costo minore
        if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non e in esplorati e non e in frontiera then
                frontiera = Inserisci(figlio, frontiera) /* in coda con priorita */
            else if figlio.Stato e in frontiera con Costo-cammino piu alto then // costo
                cammino piu alto g
                sostituisci quel nodo frontiera con figlio
```

Codice in python della ricerca:

```

def uniform_cost_search(problem): """Ricerca-grafo UC"""
    explored = [] # insieme (implementato come una lista) degli stati già visitati
    node = Node(problem.initial_state) # il costo del cammino è inizializzato nel costruttore del nodo
    frontier = PriorityQueue(f =lambda x:x.path_cost) # la frontiera è una coda coda con priorità
    #lambda serve a definire una funzione anonima a runtime
    frontier.insert(node)
    while not frontier.isempty():
        # seleziona il nodo node = frontier.pop() # estraie il nodo con costo minore, per l'espansione
        if problem.goal_test(node.state):
            return node.solution(explored_set =explored)
        else:
            # se non lo è inserisci lo stato nell'insieme degli esplorati
            explored.append(node.state)
            for action in problem.actions(node.state):
                child_node =node.child_node(problem, action)
                if (child_node.state not in explored) and (not frontier.contains_state(child_node.state)):
                    frontier.insert(child_node)
                elif frontier.contains_state(child_node.state) and
                    (frontier.get_node(frontier.index_state(child_node.state)).path_cost > child_node.path_cost):
                    frontier.remove(frontier.index_state(child_node.state))
                    frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento

```

4.6.1 Analisi

Ottimalità e completezza garantiscono che il costo degli archi sia maggiore di $\epsilon > 0$. Appunto C^* come il costo della soluzione ottima, $\lfloor C^*/\epsilon \rfloor$ è il numero di mosse nel caso peggiore, arrotondato per difetto (e.g. attratto ad andare verso tante basse)

Complessità: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Note 4.6.1. Quando ogni azione ha lo stesso costo UC somiglia a BF ma complessità $O(b^{1+d})$

Causa esame ed arresto posticipato, solo dopo aver espanso frontiera, oltre la profondità del goal.

4.7 Confronto strategie

Criterio	BF	UC	DF	DL	ID	Bidir
Completa?	si	si(^Λ)	no	si (+)	si	si (£)
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^t)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Ottimale?	si(*)	si(^Λ)	no	no	si(*)	si (£)

(*) se gli operatori/archi hanno tutti lo stesso costo.

(Λ) per costi degli archi $\geq \epsilon > 0$.

(+) per problemi per cui si conosce un limite alla profondità della soluzione (se $l > d$).

(£) usando UC (o BF).

5 Ricerca euristica

La ricerca esaustiva non è praticabile in problemi di complessità esponenziale (e.g. 10^{120} configurazioni in scacchi). Noi usiamo conoscenza del problema e esperienza per riconoscere i cammini più promettenti, usiamo una stima del costo futuro, evitando di generare gli altri. La conoscenza euristica (dal greco "eureka") aiuta fare scelte "oculate", questa ovviamente però non evita la ricerca ma la riduce, consente in genere di trovare una buona soluzione in tempi accettabili sotto certe condizioni garantisce completezza e ottimalità.

La conoscenza del problema data tramite una funzione di valutazione f , che include h detta **funzione di valutazione euristica**.

$$h : n \rightarrow R$$

La funzione si applica al nodo ma dipende solo dallo stato (n .Stato).

Note 5.0.1. Manteniamo la notazione in n per uniformità con g ; g dipende anche dal cammino fino al nodo.

$$f(n) = g(n) + h(n) \text{ ove } g(n) \text{ è il costo cammino visto con UC}$$

Per procedere preferibilmente verso il percorso migliore, seguendo problem-specific information, di stima del costo futuro:

- La città più vicina (o la città più vicina alla metà in linea d'aria - tabella esterna) nel problema dell'itinerario.
- Il numero delle caselle fuori posto nel gioco dell'otto.
- Il vataggio in pezzi della dama o negli scacchi

Esempio 5.0.1. Mappa Romania dist. in linea d'aria.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

5.1 Algoritmo di ricerca Best-first

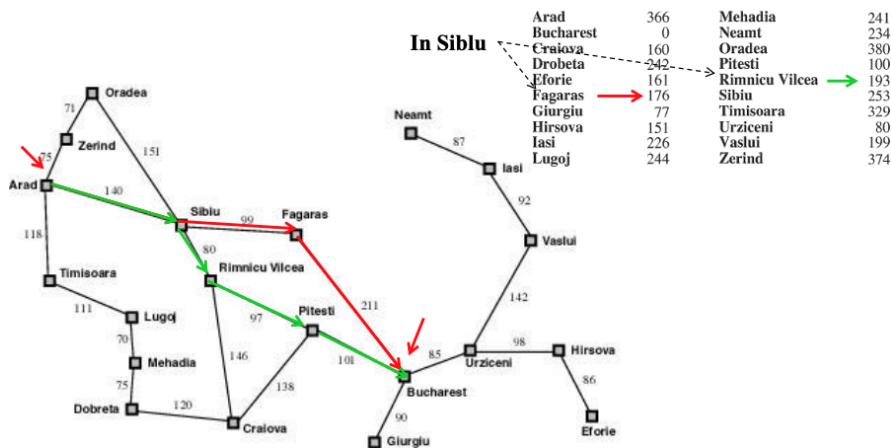
Il **best first - heuristic** usa lo stesso algoritmo di UC³ ma con uso di f (stima di costo) per la coda con priorità. Una volta scelta f determina la strategia di ricerca. A ogni passo si sceglie il nodo sulla frontiera per cui il valore della f è migliore (il nodo più promettente).

Note 5.1.1. Migliore significa "minore" in caso di un'euristica che stima la distanza della soluzione

Un caso speciale: **greedy best-first**, su usa solo $h(f = h)$.

Esempio 5.1.1. Esempio di greedy best-first con $f = h$ Da Arad a Bucharest con **Greedy best-first**: Arad, sibiu, fagaras, bucharest (450) ma non è l'ottimale che sarebbe: Arad, Sibiu, Rimnicu, Pitesti, Bucharest (418).

³Warning: AIMA ed. IV ha usato uno schema di UC diverso e alcune proprietà cambiano



5.2 Algoritmo A

Si può dire qualcosa di f per avere garanzie di completezza e ottimalità?

Definizione 5.2.1. Un **Algoritmo A** è un algoritmo di best first con una fusione di valutazione dello stato del tipo:

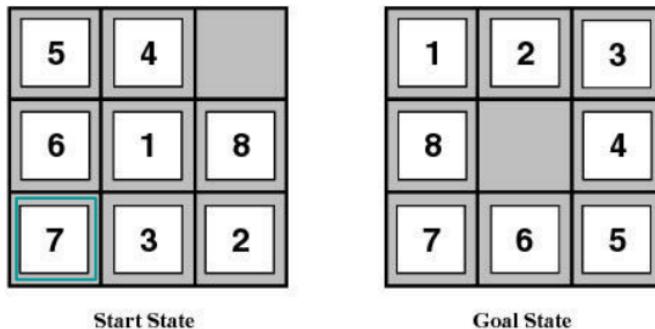
$$f(n) = g(n) + h(n) \text{ con } h(n) \geq 0 \text{ e } h(goal) = 0$$

In questa definizione abbiamo che $g(n)$ è il costo del cammino percorso per raggiungere n , mentre $h(n)$ una stima del costo per raggiungere da n un nodo goal (distanza).

Vedremo alcuni casi particolari dell'algoritmo A:

- Se $h(n) = 0$ [$f(n) = g(n)$] si ha **Ricerca Uniforme (UC)**.
- Se $g(n) = 0$ [$f(n) = h(n)$] si ha **Greedy Best First**.

Esempio 5.2.1. Esempio nel gioco dell'otto.



$$f(n) = \# \text{mosse fatte} + \# \text{caselle-fuori-posto} \quad f(start) = 0 + 7 \quad f(goal-state) = ? + 0$$

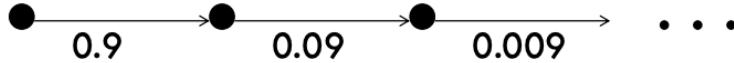
Dopo $\leftarrow, \downarrow, \uparrow, \rightarrow$ abbiamo che $f = 4 + 7$, stesso stato, g è cambiato.

Teorema 5.2.1. L'algoritmo A con la condizione:

$$g(n) \geq d(n) \cdot \epsilon \quad (\epsilon > 0 \text{ costo minimo arco})$$

è completo (con $d(n)$ che è la distanza).

Note 5.2.1. La condizione ci garantisce che non si verifichino situazioni strane del tipo: e quindi che il costo lungo un cammino non cresca "abbastanza" (se cresce abbastanza possiamo fermare quel path per costo alto di g).



Dimotrazione 5.2.1. Sia $[n_0, n_1, n_2, \dots, n', \dots, n_k = goal]$ un cammino soluzione. Sia n' un nodo della frontiera su un cammino soluzione: n' prima o poi sarà espanso. Infatti esistono solo un numero finito di nodi x che possono essere aggiunti alla frontiera con $f(x) \leq f(n')$ (è la condizione sulla crescita di g , scritta precedentemente, tale che non esista una catena infinita di archi e nodi che possa aggiungere con costo sempre $\leq f(n')$).

Quindi, se non si trova una soluzione prima, n' verrà espanso e i suoi successori aggiunti alla frontiera. Tra questi anche il suo successore sul cammino soluzione.

Il ragionamento si può ripetere fino a dimostrare che anche il nodo goal sarà selezionato per l'espansione.

5.3 Algoritmo A^*

La funzione di valutazione ideale (oracolo):

$$f^*(n) = g^*(n) + h^*(n)$$

Con $g^*(n)$ il costo del cammino minimo da radice a n , $h^*(n)$ costo del cammino minimo da n a goal, $f^*(n)$ costo del cammino minimo da radice a goal, attraverso n . Normalmente:

$$g(n) \geq g^*(n) \quad e \quad h(n) \text{ è una stima di } h^*(n)$$

($g(n) \geq g^*(n)$ rappresenta costo cammino \geq costo migliore). Si può andare in sottostima (e.g. linea d'aria) o sovrastima della distanza dalla soluzione.

Definizione 5.3.1 (Euristica ammissibile).

$$\forall n \text{ t.c. } h(n) \leq h^*(n) \quad h \text{ è una sottostima}$$

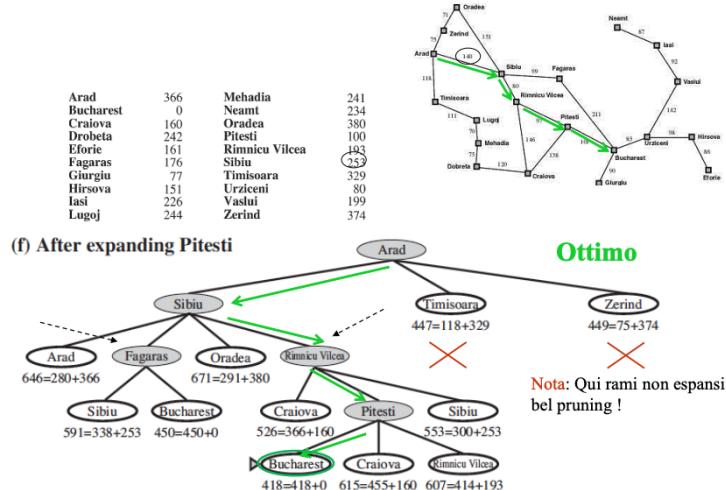
Esempio 5.3.1. L'euristica della distanza in linea d'aria.

Definizione 5.3.2 (Algoritmo A^*). *Un algoritmo A in cui h è una funzione euristica ammissibile.*

Teorema 5.3.1. Gli algoritmi A^* sono ottimali.

Corollario 5.3.1.1. $BF^{(+)}$ e UC sono ottimali ($h(n) = 0$).

Esempio 5.3.2. Itinerario con A^* (ad albero).



Osservazione 5.3.1. Alcune osservazioni su A^*

1. Rispetto a greedy best-first, la componente g fa sì che si abbandonino cammini che vanno troppo in profondità.
2. Ha sotto o sovra stima?
 - (a) Una sottostima (h) può farci compiere del lavoro inutile (tenendo anche candidati non buoni), però non ci fa perdere il cammino migliore (quando prendo nodo goal è il cammino migliore).
 - (b) Una funzione che qualche volta sovrastima può farci perdere la soluzione ottimale (taglio per causa di sovrastima, invece era buona)

5.3.1 Ottimalità su A^*

Nel caso di ricerca a/su albero l'uso di un'euristica ammissibile è sufficiente a garantire l'ottimalità su A^* . Nel caso di ricerca su grafo (con UC come visto) serve una proprietà più forte: la **consistenza** (detta anche **monotonicità**).

Per evitare rischio di scartare candidati ottimi (stato già incontrato) si vuol evitare, causa uso della lista esplorati, di far sparire, o meglio non considerare al momento dell'espansione, candidati ottimali. Cerchiamo quindi condizioni per garantire che il primo espanso sia il migliore.

Definizione 5.3.3. Un euristica **consistente** [$h(goal) = 0$] (consistenza locale).

$$\forall n \text{ t.c. } h(n) \leq c(n, a, n') + h(n') \text{ dove } n' \text{ è un successore di } n$$

Ne segue che $f(n) \leq f(n')$

Note 5.3.1. Se h è consistente la f non decresce mai lungo i cammini, da cui il termine **monotona**.

Teorema 5.3.2. Un'euristica monotona è ammissibile.

Esistono euristiche ammissibili che non sono monotone, ma sono rare. Le euristiche monotone garantiscono che la soluzione meno costosa venga trovata per prima e quindi sono ottimali anche nel caso di ricerca su grafo.

Non si devono recuperare tra gli antenati nodi con costo minore. Lista degli esplorati, stato già esplorato è sul cammino ottimo allora posso evitare di inserire il corrente ripetuto senza perdere l'ottimalità.

```
if figlio.Stato non e in esplorati e non e in frontiera then
    frontiera = Inserisci(figlio, frontiera)
```

Per la frontiera, volendo evitare stati ripetuti, resta 'if' finale di UC:

```
if figlio.Stato e in frontiera con Costo-cammino piu alto then
    sostituisci quel nodo frontiera con figlio
```

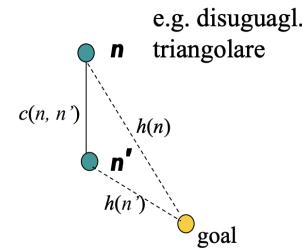
Andiamo ora a verificare l'ottimalità di A^* supponendo di avere il teorema, con h consistente.

1. Se $h(n)$ è consistente i valori di $f(n)$ lungo un cammino sono non decrescenti:

$$\text{se } h(n) \leq c(n, a, n') + h(n') \Rightarrow (\text{def. consistenza sommando } g(n)) \quad g(n) + h(n) \leq g(n) + c(n, a, n') + h(n')$$

ma siccome abbiamo $g(n) + c(n, a, n') = g(n')$ allora

$$g(n) + h(n) \leq g(n') + h(n') \Rightarrow f(n) \leq f(n') \Rightarrow f \text{ monotona}$$



2. Ogni volta che A^* seleziona un nodo (n) per l'espansione, il cammino ottimo a tale nodo è stato trovato: se così non fosse, ci sarebbe un altro nodo m nella frontiera sul cammino ottimo (a n , ancora da trovare con un cammino ottimo), con $f(m)$ minore (per la monotonia e n successore di m); ma ciò non è possibile perché tale nodo sarebbe già stato espanso (si espande prima un nodo con f minore).

3. Quando seleziona nodo goal è cammino ottimo [$h = 0, f = C^*$].

Quindi, detto questo, perché A^* è vantaggioso?

- A^* espande tutti i nodi con $f(n) < C^*$ (C^* = costo ottimo)
- A^* espande alcuni nodi con $f(n) = C^*$.
- **A^* non espande alcun nodo con $f(n) > C^*$**

Quindi alcuni nodi (e suoi sottoalberi) non verranno considerati per l'espansione (ma restiamo ottimali): pruning (h opportuna, più alta possibile tra le ammissibili, fa tagliare molto).

Più f è aderente a stima ottimale, più taglio! Ovali più stretti. Cercheremo quindi una h il più alta possibile tra le ammissibili. Se molto bassa molti (sino a tutti i) nodi restano minore di C^* → espando tutti (a cerchi). Il pruning sotto-alberi è il punto focale: non li abbiamo già in memoria e evitiamo di generarli (decisivo per i problemi di AI a spazio stati esponenziali).

In riassunto L'algoritmo è quello degli schemi usati per UC, Usando $f = g+h$ per la coda con priorità, ove h e g soddisfano quanto allo slide 9 [A], ove h è una funzione euristica ammissibile [A^*], e considerando le condizioni dette per ottenere l'ottimalità su grafi.

- A^* è **completo**: discende dalla completezza di A (A^* è un algoritmo A particolare).
- A^* con euristica monotona è **ottimale**.
- A^* è **ottimamente efficiente**: a parità di euristica nessun altro algoritmo espande meno nodi (senza rinunciare a ottimalità)

I problemi principali sono la scelta dell'euristica e ancora l'occupazione di memoria che nel caso pessimo resta esponenziale come visto per gli altri algoritmi di ricerca con stesso schema, causa frontiera.

5.4 Sotto-casi speciali: US e Greedy Best First

Ci sono due casi particolari dell'algoritmo A:

1. Se $h(n) = 0$ [$f(n) = g(n)$] si ha Uniform Cost (UC), ossia g non basta (si può migliorare).
2. Se $g(n) = 0$ [$f(n) = h(n)$] si ha Greedy Best Fist, ossia h non basta (già visto all'inizio).

5.4.1 UC vs A^*

Illustrazione dell'algoritmo di ricerca Dijkstra per trovare il percorso da un nodo iniziale (in basso sinistra, rosso) a un nodo obiettivo (in alto a destra, verde) in un problema di pianificazione del movimento del robot. I nodi aperti rappresentano l'insieme "provvisorio". I nodi pieni sono quelli visitati, con colore che rappresenta la distanza: più verde, più lontano. Nodi in tutti i diversi le direzioni vengono esplorate in modo uniforme, apparendo come un fronte d'onda più o meno circolare poiché l'algoritmo di Dijkstra utilizza un'**euristica identicamente uguale a 0**. → UC !

Illustrazione dell'algoritmo di ricerca A^* per trovare il percorso da un nodo iniziale a un nodo obiettivo in un robot è un problema di pianificazione del movimento. I cerchi vuoti rappresentano i nodi nell'insieme aperto, cioè, quelli che restano da esplorare e quelli pieni sono nell'insieme chiuso. Colore acceso ogni nodo chiuso indica la distanza dalla partenza: più è verde, più è lontano. Uno

può prima vedere A* muoversi in linea retta in direzione della porta, poi quando colpisce l'ostacolo, esplora percorsi alternativi attraverso i nodi da insieme aperto → frontiera.

L'algoritmo A* è una generalizzazione dell'algoritmo di Dijkstra che riduce le dimensioni del sottografo che deve essere esplorato, se il lower-bound della "distanza" dall'obiettivo (h) è disponibile.

5.5 Costruire le euristiche di A*

Partiamo dalle valutazione di funzioni euristiche. A parità di ammissibilità, una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore (visitare meno nodi). Questo dipende da quanto informata è l'euristica (**dal grado di informazione posseduto**)

- $h(n) = 0$ minimo di informazione (BF o UC)
- $h^*(n)$ massimo di infomrazione (oracolo)

In generale, per le euristiche ammissibili:

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

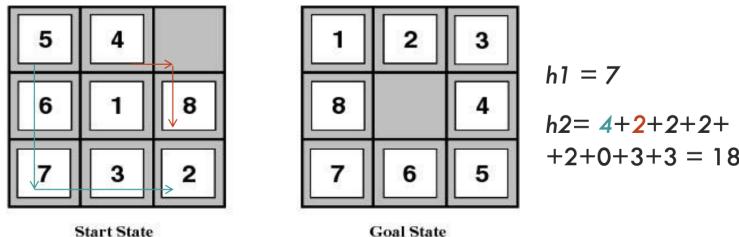
Teorema 5.5.1. Se $h_1 \leq h_2$, i nodi espansi⁴ da A* con h_2 sono un sottoinsieme di quelli espandi da A* con h_1 .

Se $h_1 \leq h_2$, A* con h_2 è almeno efficiente quanto A* con h_1 . Un'euristica più informata (accurata) riduce lo spazio di ricerca (è più efficiente), ma è tipicamente più costosa da calcolare (e.g. un caso estremo ?)

Esempio 5.5.1. Due euristiche ammissibili per il gioco dell'8 potrebbero essere le seguenti:

- h_1 : conta il numero di caselle fuori posto
- h_2 : somma delle distanze **Manhattan** (orizzontale/verticale) delle caselle fuori posto dalla posizione finale.

h_2 è più informata di h_1 , infatti $\forall n . h_1(n) \leq h_2(n)$, quindi si dice che h_2 **domina** h_1 (utile per confrontte tra ammissibili)



Definizione 5.5.1. La somma delle distanze Manhattan si definisce come:

$$h((x, y)) = MD((x, y), (x_g, y_g)) = |x - x_g| + |y - y_g|$$

Ora capiamo come valutare gli algoritmi di ricerca euristica. Introfuciamo il **fattore di diramazione effettivo** b^* , N: numero di nodi generati, d: profondità della soluzione. b^* è il fattore di diramazione di un albero uniforme con $N + 1$ nodi, soluzione dell'equazione:

$$N + 1 = b^* + (b^*)^2 + \cdots + (b^*)^d$$

Sperimentalmente una buona euristica ha un b^* abbastanza vicino a 1 (j1.5)

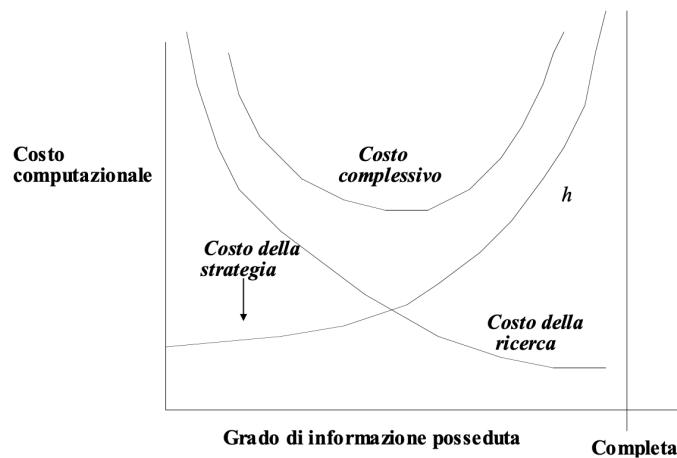


Figure 3: Costo ricerca vs costo euristico

d	ID (appr. it. non inf)	$A^*(h_1)$	$A^*(h_2)$
2	10 (2,43)	6 (1,79)	6 (1,79)
4	112 (2,87)	13 (1,48)	12 (1,45)
6	680 (2,73)	20 (1,34)	18 (1,30)
8	6384 (2,80)	39 (1,33)	25 (1,24)
10	47127 (2,79)	93 (1,38)	39 (1,22)
12	3644035 (2,78)	227 (1,42)	73 (1,24)
14	Nodi generati b^*	539 (1,44)	113 (1,23)
...	-

Esempio 5.5.2. Ricordando l'esempio dal gioco dell'otto: Sono riportati: Nodi generati e fattore di diramazione effettivo (b^* , verde) I dati sono mediati, per ogni d , su 100 istanze del problema [AIMA].

Nella **capacità di esplorazione**, l'influenza di b^* :

- Con $b=2$: $d=6$ e $N = 100$ $d=12$ e $N = 10.0000$
- Con $b=1.5$: $d=12$ $N = 100$ $d=24$ e $N = 10.000$

migliorando di poco l'euristica si riesce, a parità di nodi espansi, a raggiungere una profondità doppia di esplorazione mosse!

5.6 Inventare euristiche

Quindi abbiamo che:

1. Tutti i problemi dell'IA (o quasi) sono di complessità esponenziale ... (nel generare nodi, i.e. configurazioni possibili) ma c'è esponenziale e esponenziale!
2. L'euristica può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca.
3. Migliorando anche di poco l'euristica si riesce ad esplorare uno spazio molto più grande (più in profondità).

⁴Ricorda che A^* espande tutti i nodi con $f(n) = g(n) + h(n) < C^*$, e sono meno per h maggiore (h maggiore fa andare più nodi oltre C^*).

Per inventare un'euristica ci sono alcune strategie, che aiutano appunto ad ottenere euristiche ammissibili:

- Rilassamento del problema
- Massimizzazione di euristiche
- Database di pattern disgiunti
- Combinazione lineare
- Apprendere dall'esperienza

5.6.1 Rilassamento problema

Esempio 5.6.1. Il rilassamento del problema nel gioco dell'8 mossa da A a B possibile se:

1. **B adiacente a A**
2. **B libera**

h_1 e h_2 sono calcoli distanza esatta della soluzione in versioni semplificate del puzzle:

- h_1 (nessuna restrizione, ne 1 ne 2): sono sempre ammessi scambi a piacimento tra caselle (si muove ovunque) → numero caselle fuori posto.
- h_2 (solo restrizione 1): sono ammessi spostamenti anche su caselle occupate, purché adiacenti → somma delle distanze Manhattan.

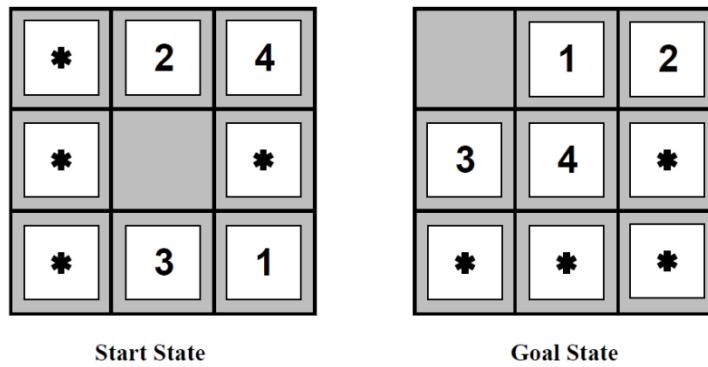
5.6.2 Massimizzazione di euristiche

Se si hanno una serie di euristiche ammissibili h_1, h_2, \dots, h_k senza che nessuna "domini" un'altra allora conviene prendere il massimo dei loro valori:

$$h(n) = \max(h_1(n), h_2(n), \dots, h_k(n))$$

Se le h_i sono ammissibili, anche la h lo è. La h domina tutte le altre.

5.6.3 Database con pattern disgiunti



Costo della soluzione ottima al sottoproblema (di sistemare 1,2,3,4) è una sottostima del costo per il problema nel suo complesso (e.g. rilevatesi più accurata della Manhattan).

Database di pattern: memorizzare ogni istanza del sottoproblema con relativo costo della soluzione. Usare poi questo database per calcolare h_{DB} (estraendo dal DB la configurazione corrispondente allo stato completo corrente).

- **Domanda:** Potremmo poi fare la stessa cosa per altri sottoproblemi: 5-6-7-8, 2-4-6-8, ottenendo altre euristiche ammissibili, poi prendere il valore massimo: ancora una euristica ammissibile. Ma potremmo sommarle e ottenere un'euristica ancora più accurata?

- **Risposta:** In generale no perché le soluzioni ai sottoproblemi interferiscono (condividono alcune mosse, se sposto 1-2-3-4, spostero anche 4-5-6-7) e la somma delle euristiche in generale non è ammissibile (potremmo sovrastimare avendo avuto aiuti mutui). Si deve eliminare il costo delle mosse che contribuiscono all'altro sottoproblema. Database di pattern disgiunti consentono di sommare i costi (euristiche additive) [e.g. solo costo mosse su1-2-3-4], sono molto efficaci: gioco del 15 in pochi ms ma per esempio difficile scomporre per cubo Rubik.

5.6.4 Apprendimento dall'esperienza

Bisogna eseguire un apprendimento dall'esperienza. Quindi far girare il programma, raccogliere dati: coppie $\langle \text{stato}, h^* \rangle$. Usare i dati per apprendere a predire la h con algoritmi di apprendimento induttivo (da istanze note stimiamo h in generale).

Gli algoritmi di apprendimento si concentrano su caratteristiche salienti dello stato (feature, x_i) [e.g. apprendiamo che da numero tasselli fuori posto 5 \rightarrow costo 14, etc].

5.6.5 Combinazione euristiche

Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare per combinare le euristiche:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) + \dots + c_k x_k(n)$$

Esempio 5.6.2. Gioco dell'8: $h(n) = c_1 \# \text{fuori-posto} + c_2 \# \text{coppie-scambiate}$

Scacchi: $h(n) = c_1 \text{ vant-pezzi} + c_2 \text{ pezzi-attacc.} + c_3 \text{ regina} + \dots$

Il peso dei coefficienti può essere aggiustato con l'esperienza, anche qui apprendendo automaticamente da esempi di gioco. $h(goal) = 0$ (e.g. gioco dell'8) ma ammissibilità e consistenza non automatiche.

5.7 Algoritmi evoluti basati su A^*

Ci sono una serie di algoritmi basati su A^* che possono andare a portare ad un miglioramento dell'occupazione della memoria. Fra questi abbiamo: Beam search, A^* con approfondimento iterativo (IDA*), ricerca best-first ricorsiva (RBFS), A^* con memoria limitata (MA*) in versione semplice (SMA*).

5.7.1 Beam search

Nel Best First viene tenuta tutta la frontiera; se l'occupazione di memoria è eccessiva si può ricorrere ad una variante: la Beam search. La Beam Search tiene ad ogni passo solo i k nodi più promettenti, dove k è detto l'ampiezza del raggio (beam). La Beam Search non è completa.

5.7.2 IDA*

L'IDA* è un A^* con approfondimento iterativo. IDA* combina A^* con ID: ad ogni iterazione si ricerca in profondità con un limite (cut-off) dato dal valore della funzione f (e non dalla profondità) il limite f -limit viene aumentato ad ogni iterazione, fino a trovare la soluzione. Punto critico: di quanto viene aumentato f -limit.

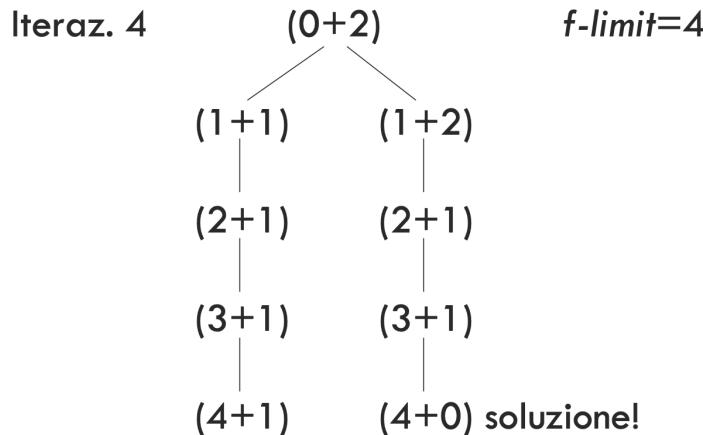
Esempio 5.7.1.

Cruciale la scelta dell'incremento per garantire l'ottimalità:

- Nel caso di costo delle azioni fisso è chiaro: il limite viene incrementato del costo delle azioni.
- Nel caso che i costi delle azioni siano variabili? O costo minimo, oppure si potrebbe ad ogni passo fissare il limite successivo al valore minimo delle f scartate (in quanto superavano il limite) all'iterazione precedente.

IDA* è sia completo che ottimale.

- Se le azioni hanno costo costante k (caso tipico 1) e f -limit viene incrementato di k .

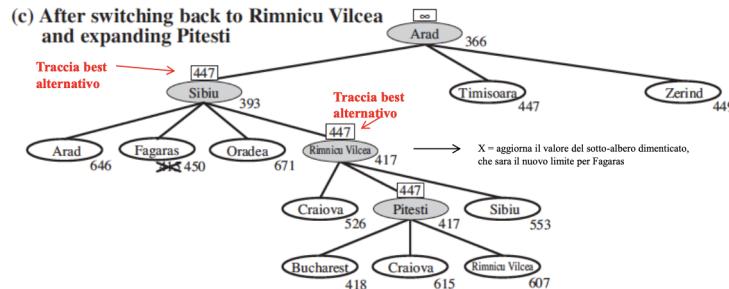


- Se le azioni hanno costo variabile e l'incremento di f-limit è $\leq \epsilon$ (minimo costo degli archi).
- Se il nuovo f-limit = min. valore f dei nodi generati ed esclusi all'iterazione precedente.

L'occupazione di memoria è $O(bd)$.

5.7.3 Best-first ricorsivo (BRFS)

Simile a DF ricorsivo: cerca di usare meno memoria, facendo del lavoro in più. Tiene traccia ad ogni livello del **migliore percorso alternativo**. Invece di fare backtracking in caso di fallimento (DF si ferma solo in fondo) interrompe l'esplorazione quando trova un nodo meno promettente (secondo f). Nel tornare indietro si ricorda il miglior nodo che ha trovato nel sottoalbero esplorato, per poterci eventualmente tornare Memoria: lineare nella profondità delle sol. ottima.



Esempio 5.7.2.

```

function Ricerca-Best-First-Ricorsiva(problema)
    returns soluzione oppure fallimento
    // all'inizio f-limite e un valore molto grande
    return RBFS(problema, CreaNodo(problema.Stato-iniziale), infinito)

function RBFS (problema, nodo, f-limite)
    // restituisce due valori
    returns soluzione oppure fallimento e un nuovo limite all f-costo
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    successori = [ ]
    for each azione in problema.Azioni(nodo.Stato) do
        aggiungi Nodo-Figlio(problema, nodo, azione) a successori // genera i successori
  
```

```

if successori vuoto then return fallimento, infinito

for each s in successori do // valuta i successori
    s.f = max(s.g + s.h, nodo.f) // un modo per rendere monotona f
loop do
    migliore = il nodo con f minimo tra i successori
    if migliore.f > f_limite then return fallimento, migliore.f
    alternativa = il secondo nodo con f minimo tra i successori
    risultato, migliore.f = RBFS(problema, migliore, min(f_limite, alternativa))
    if risultato != fallimento then return risultato

```

5.7.4 A^* con memoria limitata (versione semplice)

L'idea è quella di utilizzare al meglio la memoria disponibile. SMA^* procede come A^* fino ad esaurimento della memoria disponibile. A questo punto **“dimentica” il nodo peggiore**, dopo avere aggiornato il valore del padre. A parità di f si sceglie il nodo migliore più recente e si dimentica il nodo peggiore più vecchio. Ottimale se il cammino soluzione sta in memoria.

In conclusione in algoritmi a memoria limitata (IDA* e SMA*) le limitazioni della memoria possono portare a compiere molto lavoro inutile [esp. ripetuta stessa nodi]. Difficile stimare la complessità temporale effettiva. Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale.

6 Ricerca locale

Gli agenti risolutori di problemi “classici” assumono: ambienti completamente osservabili, ambienti deterministici, sono nelle condizioni di produrre offline un piano (una sequenza di azioni) che può essere eseguito senza imprevisti per raggiungere l’obiettivo.

La ricerca sistematica, o anche euristica, nello spazio di stati è troppo costosa. Inoltre spesso le assunzioni sull’ambiente sono da riconsiderare infatti in ambienti realistici le azioni sono non deterministiche e ambiente parzialmente osservabile oppure abbiamo addirittura ambienti sconosciuti e problemi di esplorazione (ess. ricerca online).

Per effettuare una **ricerca locale** bisogna fare alcune assunzione. Gli algoritmi visti esplorano gli spazi di ricerca alla ricerca di un goal e restituiscono un cammino soluzione, ma a volte lo stato goal è la soluzione del problema. Gli algoritmi di ricerca locale sono adatti per problemi in cui:

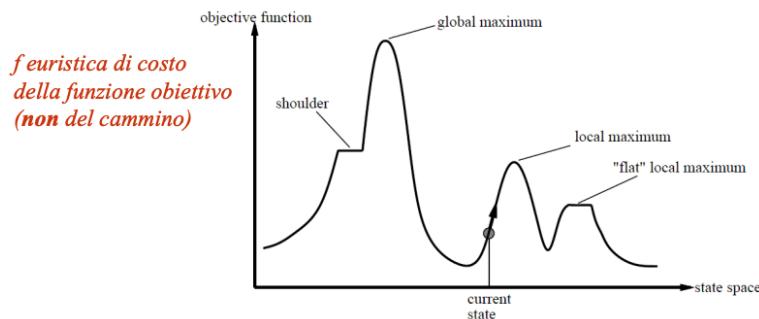
- La sequenza di azioni non è importante: quello che conta è unicamente lo stato goal.
- Tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati (Es. le regine nella formulazione a stato completo). Ci interessa di ottenere la soluzione ma non il path

Gli algoritmi di ricerca locale inoltre non sono sistematici, tengono traccia solo del nodo corrente e si spostano su nodi adiacenti. Non tengono traccia dei cammini (non servono in unscita!).

1. Efficienti in occupazione di memoria.
2. Possono trovare soluzioni ragionevoli anche in spazi molto grandi e infiniti, come nel caso di spazi continui.

Sono molto utili per risolvere problemi di ottimizzazione: lo stato migliore secondo una funzione obiettivo (f), lo stato di costo minore (ma non il path).

Esempio 6.0.1. Minimizzare numero di regine sotto attacco (f all’obiettivo?) oppure training di un modello di Machine Learning.



Uno stato ha una posizione sulla superficie e una altezza che corrisponde al valore della f. di valutazione (f. obiettivo). Un algoritmo provoca movimento sulla superficie. Trovare l’avvallamento più basso (e.g. min costo) o il picco più alto (e.g. max di un obiettivo).

6.1 Ricerca in salita (Hill climbing)

È una ricerca locale di tipo **greedy**. Vengono generati i successori e valutati; viene scelto un nodo che migliora la valutazione dello stato attuale (non si tiene traccia degli altri [no albero di ricerca in memoria]).

- Il migliore fra i successori è **Hill climbing a salita rapida/ripida**.
- Uno a caso (tra quelli che salgono) si dice **Hill climbing stocastico** (anche dipendendo da pendenza)

- Mentre il primo è il **Hill climbing con prima scelta** (il primo generato tra tanti possibili)

Se non ci sono stati successori migliori l'algoritmo termina con fallimento.

```
function Hill-climbing (problema)
    returns uno stato che e un massimo locale [esercizio: fare con min.]
    nodo-corrente = CreaNodo(problema.Stato-iniziale)

    loop do
        vicino = il successore di nodo-corrente di valore piu alto
        if vicino.Valore <= nodo-corrente.Valore then
            return nodo-corrente.Stato // interrompe la ricerca
        nodo-corrente = vicino
        // (altrimenti, se vicino e migliore, continua)
```

Note 6.1.1. Si prosegue solo se il vicino (più alto) è migliore dello stato corrente → se tutti i vicini sono peggiori o uguali si ferma.

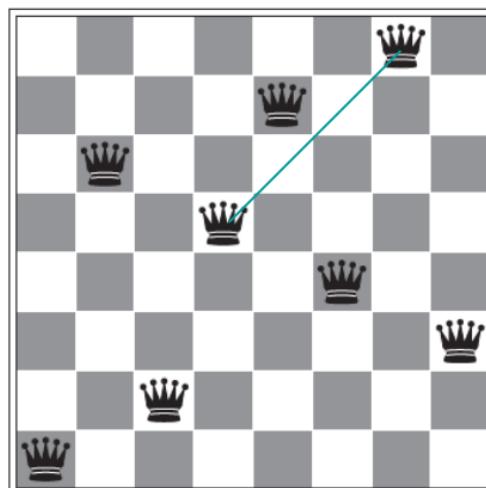
Non c'è frontiera a cui ritornare, si tiene 1 solo stato. Il tempo si calcolo come numero di cicli variabile in base al punto di partenza. Il codice in python è il seguente:

```
def hill_climbing(problem): """ Ricerca locale - Hill-climbing """
    current = Node(problem.initial_state)
    while True:
        neighbors = [current.child_node(problem, action) for action in
                     problem.actions(current.state)]
        if not neighbors:
            # se current non ha successori esci e restituisci current
            break

        # scegli il vicino con valore piu' alto (sulla funzione problem.value)
        neighbor = sorted(neighbors, key = lambda x: problem.value(x), reverse = True)[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor # (altrimenti, se vicino e migliore, continua)
    return current
```

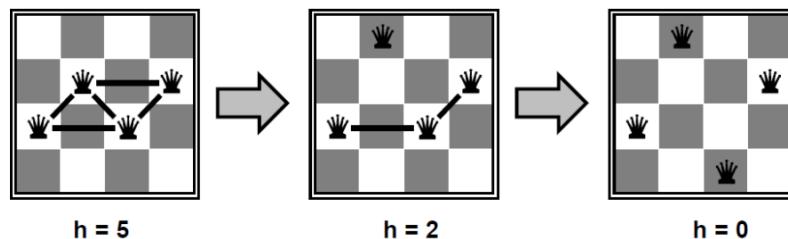
Esempio 6.1.1. Il problema delle 8 regine. Costo h (stima euristica del costo f): numero di coppie di regine che si attaccano a vicenda (nell'esempio valore 17). Si cerca il minimo, i numeri sono i valori dei successori (7×8) [7 posizioni per ogni regina, su ogni colonna], tra i migliori (di pari valore 12) si sceglie a caso, i imposta anche il minimo globale a 0.

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	15	13	16	13
17	14	17	15	15	14	16	16
18	15	16	18	15	15	15	15
14	14	14	15	15	14	14	16
14	14	13	17	12	14	12	18

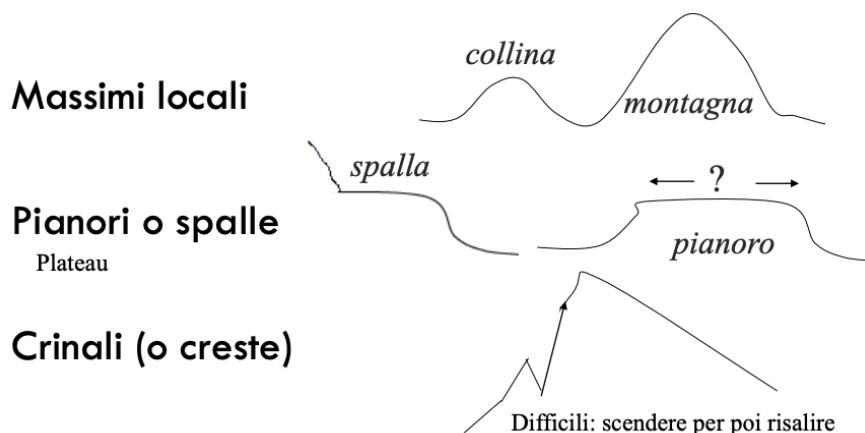


Vediamo poi nella seconda figura un esempio negativo con minimo locale. Settiamo $h = 1$, tutti gli stati successori non migliorano la situazione. Per le 8 regine Hill-climbing si blocca l'86% delle volte, ma in media solo 4 passi per la soluzione e 3 quando si blocca, su $8^8 = 16.8$ milioni di stati.

Esempio 6.1.2. Sempre sulla stessa base dell'esempio sopra questo è un esempio positivo, con successo in tre mosse. h qui è l'euristica di costo della funzione obiettivo (da minimizzare).



Alcuni problemi con Hill-climbing ci sono se la f è da ottimizzare, infatti in questo caso i picchi sono massimi locali o soluzioni ottimali.



Alcuni miglioramenti che si possono apportare solo i seguenti:

1. Consentire (un numero limitato di) mosse laterali (ossia ci si ferma per $<$ nell'alg. invece che per \leq continua anche a parità di h). L'algoritmo sulle 8 regine ha successo nel 94%, ma impiega in media 21 passi.
2. Hill-climbing stocastico: si sceglie a caso tra le mosse in salita (magari tenendo conto della pendenza). Converge più lentamente ma a volte trova soluzioni migliori.
3. Hill-climbing con prima scelta. Può generare le mosse a caso, uno alla volta, fino a trovarne una migliore dello stato corrente (si prende solo il primo che migliora). Come la stocastica ma utile quando i successori sono molti (e.g. migliaia o ben oltre), evitando una scelta tra tutti.
4. Hill-Climbing con riavvio casuale (random restart): ripartire da un punto scelto a caso. Se la probabilità di successo è p saranno necessarie in media $1/p$ ripartenze per trovare la soluzione (es. 8 regine, $p = 0.14 \rightarrow 7$ ripartenze $[1/p]$ per avere 6 fail e un successo). Per le regine: caso con 3 milioni di regine in meno di un minuto! Se funziona o no dipende molto dalla forma del panorama degli stati (molti min loc. abbassano p , si blocca spesso)

6.2 Tempra simulata

L' algoritmo di tempra simulata (Simulated annealing) [Kirkpatrick, Gelatt, Vecchi 1983] combina hill-climbing con una scelta stocastica (ma non del tutto casuale, perché poco efficiente...). Analogia con il processo di tempra dei metalli in metallurgia. I metalli vengono portati a temperature molto

elevate (alta energia/stocasticità iniziale) e raffreddati gradualmente consentendo di cristallizzare in uno stato a (più) bassa energia. (esempio di cross-fertilization tra aree scientifiche diverse).

In questo algoritmo ad ogni passo si sceglie un successore n' a caso:

- se migliora lo stato corrente viene espanso.
- se no (caso in cui $\Delta E = f(n') - f(n) \leq 0$) quel nodo viene scelto con probabilità

$$p = e^{\Delta E / T} [0 \leq p \leq 1]$$

[Si genera un numero casuale tra 0 e 1: se questo è $< p$ il successore viene scelto, altrimenti no].

Ossia: p è inversamente proporzionale al peggioramento, infatti se la mossa peggiora molto, ΔE alto neg., la p si abbassa. T (temperatura) decresce col progredire dell'algoritmo (quindi anche p) secondo un piano definito. Col progredire rende improbabili le mosse peggiorative.

Per quanto riguarda l'analisi della tempra simulata, La probabilità p di una mossa in discesa diminuisce col tempo e l'algoritmo si comporta sempre di più come Hill Climbing. Se T viene decrementato abbastanza lentamente con prob. tendente ad 1 si raggiunge la soluzione ottimale. Analogia col processo di tempra dei metalli T corrisponde alla temperatura ΔE alla variazione di energia.

I parametri da inserire sono il valore iniziale e decremento di T , ed i valori per T determinati sperimentalmente: il valore iniziale di T è tale che per valori medi di ΔE , $p = e^{\Delta E / T}$ sia all'incirca 0.5.

6.3 Ricerca local beam

La versione locale della beam search. Si tengono in memoria k stati, anziché uno solo ed ad ogni passo si generano i successori di tutti i k stati. Se si trova un goal ci si ferma, altrimenti si prosegue con i k migliori tra questi

Note 6.3.1. Diverso da K restart (che riparte da zero), diverso anche da beam search

Si introduce un elemento di casualità, come in un processo di selezione naturale (diversificare la nuova generazione). Nella **variante stocastica** della local beam, si scelgono k successori, ma con probabilità maggiore per i migliori. Tra le terminologie usate abbiamo: organismo [stato], progenie [successori], fitness [il valore della f], idoneità.

6.4 Algoritmi generici/evolutivi

Sono varianti della beam search stocastica in cui gli stati successori sono ottenuti combinando due stati genitore (anziché per evoluzione). Un po' di terminologia che si userà: **popolazione di individui** [stati], **fitness**, **accoppiamenti + mutazione genetica, generazioni**.

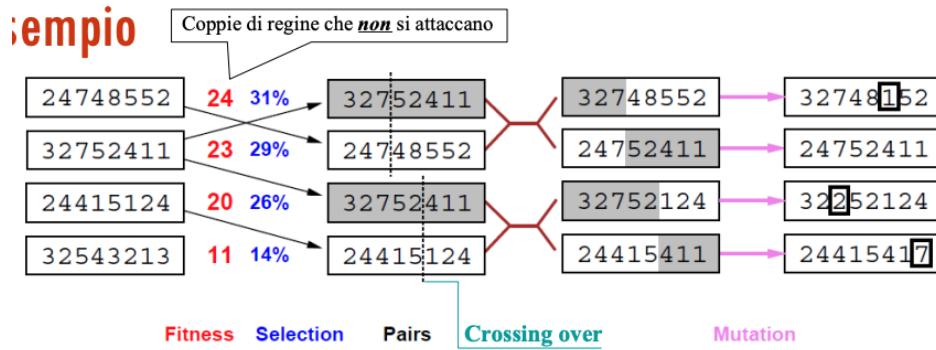
Allora per spiegare il funzionamento vediamo insanzitutto che la **popolazione** iniziale è formata da:

- k stati/**individui** generati casualmente.
- ogni individuo è rappresentato come una stringa (esempio: posizione nelle colonne ("24748552") stato delle 8 regine o con 24 bit*).

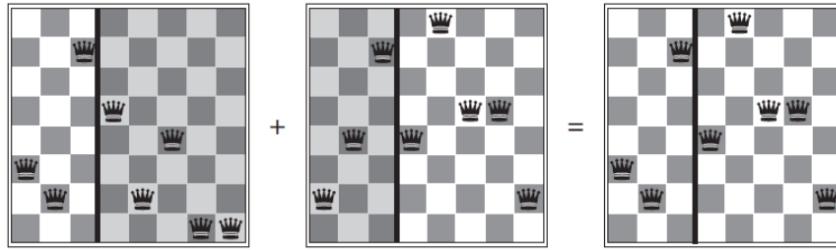
A questo punto gli individui sono valutati da una **funzione di fitness** (Esempio: n. di coppie di regine che non si attaccano).

Poi si **selezionano** gli individui per gli **"accoppiamenti"** con una probabilità proporzionale alla fitness. Le coppie danno vita alla **generazione** successiva combinando materiale genetico (crossover) con un meccanismo aggiuntivo di mutazione genetica (casuale). La popolazione ottenuta dovrebbe essere migliore. La cosa si ripete fino ad ottenere stati abbastanza buoni (stati obiettivo) o finché non miglioriamo più.

Esempio 6.4.1. Per ogni coppia (scelta con probabilità(blu) proporzionale alla fitness(rosso)) viene scelto un punto di crossing over(verde) e vengono generati due figli scambiandosi pezzi (del DNA)



Viene infine effettuata una mutazione(rosa) casuale che dà luogo alla prossima generazione, la fitness progressivamente tenderà a favorire generazioni migliori. Emula i meccanismi genetici ma anche l'evoluzione della specie. La nascita di un figlio avviene come:



Le parti chiare sono passate al figlio, le parti grigie si perdono, se i genitori sono molto diversi anche i nuovi stati sono diversi All'inizio spostamenti maggiori che poi si raffinano.

In conclusione gli algoritmi genetici sono suggestivi (area del Natural computing: e.g. swarm, ...). Usati in molti problemi reali e.g. configurazione di circuiti e scheduling di lavori, fra i loro vantaggi abbiamo che combinano: tendenza a salire della beam search stocastica, interscambio info (indirettamente) tra thread paralleli di ricerca (blocchi utili che si combinano). Funziona meglio se il problema (soluzioni) ha componenti significative rappresentate in sottostringhe. Il maggior punto critico è la rappresentazione del problema in stringhe.

6.5 Spazi continui

Molti casi reali hanno spazi di ricerca continua e.g. fondamentale per Machine Learning! Stato descritto da variabili continue x_1, \dots, x_n , vettore x . Prendiamo ad esempio movimenti in spazio 3D, con posizione data da $x = (x_1, x_2, x_3)$, apparentemente ostico, fattori di ramificazione infiniti con gli approcci precedenti, in realtà molti strumenti matematici per spazi continui, che portano ad approcci anche molto efficienti...

Se la f è continua e differenziabile, e.g. quadratica rispetto a x (vettore). Il minimo o massimo si può cercare utilizzando il **gradiente**, che restituisce la direzione di massima pendenza nel punto. Data f obiettivo su 3D

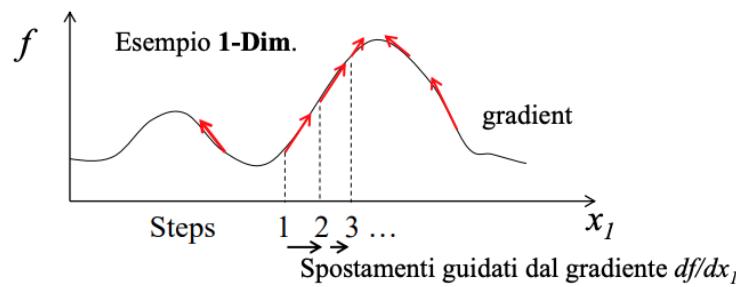
$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3} \right)$$

Nell Hill climbing iterativo:

$$x_{new} = x + \eta \nabla f(n)$$

Dove il '+' per salire (maximization), il '-' per scendere (minimization), la η indica lo step size (e.g. costante positiva <1) Quantifica direzione e spostamento, senza cercarlo tra gli infiniti possibili successori!

Note 6.5.1. non sempre è necessario il min/max assoluto: vedremo nel ML



Esempio 6.5.1. Discesa verso il minimo. $f(x) = x^2 \quad f'(x) = 2x$

Discesa di gradiente verso il minimo.

$$x_{new} = x - \eta \nabla f(x) \rightarrow (1 - d) \rightarrow x_{new} = x - \eta f'(x)$$

Mettiamoci in $x = 2$, la derivata vale $2 \times 2 = 4$, mi devo spostare di $-\eta f'(x) = -\eta 4$, quindi ad esempio ($\eta = 0.2$) ottengo $-\eta f'(x) = -0.2 \times 4 = -0.8$ andando a sinistra al punto $x_{new} = 2 - 0.8 = 1.2$ avvicinandomi quindi al minimo.

6.6 Assunzioni sull'ambiente da considerare

Negli ambienti più realistici gli agenti risolutori di problemi “classici” assumono:

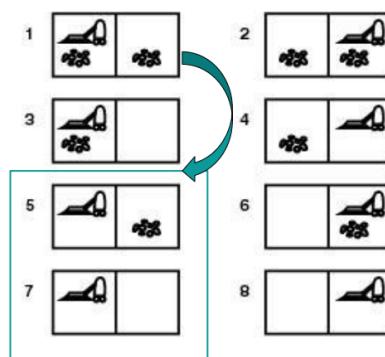
- Ambienti completamente osservabili.
- Azioni/ambienti deterministici.
- Il piano generato è una sequenza di azioni che può essere generato offline e eseguito senza imprevisti.
- Le percezioni non servono se non nello stato iniziale.

In un ambiente parzialmente osservabile e non deterministico le **percezioni** sono importanti: restringono gli stati possibili, informano sull'effetto dell'azione. Più che un piano l'agente può elaborare una ”strategia”, che tiene conto delle diverse eventualità: un **piano con contigenza**.

Esempio 6.6.1. L'aspirapolvere con assunzioni diverse.

L'aspirapolvere imprevedibile: ci sono più stati possibili risultato dell'azione.

- Comportamento: Se aspira in una stanza sporca, la pulisce ma talvolta pulisce anche una stanza adiacente. Se aspira in una stanza pulita, a volte rilascia sporco.
- Variazioni necessarie al modello: Il modello di transizione restituisce un **insieme di stati**: l'agente non sa in quale si troverà. Il piano di **contingenza** sarà un piano condizionale e magari con cicli.



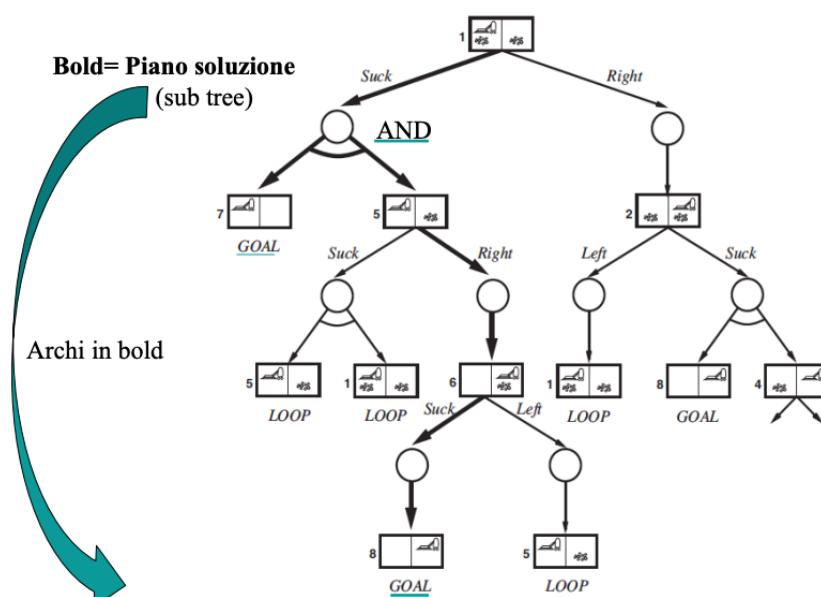
Abbiamo che Risultati(aspira, 1) = {5, 7}. Il piano possibile potrebbe essere:

```
Aspira, if stato = 5
    then [Destra, Aspira]
    else []
```

La soluzione è da sequenza di azioni a piano (albero).

6.6.1 Alberi di ricerca AND-OR

Nodi OR le scelte dell'agente [1 sola azione]. **Nodi AND** le diverse contingenze (le scelte dell'ambiente, più stati possibili), da considerare tutte. Una soluzione a un problema di ricerca ANDOR è un albero che: ha un nodo obiettivo in ogni foglia specifica un'unica azione nei nodi OR include tutti gli archi uscenti da nodi AND (tutte le contingenze).



Esempio 6.6.2. Un piano di ricerca potrebbe essere il seguente:

```
Aspira, if stato = 5 then [Destra, Aspira] else []
```

7 Agenti basati su coscenza

Abbiamo già visto agenti con stato e con obiettivo in mondi osservabili con stati atomici e azioni descrivibili in maniera semplice, mettendo enfasi sul processo di ricerca.

Ora andremo ad parlare di come migliorare le **capacità di ragionamento** degli agenti, dotandoli di rappresentazioni di mondi complessi e astratti, non descrivibili semplicemente. Agenti **basati su conoscenza**, dotati di una KB (**Knowledge Base**) con conoscenza espressa in maniera esplicita e dichiarativa.

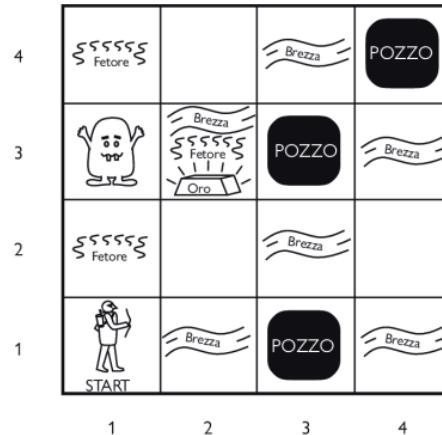
7.1 Introduzione

La maggior parte dei problemi di I.A. sono “knowledge intensive”. Il mondo è tipicamente complesso: ci serve una rappresentazione **parziale e incompleta** (un’astrazione) del mondo funzionale agli scopi dell’agente. Per ambienti parzialmente osservabili e complessi ci servono linguaggi di rappresentazione della conoscenza più espressivi e capacità inferenziali. La conoscenza può essere codificata a mano ma anche estratta dai testi o appresa dall’esperienza o estratta/elicitata dagli esperti.

La KB racchiude tutta la conoscenza necessaria a decidere l’azione da compiere in forma **dichiarativa**. Un agente basato su conoscenza può essere costruito dicendogli (TELL) ciò che deve sapere. Si inizia con una base di conoscenza vuota si aggiungono progressivamente formule alla base di conoscenza, una alla volta.

L’alternativa (approccio procedurale) è scrivere un programma che implementa il processo decisionale, una volta per tutte. Un agente KB è più flessibile: più semplice acquisire conoscenza incrementalmente e modificare il comportamento con l’esperienza.

Esempio 7.1.1. Il mondo del Wumpus. Il mondo del Wumpus è una caverna fatta di stanze connesse tra di loro da passaggi. Il wumpus è una bestia puzzolente che mangia chiunque entri nella stanza in cui si trova. Il wumpus può essere ucciso dall’agente, che ha solo una freccia a disposizione.



- Ci sono stanze con dei pozzi: se l’agente entra in una di queste stanze cade nel pozzo e muore. Il Wumpus non muore nel pozzo.
- In una delle stanze si trova l’oro e l’obiettivo dell’agente è di trovare l’oro e tornare a casa con l’oro, sano e salvo.
- L’agente non conosce l’ambiente, né la sua locazione. Solo all’inizio sa dove si trova (in [1,1]).

Abbiamo poi delle misure di prestazioni:

- +1000 se trova l’oro, trona in [1,1] ed esce.

- -1000 se muore.
- -1 per ogni azione.
- -10 se usa la freccia.

L'ambiente è strutturato in una griglia 4 x 4 di stanze, circondata da pareti di delimitazione. L'agente comincia sempre dalla posizione [1,1], rivolto verso est (dx) ([1,1] è safe). Le posizioni dell'oro e del Wumpus sono scelte casualmente tra tutti I riquadri tranne quello iniziale. Tutti I riquadri (tranne quello iniziale) hanno una probabilità' 0.2 di contenere un pozzo.

Le **azioni** disponibili sono: avanti, a destra di 90°, a sinistra di 90°, afferra un oggetto, scaglia la freccia (solo una), esce. Le cose che vengono **percepite** (sesori) sono:

- fetore nelle caselle adiacenti al Wumpus.
- brezza nelle caselle adiacenti ai pozzi.
- Luccichio nelle caselle con l'oro.
- Bump se sbatte in un muto
- Urlo se il wumpus viene ucciso.
- L'agente NON percepisce la sua locaizone.

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	2,1	3,1	4,1
A OK			

(a)

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	2,1	3,1	4,1
V OK			

(b)

1,4	2,4	3,4	4,4
X ³ W!	2,3	3,3	4,3
1,2	2,2	3,2	4,2
A F OK			
1,1	2,1	3,1	4,1
V OK			

(a)

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
F V OK			
1,1	2,1	3,1	4,1
V OK			

(b)

(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

1. La percesione sono quintuple: [fetore, brezza, luccichio, bump, urlo].
Percesione [1,1] = [none, none,none, none], non essendoci ne brezza ne fetore in [1,1] allora [1,2] e [2,1] sono sicure (OK), l'agente decide di spostarsi in [2,1].
2. Percezione in [2,1] [none,Brezza,none,none,none].
L'agente percepisce una Brezza. Quindi c'è un pozzo in [2,2] o [3,1] (P?). L'agente torna in [1,1] e poi si sposta in [1,2].
3. Percezione in [1,2]: [Fetore,none,none,none,none]
Il Wumpus non può essere in [1,1] (safe per definizione), né in [2,2] (altrimenti avrebbe percepito Fetore anche in [2,1]). Quindi è in [1,3].
Siccome non c'è Brezza in [1,2], [2,2] è OK e ci deve essere un Pozzo in [3,1].
4. L'agente si sposta in [2,2] e poi in [2,3]. Percezione in [2,3] [Fetore, Brezza, Luccichio, none,none]. Percepisce un Luccichio, afferra l'Oro e torna sui suoi passi, percorrendo caselle OK.

7.2 Agenti basati sulla conoscenza

Un agente basato su conoscenza mantiene una **base di conoscenza** (KB): un insieme di **enunciati** (formule) espressi in un linguaggio di rappresentazione. Interagisce con la KB mediante una interfaccia funzionale Tell-Ask:

- Tell: per aggiungere nuovi enunciati a KB.
- Ask: per interrogare la KB.
- Retract: per eliminare enunciati.

Gli enunciati nella KB rappresentano le **opinioni/credenze dell'agente**. Le risposte α devono essere tali che α **discende necessariamente** dalla KB.

Il problema: data una base di conoscenza KB, contenente una rappresentazione dei fatti che si **ritengono veri**, come dedurre che un certo fatto a è vero di conseguenza?

$$KB \models \alpha \quad \text{conseguenza logica}$$

```
//Prende in input una percezione e restituisce un azione
function Agente-KB (percezione) returns un azione
    //Mantiene in memoria una base di conoscenza KB che puo contenere una conoscenza
    //iniziale.
    persistent: KB, una base di conoscenza
    t, un contatore, inizialmente a 0, che indica il tempo

    // Comunica la percezione all KB
    TELL(KB, Costruisci-Formula-Percezione(percezione, t))

    // Chiede alla KB quale azioni deve compiere
    azione <- ASK(KB, Costruisci-Query-Azione( t ))

    // Comunica alla KB che l azione e stata compiuta al tempo t
    TELL(KB, Costruisci-Formula-Azione(azione, t ))
    t <- t + 1
    return azione
```

Base di conoscenza vs base di dati:

- **base di dati**: solo fatti specifici, solo recupero (retrieval).
- **Base di conoscenza**: una rappresentazione esplicita, parziale e compatta, in un linguaggio simbolico, che contiene: fatti di tempo specifico (casella [1,1] ok, c'è un posso in [3,1]), fatti di tipo generale, o regole (c'è brezza nella caselle adiacenti ai pozzi).

Quello che caratterizza una KB è la capacità inferenziale, derivare nuovi fatti da quelli memorizzati esplicitamente (es. c'è un pozzo in [3,1] il wumpus è in [1,3]).

Sfortunatamente più il linguaggio è **espressivo**, meno **efficiente** è il meccanismo inferenziale. § Il problema ‘fondamentale’ in R.C. è trovare il giusto compromesso tra: **espressività** del linguaggio di rappresentazione, **complessità** del meccanismo inferenziale.

Questi due obiettivi sono in contrasto e si tratta di mediare tra queste due esigenze.

7.3 Logica

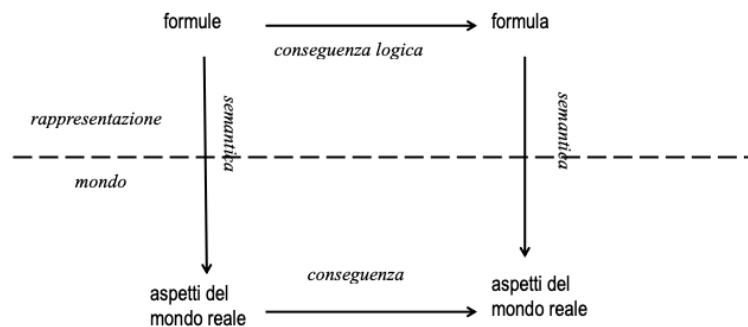
Le basi di conoscenza sono costituite da enunciati (formule). Le formule sono espresse secondo le regole della sintassi, che specifica quali di esse sono “ben formate”. La semantica di una formula esprime il “significato” della formula. Un modello è una configurazione dei valori di verità che si possono assegnare alle variabili coinvolte in una formula.

Un formalismo per la rappresentazione della conoscenza ha tre componenti:

1. Una **sintassi**: un linguaggio composto da un vocabolario e regole per la formazione delle frasi (enunciati, formule).
2. Una **semantica**: stabilisce una corrispondenza tra gli enunciati e i fatti del mondo; se un agente ha un enunciato α nella KB, crede che il fatto corrispondente sia vero nel mondo.
3. Un **meccanismo inferenziale** (codificato o meno tramite regole di inferenza come nella logica) che ci consente di inferire nuovi fatti.

Definizione 7.3.1 (Semantica). *La semantica specifica le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello.*

Nella logica proposizionale, un modello specifica il valore di verità (True o False) di ogni simbolo proposizionale.



Le formule sono configurazioni fisiche dell'agente. Il ragionamento è il processo di costruzione di nuove configurazioni partendo dalle vecchie. Il ragionamento logico dovrebbe assicurare che le nuove configurazioni rappresentino aspetti del mondo che sono effettive conseguenze degli aspetti del mondo rappresentati dalle vecchie configurazioni.

8 Logica Proposizionale

8.1 Sintassi

Definizione 8.1.1 (Sintassi). *La sintassi definisce quali sono le frasi leggittime (ben formate) del linguaggio.*

$$\begin{aligned}
 formula &\longrightarrow formulaAtomica \mid formulaComplessa \\
 formulaAtomica &\longrightarrow True \mid False \mid simbolo \\
 simbolo &\longrightarrow P \mid Q \mid R \mid \dots \\
 formulaComplessa &\longrightarrow \neg formula \text{ not (negazione)} \\
 &\quad \mid (formula \wedge formula) \text{ and (congiunzione)} \\
 &\quad \mid (formula \vee formula) \text{ or (disgiunzione)} \\
 &\quad \mid (formula \Rightarrow formula) \text{ implicazione} \\
 &\quad \mid (formula \Leftrightarrow formula) \text{ se e solo se}
 \end{aligned}$$

Esempio 8.1.1. $((A \wedge B) \Rightarrow C)$. Possiamo ottenere le parentesi assumendo questa precedenza tra gli operatori:

$$\neg > \wedge > \vee > \Rightarrow, \Leftrightarrow$$

$\neg P \wedge Q \vee R \Rightarrow S$ è la stessa cosa di $((\neg P) \vee (Q \wedge R)) \Rightarrow$. Per esempio dal mondo del Wumpus $P_{1,1}$ c'è il posso in $[1, 1]$, $W_{2,3}$ il wumpus è in $[2, 3]$.

8.2 Semantica

Definizione 8.2.1. *La semantica specifica le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello.*

Nella logica proposizionale, un modello specifica il valore di verità (True o False) di ogni simbolo proposizionale.

La semantica ha a che fare col significato delle frasi: definisce se un enunciato è vero o falso rispetto a una interpretazione (mondo possibile). Una interpretazione definisce un valore di verità per tutti i simboli proposizionali.

Esempio 8.2.1. $\{P_{1,1} \text{ vero}, P_{1,2} \text{ falso}, W_{w,3} \text{ vero}\}$

8.2.1 Semantica compositonale

Il significato di una frase è determinato dal significato dei suoi componenti, a partire dalle frasi atomiche (i simboli proposizionali).

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

La logica proposizionale non richiede alcuna relazione di causalità. Per esempio per $P \Rightarrow Q$ in effetti significa che "Se P è vero, allora sostengo che lo è anche Q . Se P è falso, allora non faccio alcuna asserzione"

Una formula α è conseguenza logica di un insieme di formule KB se e solo se in ogni modello di KB , anche α è vera ($KB \models \alpha$).

Indicando con $M(KB)$ i modelli dell'insieme di formule in KB e con $M(\alpha)$ l'insieme delle interpretazioni che rendono α vera (i modelli di α)

$$KB \models \alpha \text{ sse } M(KB) \subseteq M(\alpha)$$

8.2.2 Model checking

Un modo (semplice) per determinare la conseguenza logica. Si enumerano i modelli. Si mostra che la formula α deve valere in tutti i modelli in cui è vera la KB. Si usa per eseguire “inferenze logiche”, derivare una conclusione che segue logicamente da una KB

Esempio 8.2.2. La KB iniziale KB_0 è costituita dalle regole generali del WW e deal fatto che la casella iniziale è safe per definizione.

$\neg W_{1,1} \ \neg P_{1,1}$ la casella iniziale è safe

Esempio di regole generate:

$$B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) \quad B_{1,1} \Leftrightarrow (P_{1,2} \wedge P_{2,1})$$

L'agente non ha percepito nulla in [1,1] e si è spostato in [2,1] dove ha percepito una Brezza. KB_1 a questo punto è KB_0 più i fatti corrispondenti alle percezioni

$$KB_0 \cup \{\neg B_{1,1}, B_{2,1}, \neg F_{1,1}, \neg F_{2,1}, \dots\}$$

Domandiamoci se le stanze adiacenti non contengono pozzi

$$KB_1 \models \neg P_{1,2} \quad KB_1 \models \neg P_{2,2} \quad KB_1 \models \neg P_{3,1}$$

Cosa sappiamo all'inizio (quindi cosa c'è in KB_0):

- In $[1,1]$ non ci sono pozzi: $R1 : \neg P_{1,1}$
 - In una stanza si percepisce brezza se e solo se c'è un pozzo in una stanza adiacente (di seguito solo per le posizioni rilevanti):

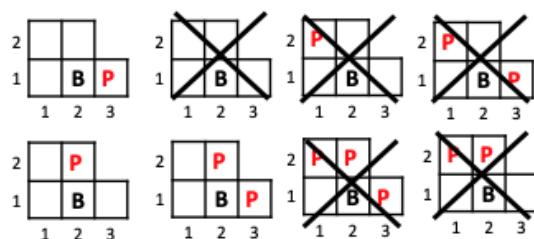
$$R2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \quad R3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

Cosa sappiamo in base alle percezioni (cosa aggiungiamo alla KB):

- Non c'è brezza in [1,1]: $R4 : \neg B_{1,1}$
 - C'è brezza in [2,1]: $R5 : B_{2,1}$

B _{1,1}	B _{2,1}	P _{1,1}	P _{1,2}	P _{2,1}	P _{2,2}	P _{3,1}	R ₁	R ₂	R ₃	R ₄	R ₅	KB
false	true	true	true	false	false	false						
false	false	false	false	false	false	true	true	false	false	false	false	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	true	false	false	false	false	false	true	true	false	true	true	false
<hr/>												
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	false	false	true	true	true	true	true	true	true
<hr/>												
false	true	false	false	true	false	false	true	false	false	true	true	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	false	true	true	false	true	false						

Considerando solo l'esistenza di pozzi nelle 3 caselle $P_{1,2}, P_{2,2}, P_{3,1}$ ci sono $2^3 = 8$ possibili interpretazioni o mondi possibili.



$KB_1 \models \neg P_{1,2}$ $KB_1 \neg P_{2,2} \vee P_{3,1}$ possiamo concludere che $\neg P_{2,2} \wedge \neg P_{3,1}$ (ne su $P_{2,2}$ e $P_{3,1}$...)

9 Calcolo proposizionale

9.1 Dimostrazione di teoremi

Fin qui abbiamo visto come determinare la conseguenza logica tramite il model checking, enumerare i modelli e mostrare che la formula deve valere in tutti. La conseguenza logica puo' essere ottenuta anche tramite la dimostrazione di teoremi:

- applicando regole di inferenza direttamente alle formule della KB
- costruire una dimostrazione della formula desiderata senza consultare i modelli

Per quest'ultimo caso abbiamo tre fondamentali concetti relativi alla conseguenza logica: equivalenza logica, validità, soddisfabilità

9.1.1 Equivalenza logica

Due formule α e β sono **equivalenti** se sono vere nello stesso insieme di modelli.

$$\alpha \equiv \beta \Leftrightarrow \alpha \models \beta \text{ e } \beta \models \alpha$$

Esempio 9.1.1. $A \wedge B \equiv B \wedge A$ $\neg(A \wedge B) \equiv \neg A \vee \neg B$ $(A \wedge (A \vee B)) \equiv A$

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutatività di \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutatività di \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associatività di \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associatività di \vee
$\neg(\neg \alpha) \equiv \alpha$	eliminazione della doppia negazione
$(\alpha \Rightarrow \beta) \equiv (\neg \beta \Rightarrow \neg \alpha)$	contrapposizione
$(\alpha \Rightarrow \beta) \equiv (\neg \alpha \vee \beta)$	eliminazione dell'implicazione
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	eliminazione del bicondizionale
$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributività di \wedge su \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributività di \vee su \wedge

9.1.2 Validità

Una formula α **valida** sse è vera in tutte le interpretazioni.

Esempio 9.1.2. $A \vee \neg A$

Le formule valide sono anche dette **tautologie**, sono formule necessariamente vere.

Teorema 9.1.1 (Teorema di deduzione). Date due formule α e β

$$\alpha \models \beta \Leftrightarrow (\alpha \Rightarrow \beta) \text{ è valida}$$

9.1.3 Soddisfabilità

Una formula α è **soddisfacibile** sse esiste una interpretazione in cui α è vera (ovvero se esiste un modello di α). **SAT**: determinare la soddisfabilità di formule della logica proposizionale.

Validità e soddisfabilità sono strettamente connesse:

- α è valida sse $\neg \alpha$ è insoddisfacibile
- α è soddisfacibile sse $\neg \alpha$ non è valida

9.1.4 Dimostrazione per assurdo

Teorema di deduzione: $\alpha \models \beta$ se e solo se $(\alpha \Rightarrow \beta)$ è **valida**.

$\alpha \models \beta$ se e solo se $\neg(\alpha \Rightarrow \beta)$ è **insoddisfacibile**. Poi applico delle equivalenze logiche:

$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \text{ [eliminazione dell'implicitazione]}$$

$$\neg(\alpha \Rightarrow \beta) \equiv \neg(\neg\alpha \vee \beta) \equiv (\alpha \wedge \neg\beta) \text{ [De Morgan]}$$

Da questo consegue che:

$$\alpha \models \beta \Leftrightarrow (\alpha \wedge \neg\beta) \text{ è insoddisfacibile}$$

Dimostrazione per refutazione o per contraddizione, si assume che β sia falsa e si dimostra che questo porta a una contraddizione con gli assiomi α

9.1.5 Inferenza per PROP

- **Model checking:** una forma di inferenza che fa riferimento alla definizione di conseguenza logica (si enumerano i possibili modelli), Tecnica delle tabelle di verità (TV)
- Algoritmi per la **soddisfabilità (SAT)**: $KB \models \alpha \Leftrightarrow (KB \vee \neg\alpha)$ è insoddisfacibile (**Teorema di refutazione**)
 - dimostrare α partendo da KB verificando che $(KB \wedge \neg\alpha)$ non è mai vero (dimostrazione per assurdo)
 - si parte da $\neg\alpha$ e si dimostra che questo porta a una contraddizione con gli assiomi in KB (che sono noti).

La conseguenza logica può essere ricondotta a un problema SAT

10 Introduzione Machine Learning

Questo per molti è un nuovo campo, con una metodologia diversa di approccio. Infatti qui gli algoritmi non vengono usati per risolvere un problema ma invece per creare modelli dai dati.

Con "Learning" (apprendimento) si intende i principi universali per gli esseri viventi, la società e le macchine:

Il problema dell'apprendimento è senza dubbio al centro stesso del problema intelligenza, sia biologica che artificiale (Poggio, Shelton, AI Magazine 1999)

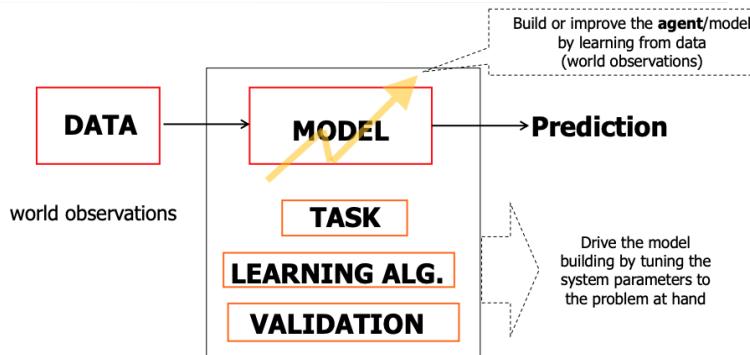
Il "Learning" è quindi un obiettivo complesso, ed un campo di ricerca in continua crescita. Il **machine learning** è emersa come un'area di ricerca che combina l'obiettivo di creare computer in grado di apprendere (IA) e nuovi potenti strumenti adattivi/statistici con basi rigorose scienza computazionali. Macchine che imparano da sole. Perché? Lusso o necessità?

- Crescente disponibilità e necessità di analisi di dati empirici.
- Difficile fornire adattività/intelligenza mediante la programmazione.

Gli obiettivi principali del ML includono:

- Come metodologia AI, costruisci sistemi intelligenti adattivi (Dal motore di ricerca alla robotica).
- Come apprendimento statistico, costruisci un potente sistema predittivo per l'analisi intelligente dei dati (strumenti per il "data scientist").
- Come metodo informatico per ambiti applicativi innovativi, usare modelli come tool per problemi complessi, interdisciplinari (dalla analisi di dati biologici per capire immagini).

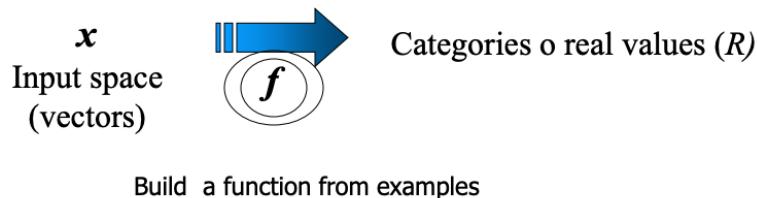
Noi in particolare andiamo a studiare il ML perché è un'opportunità per conoscere **nuovi paradigmi informatici**, con un approccio diverso rispetto a alla programmazione standard, IA algoritmica/clas-sica approcci. Tipico dell'area del soft computing/intelligenza computazionale. Per trovare soluzioni approssimative per problemi difficili, difficile da formalizzare per algoritmo "fatto a mano". Costruire nuovi **sistemi intelligenti** robusti e ampiamente applicabili.



Esempio 10.0.1. Un esempio potrebbe essere riconoscere dei numeri scritti a mano.

- **Input:** abbiamo un insieme di immagini di numeri scritti a mano.
- **Problem:** costruire un modello che riceve come input un'immagine di un numero scritto a mano e capisce che numero è.

La difficoltà qui sta nel **formalizzare** esattamente la soluzione del problema: infatti ci potrebbe essere la presenza di **"rumore"** e **ambiguità** dei dati.



10.1 Apprendimento supervisionato

Questo problema rientra nell'insieme del **Supervised learning** (classificazione, regressione). Nel apprendimento supervisionato abbiamo che:

- **Given:** vengono dati degli esempi di allenamento $<input, output> = (x, d)$, per una funzione sconosciuta f . Il **target value** è il valore desiderato d o t o y ... viene fornito dall'insegnamento secondo $f(x)$ per etichettare i dati.
- **Find:** viene trovato una buona approssimazione di f .

Come abbiamo già detto prima l'apprendimento supervisionato di divide in **classificazione** e **regressione**.

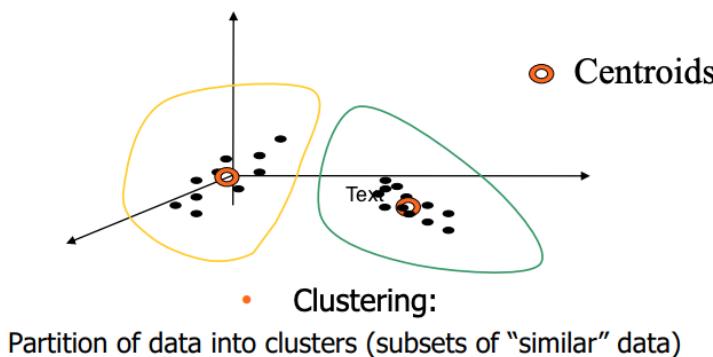
Definizione 10.1.1 (Classificazione). $f(x)$ ritorna (la presunta) classe di correttezza per x , $f(x)$ è funzione a valori discreti $\in \{1, 2, \dots, k\}$.

Definizione 10.1.2 (Regressione). Valori di uscita conitnui reali (approssimare una funzione target a valore reale, in R, R^*)

10.2 Apprendimento non supervisionato

Definizione 10.2.1 (Training Set (TS)). Insieme di dati senza etichetta.

Per esempio. Per trovare raggruppamenti naturali in un insieme di dati (clustering, Riduzione dimensionale/Visualizzazione/Preelaborazione, modellazione della densità di dati.).



10.3 Model

Un **modello** ha l'obiettivo di catturare/descrivere le relazioni tra i dati (sul base del compito). Va anche a definire la classe di funzioni che la macchina che apprende può eseguire implementare (spazio delle ipotesi)

Esempio 10.3.1. Un insieme di funzioni $h(x, w)$ dove w sono dei parametri (astratti).

Altre definizioni importanti da dire sono le seguenti:

- **Training example:** un esempio delle forma $(x, f(x) + noise)$ con x che solitamente è un vettore di caratteristiche, (d o t) $y = f(x) + noise$ è chiamato il valore target.

- **Target function:** la vera funzione f .
- **Ipotesi:** Una funzione proposta h ritenuta simile a f . Un'espressione in un dato linguaggio che descrive le relazioni tra i dati.
- **Spazio di ipotesi:** Lo spazio di tutte le ipotesi (modelli specifici) che può, in linea di principio, essere prodotto dall'algoritmo di apprendimento.

Fortunatamente già conoscete alcuni linguaggi in cui esprimere relazioni che possiamo usare per esprimere modelli di ML (le ipotesi h):

- **Logica** (proposizionale)
- **Equazioni numeriche**
- **Probabilità:** questo verrà reintrodotto per la rappresentazione della conoscenza incerta in AI (e quindi per esprimere modelli di ML)

Una prima vista dei differenti modi in cui verranno rappresentate le ipotesi.

- **Liner models**
- **Symbolic rules:** (lo spazio delle ipotesi è basato sulla rappresentazione discreta); sono possibili regole diverse, ad es:

```
if(x1 = 0) and (x2 = 1) then h(x) = 1
else h(x) = 0
```

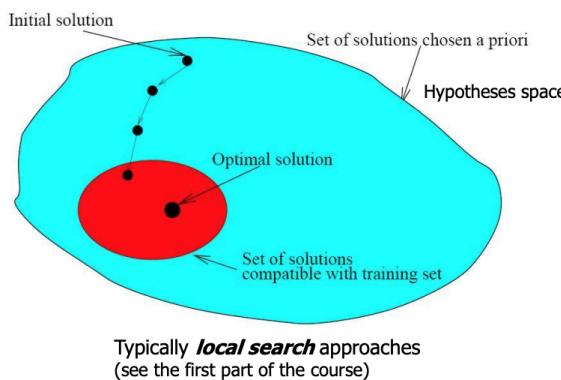
- **Probabilistic models:** una stima $p(x, y)$

10.4 Learning algorithms

Basandosi su dati, attività e modello, apprendimento come ricerca (euristica) attraverso lo spazio delle ipotesi H di l'ipotesi migliore. Cioè la migliore approssimazione alla funzione target (sconosciuta). Tipicamente ricercando la h con il minimo “errore”.

Esempio 10.4.1. Per esempio i parametri liberi del modello sono adattati al compito da svolgere, per esempio migliore w nei modelli lineari, migliori regole per i modelli simbolici, ecc.

H potrebbe non coincidere con l'insieme di tutte le possibili funzioni e la ricerca non può essere esaustiva: occorre fare delle ipotesi, noi vedremo il ruolo del bias induttivo.



Apprendimento: cercare una **buona funzione** in uno spazio di dati noti (in genere riducendo al minimo un errore/perdita).

Definizione 10.4.1. Buon w.r.t. errore di generalizzazione: misura come il modello prevede con precisione su nuovi campioni di dati (Errore/perdita misurata su nuovi dati) (errore basso, alta precisione e viceversa)

La generalizzazione è un punto cruciale nel ML. Essa comprende:

- **Fase di apprendimento**(formazione, adattamento): costruire il modello dai dati conosciutidati di addestramento (e bias)
- **Fase predittiva** (test): applicare a nuovi esempi (prendiamo gli input x' ; calcoliamo la risposta secondo il modello; confrontiamo con il suo bersaglio che il modello non ha mai visto). Valutazione dell'ipotesi predittiva, cioè del **capacità di generalizzazione**.
- **Teoria:** Per esempio. Teoria dell'apprendimento statistico. In quali condizioni (matematiche) un modello è in grado di farlo generalizzare?

Note 10.4.1. Le performance in ML sono uguali all'accuratezza nelle previsioni, stimato dall'errore calcolato sul set di test (Hold out).

11 Concept learning

Il Concept Learning per esempio è la ricerca in spazi impotetici. Lo guardiamo perché è un'opportunità per dare uno sguardo ad un semplice apprendimento simbolico e come base per la successiva introduzione di modelli flessibili.

Un altro caso sono gli spazi discreti (rappresentazione simbolica del bersaglio funzione), sono struttura dello spazio delle ipotesi.

Il problema dell'apprendimento prevede di imparare come migliorare con l'esperienza in qualche compito. Migliorare oltre una task T, tenendo conto di una misura di performance P e basandosi sull'esperienza E.

Definizione 11.0.1. *Il concept learning si definisce come inferire una funzione booleana da esempi di training positivi e negativi.*

$$c : X \rightarrow \{t, f\} \text{ or } \{+, -\} \text{ or } \{\text{yes}, \text{no}\} \text{ or } \{0, 1\}$$

con X che è lo spazio di istanze.

Un esempio di concepts potrebbero essere "gatto" in animale o "sport-divertente" in giorni.



Definizione 11.0.2. Si definisce "esempio": $\langle x, c(x) \rangle$ ($x < x, d >$) in D (o nell'insieme TR).

Definizione 11.0.3. Si dice che $h : X \rightarrow \{0, 1\}$ soddisfa x se $h(x) = 1$

Definizione 11.0.4. Una ipotesi h è consistente

- Con un esempio $\langle x, c(x) \rangle$, con $x \in X$ se $h(x) = c(x)$
- Con D , se $h(x) = c(x)$ per ogni esempio di training $\langle x, c(x) \rangle \in D$

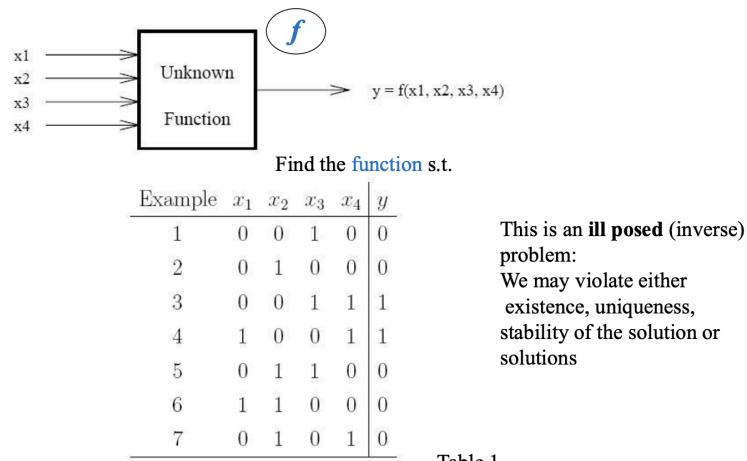


Table 1

Il problema dell'**ill posed** dice che: ci sono $2^{16} = 2^{2^4} = 65536$ possibili funzioni booleane oltre quattro funzioni booleane di input. Noi non possiamo sapere quale sia corretta finché non abbiamo visto ogni possibile coppia input-output. Dopo 7 esempi, noi continuiamo ad avere 2^9 possibilità.

Definizione 11.0.5. Nel caso generale $|H| = 2^{\#-instances} = 2^{2^n}$ per inputs/outputs binari, e con n = dimensione input.

Per lavorare con spazi di ipotesi ristretti: dobbiamo iniziare scegliendo uno spazio di ipotesi H che è considerabile più piccolo rispetto allo spazio di tutte le possibili soluzioni.

11.1 Regole congiuntive

Quando ci chiediamo quante possibili h abbiamo, è come chiedersi quante regole congiuntive semplici nell'esempio della tabella dei valori binari.

Nel caso generale:

- Letterali positivi, per esempio: $h_1 = l_2 \ h_2 = (l_1 \wedge l_2) \ h_3 = \text{true}, \dots$
- **Regole congiuntive semplici** $|H| = 2^n$ (immaginando l_i come un bit di una stringa di lunghezza n)
- Letterali (anche $\text{not}(l_i)$) $|H| = 3^n + 1$

Esempio 11.1.1. Esempio di training per il concept.

- **Concept:** giorni il cui l'amico Aldo si diverte con il suo sport preferito.
- **Task:** prevedere il valore di "Enjoy Sport" per un giorno arbitrario in base ai valori degli altri attributi attribuiti.

Target						
Sky	Temp	Humid	Wind	Water	Fore- cast	Enjoy Sport
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Warm	High	Strong	Warm	Same	Yes
Rainy	Cold	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

example

11.1.1 Rappresentazione di ipotesi

Un ipotesi h è una congiunzione di vincoli sugli attributi. Ogni vincolo può contenere:

- Uno specifico valore. Ess. Water=Warm
- Un valore che non mi interessa: Ess. Water=?
- Un valore non consentito (ipotesi null). Ess. Water=∅

Esempio 11.1.2. Un esempio di ipotesi h :

Sky	Temp	Humid	Wind	Water	Forecast
Sunny	?	?	Strong	?	Same

Note 11.1.1. Questa è una funzione h , non un esempio di input.

Questa roba corrisponde alla funzione booleana:

$$Sky = \text{Sunny} \wedge \text{Wind} = \text{Strong} \wedge \text{Forecast} = \text{Same}$$

$$\begin{array}{ccccccc} < & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset > \text{Most specific} \\ < & ? & ? & ? & ? & ? & ? > \text{Most general} \end{array}$$

Definizione 11.1.1. Task di apprendimento di concetti prototipici:

- **Given**

- **Istanza X:** Per esempio i possibili giorni descritti da degli attributi (Sky , $Temp$, $Humidity$, $Wind$, $Water$, $Forecast$)
- **Target function c:** $EnjoySport : X \rightarrow \{0, 1\}$
- **Ipotesi H:** una congiunzione di (insieme finito di) letterali ess. $\langle \text{Sunny} \ ? \ ? \ \text{Strong} \ ? \ \text{Same} \rangle$

- **Training examples** D : esempi positivi e negativi di una funzione target: $\langle x_1, c(x_1), \dots, \langle x_n, c(x_l) \rangle \rangle$ (l esempi)

- **Trovare:** una ipotesi $h \in H$ tale che $h(x) = c(x)$ per ogni $x \in X$

Chiamiamo l'operazione di **Learning** la ricerca di ipotesi nello spazio H .

Ipotesi dell'apprendimento induttivo: Assumiamo in queste lezioni che "qualsiasi ipotesi trovata per approssimare la funzione target rispetto agli esempi, andrà inoltre ad approssimare la funzione target ben oltre gli esempi osservati".

$$h = (x) \text{ per ogni } x \in D \text{ (ess. consistenza con } D) \quad h(x) = c(x) \text{ per ogni } x \in X^5$$

La rappresentazione per H determina lo spazio di ricerca:

- instance distinte: $3 * 2 * 2 * 2 * 2 = 96$
- Concetti distinti: $2^{96} = 2^{\#\text{-instanze}}$
- Ipotesi distinte sintatticamente (congiunzioni): $5*4*4*4*4*4 = 5120$ con 5 = sunny/cloudy/rainy/?/ \emptyset e 4=warm/cold/?/ \emptyset .
- Ipotesi distinte semanticamente: $1 + 4*3*3*3*3*3 = 973$ (perché tutte le h con \emptyset sono equivalenti a "false")

In generale: Dimensione elevata, addirittura infinita: strutturare lo spazio di ricerca può aiutare nella ricerca in modo più efficiente!

11.1.2 Ordinamento da generale a specifico

Consideriamo due ipotesi:

$$h_1 = \langle \text{Sunny}, ?, ?, \text{Strong}, ?, ? \rangle \quad h_2 = \langle \text{Sunny}, ?, ?, ?, ?, ? \rangle$$

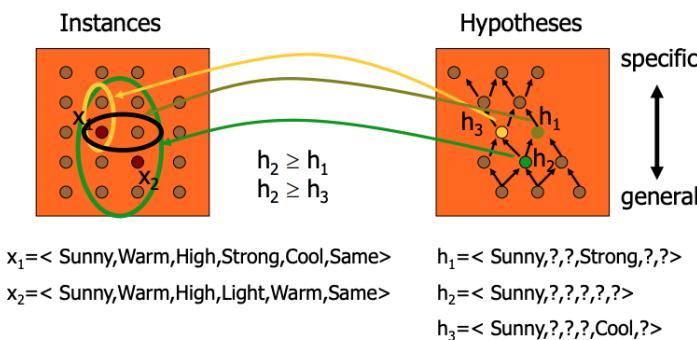
Ed un insieme di instance coperte da h_1 e h_2 .

h_2 impone meno vincoli rispetto a h_1 e quindi classifica più istanze x come positivo.

Definizione 11.1.2. Prendiamo h_j e h_k funzioni a valori booleani definite su X . Diciamo allora che h_j è **più generale di o uguale a** h_k (scritto come $h_j \geq h_k$) se e solo se

$$\forall x \in X : [(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

Esempio 11.1.3. Esempi binari $l_i : l_1 \geq (l_1 \wedge l_2)$, l_1 versus l_2 , non comparabili. La relazione \geq impone un ordine di partizione (p.o.) sullo spazio di ipotesi H che viene utilizzato da molti metodi di concept learning. Possiamo approfittare di questo p.o. organizzare in modo efficiente la ricerca in H



⁵Un problema fondamentale del ML, analisi teorica ed empirica

11.2 Algoritmo Find-S

Sfrutta l'ordine parziale m.g.t per cercare in modo efficiente h (senza fare un enumerazione esplicita per ogni $h \in H$).

Definizione 11.2.1. Definiamo l'Algoritmo con i seguenti passaggi:

1. Inizializzo h con l'ipotesi più specifica in H .

2. Per ogni istanza di training x positiva

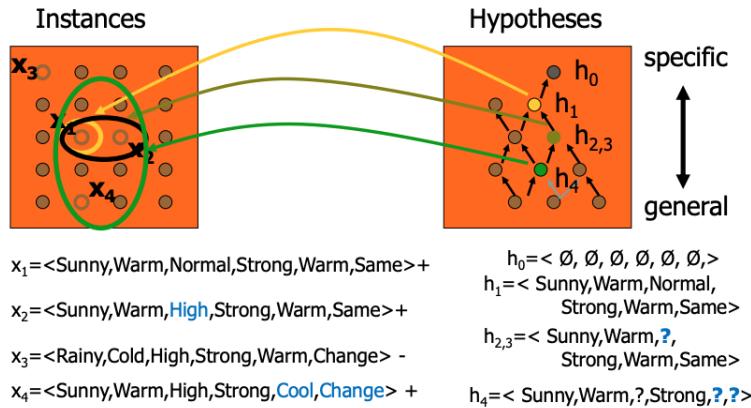
```

for each attribute a_i in h
    if a_i in h e soddisfatto da x
        non fare nulla
    else
        sostituisci a_i in h dal successivo vincolo più generale, cioè
        soddisfatto da x [es. rimuovi da h i letterali non soddisfacenti x]

```

3. In output ritorna un ipotesi h .

Ricerca su spazio di ipotesi fatto dall'algoritmo Find-S:



Alcune proprietà importanti dell'algoritmo find-S sono:

- Lo spazio di ipotesi deve essere descritto da congiunzioni di attributi (forte limitazione)
- Find-S produrrà l'ipotesi più specifica fra quelle di H che sia anche coerente con gli esempi di training **positivi**.
- L'ipotesi di output h sarà coerente con gli esempi negativi, a condizione che il target concept sia contenuto in H . (perché $c \geq h$)

Mentre invece alcuni dei problemi relativi al Find-S sono i seguenti:

- Non si può dire se l'apprendimento è convergente verso un concept target, nel senso che non è in grado di determinare se ha trovato l'unica ipotesi coerente con gli esempi di training.
- Non è possibile stabilire quando i dati di training sono inconsistenti, poiché ignora gli esempi di training negativo: nessuna tolleranza al rumore!

Perché preferire l'ipotesi più specifica? Cosa succede se ci sono più ipotesi massimamente specifiche?

11.3 Algoritmo List-Then eliminate

Per parlare di questo algoritmo bisogna prima parlare del **version spaces**. L'idea chiave è che: l'output è un'descrizione dell'insieme di tutte le h consistenti con D . Possiamo fare questo senza enumerare tutti loro:

$$\text{Consistent}(h, D) := \forall \langle x, c(x) \rangle \in D \ h(x) = c(x)$$

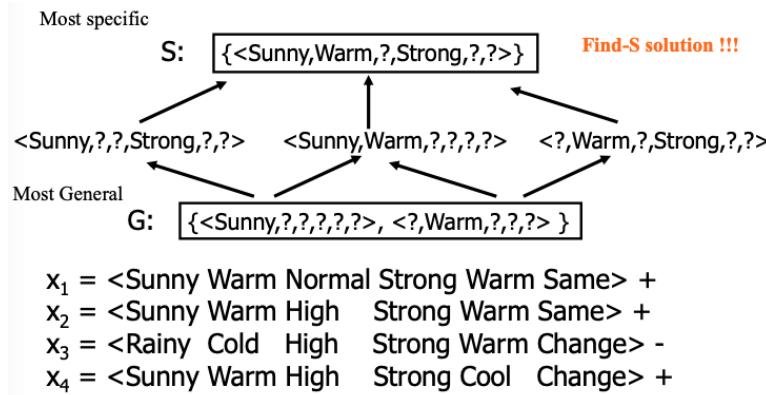
Definizione 11.3.1. Il **version space**, $VS_{H,D}$ che rispetta lo spazio di ipotesi H , e l'insieme di training D , è un sottoinsieme delle ipotesi di H consistenti con tutti gli esempi di training:

$$VS_{H,D} = \{h \in H \mid Consistent(h, D)\}$$

L'algoritmo ora si definisce con i seguenti punti:

1. Dato un VersionSpace \rightarrow una lista contenente ogni ipotesi in H .
2. Per ogni esempio di training $\langle x, c(x) \rangle$ si rimuove dal VersionSpace ogni ipotesi che è inconsistente con l'esempio di training, $h(x) \neq c(x)$
3. L'output è una lista di ipotesi in VersionSpace.

Una miglior rappresentazione per il Version space (by G e S):



11.4 Algoritmo candidate elimination

Definizione 11.4.1. Ci sono vari modi per rappresentare il version space:

- Il **confine generale**, G , dello spazio delle versioni $VS_{H,D}$ è l'insieme di membri generali massimi. (per H consistente con D).
- Il **confine specifico**, S , dello spazio delle versioni $VS_{H,D}$ è l'insieme di membri specifici massimi (per H consistente con D)
- **Teorema:** ogni membro della spazio di versione si trova fra questi confini.

$$VS_{H,D} = \{h \in H \mid (\exists s \in S) (\exists g \in G) (g \geq h \geq s)\}$$

dove $x \geq y$ vuol dire che x è più o uguale generale rispetto a y . In questo algoritmo abbiamo che:

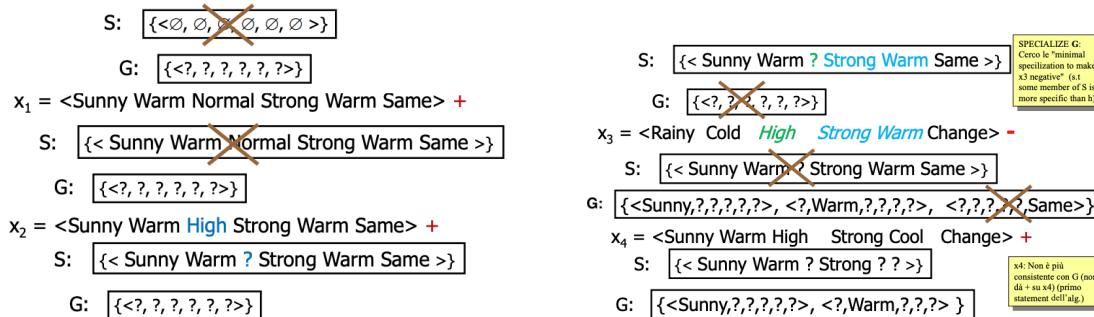
- $G \rightarrow$ ipotesi massimali generali in H
- $S \rightarrow$ ipotesi massimali specifiche in H

For each esempio di training $d = \langle x, c(x) \rangle$.

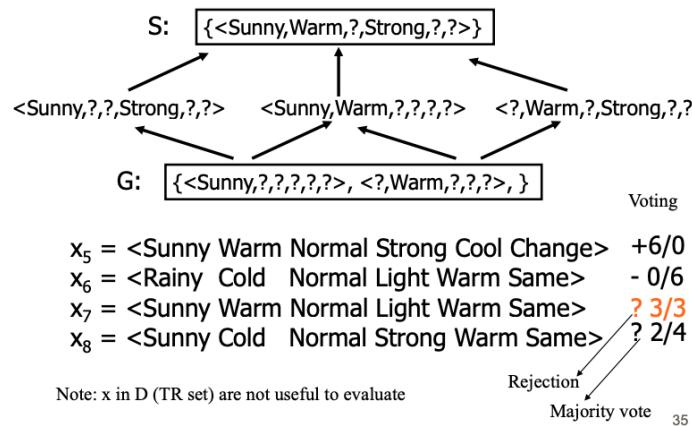
- if d è un esempio **positivo** allora rimuovi da G qualsiasi ipotesi che sia incoerente con d (def di VS).
for each ipotesi $s \in S$ che non è consistente con d [generalizzare S]
 - Rimuovere s da S .
 - Aggiungere ad S tutti le generalizzazioni minime h di s tale che: h consistente con d ed un qualche membro di G è più generale rispetto ad h .
 - Rimuovere da S ogni ipotesi che è più generale rispetto ad altre ipotesi in S .

- if d è un esempio **negativo** allora rimuovere da S ogni ipotesi che è inconsistente con d.
for each ipotesi $g \in G$ che non è consistente con d [specilizzazione G]
 - rimuovere g da G.
 - Aggiungere a G tutte le specializzazioni minime h di g tali che: h sia consistente con d e cia sia qualche membro di S che è più specifico rispetto a h.
 - Rimuovere da G ogni ipotesi che è meno generale rispetto ad altre ipotesi in G.

Esempio 11.4.1. Esempio dell'algoritmo Candidate Elimination.



La classificazione di nuovi dati si può fare nel seguente modo.



11.5 Bias induttivo ed il suo ruolo

Il nostro spazio di ipotesi non è capace di rappresentare un semplice concetto target di disgiunzione come il seguenti: $(\text{Sky} = \text{Sunny}) \vee (\text{Sky} = \text{Cloudy})$. La candidate elimination sarebbe:

$$x_1 = \langle \text{Sunny, Warm, Normal, Strong, } \langle \text{Cool, Change} \rangle \rangle +$$

$$x_2 = \langle \text{Cloudy, Warm, Normal, Strong, } \langle \text{Cool, Change} \rangle \rangle +$$

$$S : \{\langle ?, \text{Warm, Normal, Strong, Cool, Change} \rangle\}$$

$$x_3 = \langle \text{Rainy, Warm, Normal, Strong, Cool, Change} \rangle -$$

$$S : \{\}$$

Bias: Assumiamo che lo spazio di ipotesi H contenga il concetto target c. In altre parole che c sia descritto da con congiunzione (operatore AND) o da un letterale.

11.5.1 Unbiased learning

L'idea dietro al unbiased learning è la seguente: scegliere H che esprime ogni concetto che può essere insegnato, ciò significa che H è l'insieme di tutti i possibili sottoinsiemi di X : il power set $P(X)$.

Esempio 11.5.1. $|H| = 96, |P(X)| = 2^{96} \sim 10^{28}$ concetti distinti.

H = disgiunzione, congiunzione, negazione. Ess. $< Sunny, Warm, Normal, ?, ?, ? > \vee < ?, ?, ?, ?, ?, Change >$. H contiene sicuramente il concetto target. A questo punto capiamo come generalizzare.

Che cosa sono S e G in questo caso?

Assumiamo esempi positivi (x_1, x_2, x_3) ed esempi negativi (x_4, x_5)

$$S : \{(x_1 \vee x_2 \vee x_3)\} \quad G : \{\neg(x_4 \vee x_5)\}$$

Gli unici esempi che sono classificati in modo non ambiguo sono gli esempi di training stessi (H può rappresentarli). In altre parole per apprendere il concetto target si dovrebbe presentare ogni singola istanza in X come esempio di training.

Definizione 11.5.1. *Un'importante proprietà è che un unbiased learner non è in grado di generalizzare.*

Dimotrazione 11.5.1. Ogni istanza non osservata verrà classificata positiva per precisamente metà delle ipotesi nel VS e negativa nell'altra metà (rejecion). Infatti: $\forall h$ consistenti con x_i (test), $\exists h'$ identico ad h eccetto per il fatto che $h'(x_i) \neq h(x_i)$, $h \in VS \Rightarrow h' \in VS$ (ci sono identici in D , l'insieme TR).

Un entità che apprendet e che non fa ipotesi preliminari riguardo all'identità del concetto target non ha basi razionali per classificare ogni istanza invisibile. (Restrizione, preferenza) Bias non vengono solo assunti per l'efficienza, essi servono anche per la generalizzazione, tuttavia, non ci dice ancora (quantifica) quale sia la soluzione migliore generalizzante.

Esempio 11.5.2. Esempio trivial (TR = Training set, TS = Test set)

X	$c(x)$	$H = \{x, \text{not}(x), \mathbf{0}, \mathbf{1}\}$, no restriction
TR	0 0	VS = {x, 0}
TS	1 ?	→ Can be 1 or 0 ... Unless you use all X as TR set.

Problema: caratterizzare i bias per i diversi approcci di apprendimento.

11.5.2 Bias induttivo

Consideriamo le seguenti cose:

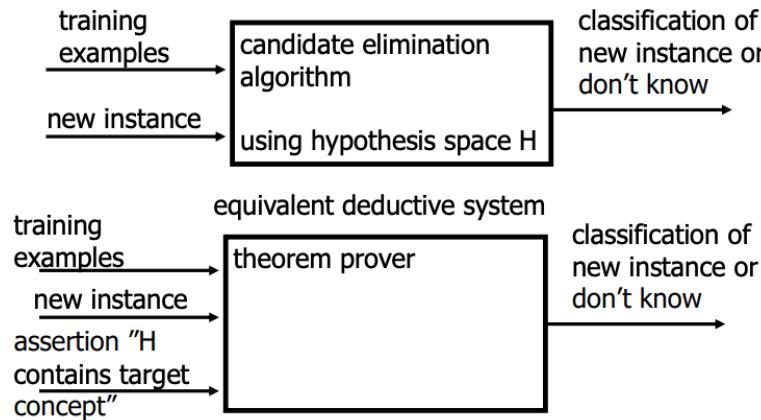
- Un algoritmo di concept learning L .
- Instance X , concetto target c .
- Esempi di training $D_c = \{< x, c(x) >\}$
- Sia $L(x_i, D_c)$ a denotare l'assegnamento di classificazione all'istanza x_i da L dopo il training con D_c .

Definizione 11.5.2. Il bias induttivo di L è ogni insieme minimo di affermazioni B tali che per ogni concetto target c e corrispettivi dati di training D_c

$$(\forall x_i \in X)[B \wedge D_c \wedge x_i] \vdash L(x_i, D_c)$$

Dove $A \vdash B$ vuol dire che A implica logicamente B (segue deduttivamente da)

Sistema induttivo e **equivalente** sistema deduttivo. Visto questo, vediamo 3 apprendimento con



il bias diversi:

- **Rote learner** (lookup table): memorizzare esempi, classificare x se e solo se esso matcha con un precedente esempio osservabile. No bias induttivo → no generalizzazione.
- **Version space candidate elimination algorithm.** Bias: lo spazio di ipotesi contiene il concetto target $|H| = 973$ versus 10^{28}
- **Find-S:** Bias: Lo spazio delle ipotesi contiene il concetto target e tutto il resto delle istanze sono istanze negative a meno che non sia implicato il contrario dalle sue altre conoscenze (viste come esempi positivi). In altre parole: abbiamo una bias linguistico dovuto all'AND sul letterali più il bias di ricerca dovuto alla preferenza della ipotesi più specifica.

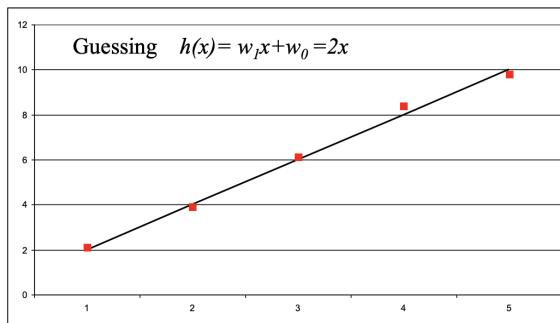
12 Modelli linari

12.1 Regression

Un possibile esempio di regressione potrebbe essere un processo di stima di una funzione di valore reale sulla base di un insieme finito di campioni rumorosi. Le conoscenze sarebbero $pairs(x, f(x) + randomnoise)$.

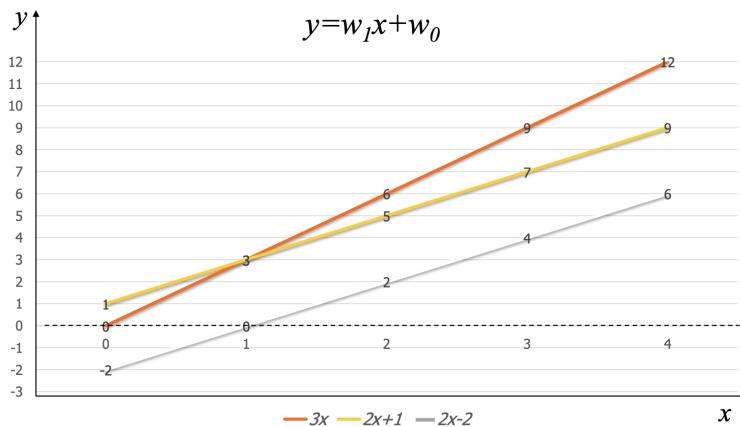
La task a questo punto sarebbe trovare f per i dati nella seguente tabella.

x	$target$
1	2.1
2	3.9
3	6.1
4	8.4
5	9.8
...	...



We want to solve it (how to find w_1 and w_0) in a «systematic» way

Esempio 12.1.1. Esempio di modelli linari:



12.1.1 Univariate linear regression

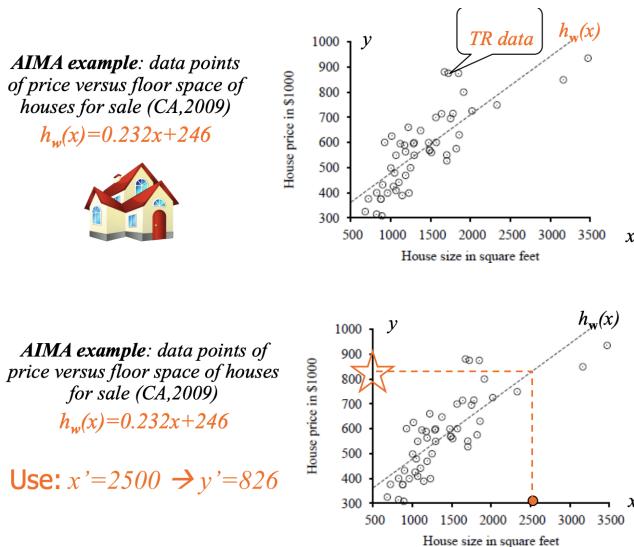
Il caso univariante, semplice regressione lineare: iniziamo con 1 valore di input, x e 1 valore di output y .

Definizione 12.1.1. Assumiamo che un modello $h_w(x)$ espresso come:

$$out = h(x) = w_1x + w_0$$

dove w è il coefficiente a valore reale/parametro libero (peso).

Si cerca di avere un adattamento dei dati secondo una “linea retta”. Trovare h (modello lineare) che meglio si adattano ai dati (da gli insieme di dati osservati dei valori x e y). Assumiamo che la variabile data (y) è (linearmente) associata ad un’altra variabile (x) o più variabili, da $y = w_1x + w_0 + noise$, dove w è il **free pair** e **noise** è un **errore misurato al target** (con una distribuzione normale). Andiamo a costruire un modello (trovare il valore w) per predire/stimare il prezzo (y) dei punti per altri (osservabile) valori x (prediction). Usiamo poi questi valori per **predire/stimare** il prezzo (t) dei punti per altri valori osservabili.



12.1.2 Learning via LMS

Abbiamo capito che quindi dobbiamo andare a trovare i valori dei parametri w (w_1 e w_2 nei casi univarianti) per andare a minimizzare l'output dell'errore del modello (per eseguire un buon fitting). In uno spazio di ipotesi infinito (valore w continuo) abbiamo una buona soluzione data dalla matematica classica, possiamo "imparare" da dei semplici tools, sebbene semplice incluse moltri concetti rilevanti per il ML moderno ed è alla base dei metodi evoluti in the field. Definiamo qui una **funzione loss / error** e usiamo il **Least Mean Square (LMS)** approccio.

Il **training** si fa trovando w tale che minimizzi l'**errore/la perdita empirica** (il miglior dato che fitta nel training set con l'esempio l)

Definizione 12.1.2. Andiamo allora a definire LMS con:

- **Given** un insieme di esempi di training l (x_p, y_p) $p = 1 \dots l$
- **Trovare** $h_w(x)$ nella forma $w_1x + w_0$ (quindi i valori di w) che minimizzano la perdita attesa dei dati di training.

Per la perdita usiamo la radice dell'errore.

Da qui il **least (mean) square** si tratta di trovare la w che minimizzi la somma residua delle radici [$\text{argmin}_w \text{Error}(w) \in L_2$]

$$\text{Loss}(h_w) = E(w) = \sum_{p=1}^l (y_p - h_w(x_p))^2 = \sum_{p=1}^l (y_p - (w_1x_p + w_0))^2$$

Dove x_p è p -th input/pattern/example, y_p l'output per p , w free par., l numero degli esempi.

Perché usare LMS per adattarsi ai dati con h ? Il least (mean) square trova w per minimizzare la **somma di radici residua** [$\text{argmin}_w \text{Error}(w) \in L_2$]:

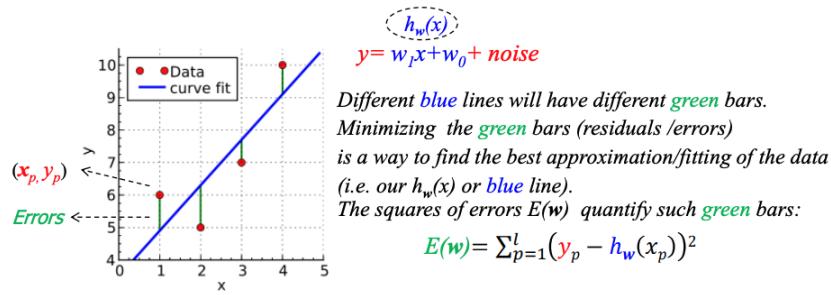
Il metodo dei minimi quadrati è un approccio standard alla soluzione approssimata di sistemi sovradeterminati, cioè insiemi di equazioni in cui ci sono più equazioni che "sconosciuti".

Come si risolve? Ricorda che il minimo locale è un punto stazionario, il gradiente è nullo.

$$\frac{\partial E(w)}{\partial w_i} = 0 \quad i = 1, \dots, \text{dim_input} + 1$$

Per una semplice regressione lineare (2 parametri liberi) abbiamo che:

$$\frac{\partial E(w)}{\partial w_0} = 0 \quad \frac{\partial E(w)}{\partial w_1} = 0$$



Funzione di perdita convessa \Rightarrow abbiamo la seguente soluzione (senza minimo locale).

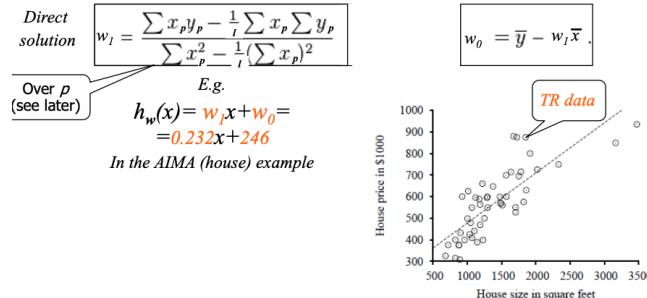
$$w_1 = \frac{\sum x_p y_p - \frac{1}{l} \sum x_p \sum y_p}{\sum x_p^2 - \frac{1}{l} (\sum x_p)^2} = \frac{Conv[x, y]}{Var[x]}, \quad w_0 = \bar{y} - w_1 \bar{x} \quad \bar{y} = \frac{1}{l} \sum t_p \quad \bar{x} = \frac{1}{l} \sum x_p$$

Calcolare il gradiente per 1 (ciascuno, omettendo quindi p per x) pattern p.

$$\frac{\partial E}{\partial w_i} = \frac{\partial(y - h_w(x))^2}{\partial w_i} = 2(y - h_w(x)) \frac{\partial(y - h_w(x))}{\partial w_i} = 2(y - h_w(x)) \frac{\partial(y - (w_1 x + w_0))}{\partial w_i}$$

Facendo una sintesi della LMS abbiamo che:

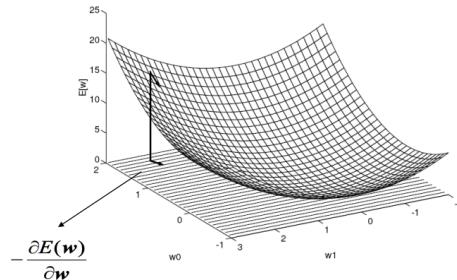
- **Given** un insieme di l esempi di training (x_p, y_p) ed una funzione di perdita.
- **Find** il valore del peso w per costruire $h_w(x)$ espresso come $y/out = w_1 x + w_0$ (minimizzare la perdita attesa dei dati di training)



Ora è possibile usarlo per nuovi x' per predire y' . Vediamo ora un approccio differente.

12.2 Gradiant descent (ricerca locale)

Errore delle superfici per il linear model con 2 pesi. Parabolico per $E(w) = E([w_0, w_1]^T)$ (funzione quadratica convessa)



Spazio ipotetico con 2 parametri w_0, w_1 . Il gradiente di $E(w)$ è il nostro "compasso" per trovare il minimo.

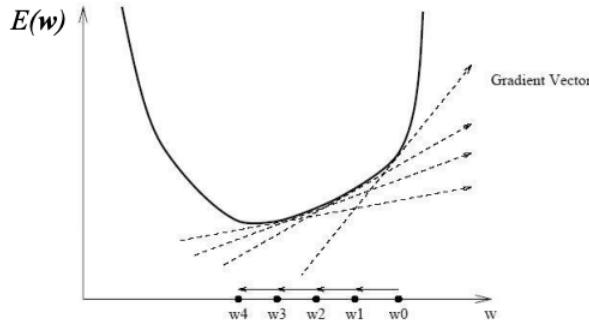
La derivazione precedente suggerisce la linea per costruire un **algoritmo iterativo** basato su $\frac{\partial E(w)}{\partial w_i}$

Definizione 12.2.1. *Gradiente = direzione salita, possiamo muoverci verso il minimo con un gradiente in discesa ($\Delta w = -\text{gradiente di } E(w)$)*

Definizione 12.2.2. *ricerca locale* inizia con vettore di pesi iniziali. Esso viene modificato iterativamente per diminuire fino a minimizzare la funzione di errore (discesa più ripida).

$$w_{new} = w + \eta \cdot \Delta w$$

Dove η è una costante (step size) chiamata **learning rate**



Le motivazioni intuitive del gradient descent sono che dalla formulazione, $w_{new} = w + \eta \cdot \Delta w$ possiamo ottenere:

$$\Delta w_0 = -\frac{\partial E(w)}{\partial w_0} = 2(y - h_w(x)) \quad \Delta w_1 = -\frac{\partial E(w)}{\partial w_1} = 2(y - h_w(x)) \cdot x$$

Definizione 12.2.3. Questa è una "error correction rule" chiamata **delta rule** che cambia la proporzione w con l'errore (target-output):

- $(\text{target } y - \text{output}) = err = 0 \rightarrow \text{nessuna correzione}$
- $\text{output} > \text{target} \rightarrow (y - h) < 0 \text{ (output troppo alto)}$
 - Se Δw_0 negativo \rightarrow si riduce w_0 e
 - if (input $x > 0$) Δw_1 negativo \rightarrow si ricude w_q else si aumenta w_1
- $\text{output} < \text{target} \rightarrow (y - h) > 0 \text{ (output è troppo lento)}$

In questo modo andiamo a migliorare l'apprendimento dai precedenti errori.

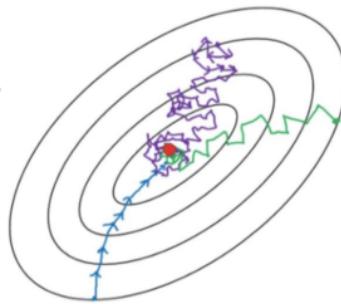
In conclusione possiamo dire che l'approccio del Gradient descent è semplice ed efficace per ricerca locale e approcci alla soluzione LMS, inoltre ci permette di cercare in uno spazio infinito di ipotesi, può essere facilmente applicato sempre per H continuo e perdite differenziabili non solo ai modelli lineari (ess. reti neurali e deep learning models). A livello di efficienza molti miglioramenti sono possibili come il Newton's method, Conjugate Gradient.

12.3 Estensioni a l dati

Definizione 12.3.1. Possiamo riassumere per l patterns (x_p, y_p) nel seguente modo.

$$\Delta w_0 = -\frac{\partial E(w)}{\partial w_0} = 2 \sum_{p=1}^l (y_p - h_w(x_p)) \quad \Delta w_1 = -\frac{\partial E(w)}{\partial w_1} = 2 \sum_{p=1}^l (y_p - h_w(x_p)) \cdot x_p$$

Dove x_p è un p -esimo input, y_p l'output per p , w una coppia libera, l numero di esempi.



Possiamo aggiornare w dopo (ripetendo) una fase di dati di training $l \rightarrow$ **algoritmo di batch** (blu). Oppure possiamo aggiornare w dopo ogni pattern $p \rightarrow$ **on-line algorithm** (discesa del gradiente stocastico) (purple and green), questo caso potrebbe essere più veloce, ma necessitare di step più piccoli.

Questo è un caso standard, usare da 2 fino a centinaia di variabili/features di input $x = [x_1, x_2, \dots, x_n]$, il **pattern** di input è un vettore.

Esempio 12.3.1. Degli esempi potrebbero essere calcolare lo stato di una scacchiera in un gioco di dama per num. di pezzi bianchi/neri/re/catturati nel turno successivo: 6 variabili. w peso che viene dalle "features sul tavolo".

Facciamo un piccolo riassunto della notazione per i dati. Prendiamo una X matrice l, x, n dove l sono le righe mentre n le colonne (feature, variabili, attributi), $p = 1, \dots, l$ e $i = 1, \dots, n$. Abbiamo spesso bisogno di omettere alcuni indici quando il contesto è chiaro, ad es:

Pattern	x_1	x_2	x_i	x_n
Pat 1	$x_{1,1}$	$x_{1,2}$		$x_{1,n}$
...				
Pat p	$x_{p,1}$	$x_{p,2}$	$x_{p,i}$	$x_{p,n}$
...				

- Ogni riga, x generica (vettore in grassetto), un raw nella tabella: (input) esempio, pattern, instance, sample, ..., input vector, ...
- x_i o x_j (scalari): componenti i o j (dato un pattern, omettendo p).
- x_p (o x_i) (vettore in grassetto) il p-esimo (o i-esimo) raw nella tabella = pattern p (o i)
- $x_{p,i}$ (scalare) anche come $(x_p)_i$: componente i nel pattern p (o usiamo $x_{p,j}$ per il componente j, ecc.)
- Per il target sono solitamente usiamo solo y_p con $p = 1, \dots, l$ (lo stesso per d o t)

Per la notazione di input multidimensionali andiamo ad assumere una colonna vettore per x e w (in grassetto). Numero del dato l , dimensione del vettore di input n , y_p (targets) e $p = 1, \dots, l$

$$w^t x + w_0 = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n = w_0 + \sum_{i=1}^n w_i x_i$$

Note 12.3.1. Notare che spesso, come prima, la notazione di trasposta T in w^T è omessa.

w_0 è l'intercetta, la soglia, il bias, l'offset.. Spesso è conveniente per includere la costante $x_0 = 1$ in modo che possiamo scriverlo come:

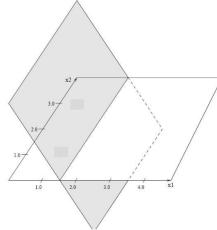
$$w^T x = x^T w (\text{inner product}) \quad x^T = [1, x_1, x_2, \dots, x_n] \quad w^t = [w_0, w_1, w_2, \dots, w_n]$$

Note 12.3.2. Notare che w è un parametro (libero) continuo "pesi".

In una vista geometrica avremmo quello che si chiama **iperpiano**. Per 2 variabili avremmo:

$$x^T w = w^T x = w_0 + w_1 x_1 + w_2 x_2 \quad (\text{n dim}) w^t = [w_0, w_1, w_2, \dots, w_n]$$

Quindi possiamo definirlo come:



$$h(x_p) = x_p^T w = \sum_{i=0}^n x_{p,i} w_i$$

Possiamo sintetizzare il caso con input di dimensione n , e l patterns nel seguente modo:

- **Given** un insieme di l esempi di training x_p, y_p
- **Find** il peso del vettore w che minimizzi la perdita attesa dei dati di training.

$$E(w) = \sum_{p=1}^l (y_p - x_p^T w)^2 = \|y - Xw\|^2$$

Dove x_p è il p -esimo input del vettore, y_p è l'output per p , w è la coppia libera, l il numero di esempi e n la dimensione dell'input.

12.4 Algoritmo iterativo di discesa del gradiente

Definizione 12.4.1. Definiamo ora l'*algoritmo iterativo di discesa del gradiente*

$$\Delta w_i = -\frac{\partial E(w)}{\partial w_i} = 2 \sum_{p=1}^l (y_p - h_w(x_p)) \cdot x_{p,i} = 2 \sum_{p=1}^l (y_p - x_p^T w) x_{p,i}$$

Abbiamo che il 2 è una costante che può essere ignorata nello sviluppo dell'algoritmo, mentre $x_{p,i}$ è il componente i del pattern p

Questo è una estensione vettoriale al gradiente molto semplice nel quale ogni x , ha il suo w_i e il suo Δw_i (come il singolo x prima). Il vettore di gradienti per una dimensione n sarebbe il seguente:

$$\Delta w = -\frac{\partial E(w)}{\partial w} = \begin{bmatrix} -\frac{\partial E(w)}{\partial w_1} \\ -\frac{\partial E(w)}{\partial w_2} \\ -\frac{\partial E(w)}{\partial w_i} \\ \vdots \\ -\frac{\partial E(w)}{\partial w_n} \end{bmatrix} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \Delta w_i \\ \vdots \\ \Delta w_n \end{bmatrix}$$

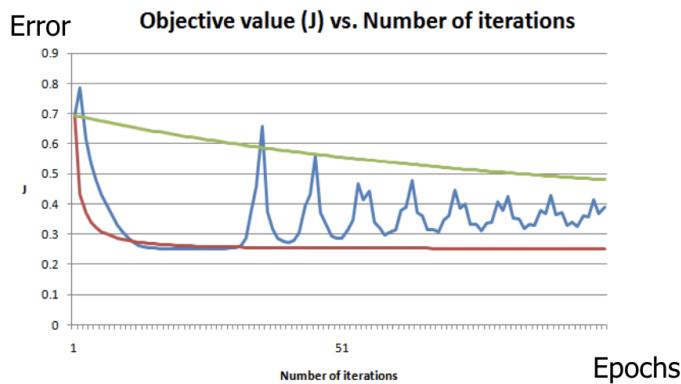
Possiamo lavorare in uno spazio a più luci senza la necessità di visualizzarlo. PS w_0 viene omesso sopra, puoi includerlo facilmente.

Per riassumere possiamo spiegare l'algoritmo in dei semplici passaggi:

1. Iniziare con un vettore dei pesi $w_{initial}$ (piccolo), fissando un η ($0 < \eta < 1$).
2. Calcolare $\Delta w = \text{"gradiente di } E(w) = -\frac{\partial E(w)}{\partial w}$ (o per ogni w_j)
3. Calcolare $w_{new} = w + \eta \cdot \Delta w$ (o per ogni w_j)
4. Ritornare al 2 e ripetere finché non si converge o $E(w)$ è "sufficientemente piccolo"

$\Delta w/l$ least mean square:

- Batch versions (Δw dopo ogni "epoch" dei dati "l")
- Stochastic/on-line version: aggiornare w per ogni pattern p (da $\Delta_p w = x_p(y_p - x_p^T w)$ senza aspettare la somma totale su l)
- $\eta = \text{learning rate}$: seed/stability trade-off: può essere (gradualmente) decrementato a zero (garantendo convergenza, evitando oscillazioni intorno al min.)

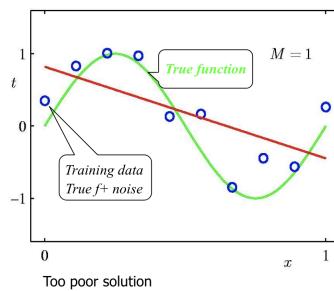


Definizione 12.4.2. Queste sono le **curve di apprendimento**: Mostrano come si verifica l'errore diminuisce attraverso il gradiente iterazioni di discesa.

I vantaggi dei modelli lineari sono che se funzionano bene sono un modello fantastici: molto semplice, tutte le informazioni dei dati sono in w , facili da interpretare, dati "rumorosi" sono accettati, Fenomeni lineari: un sogno per la scienza: ideale per realizzare un fenomeno "naturale". legge.

12.5 Limitazioni

Una delle limitazioni dei modelli lineari sono le tasks di regressione per problemi non lineari.



Proviamo ora a muoverci verso delle relazioni non-lineari. Notare che in $h_w(x) = w_1x + w_0$ o $h_w(x) = w^T \cdot x$ come modelli statistici parametrici: "lineare" non si riferisce a questa linea retta, piuttosto al modo in cui si verificano i coefficienti di regressione w nell'equazione di regressione.

Possiamo quindi utilizzare anche input trasformati, come ad esempio x, x^2, x^3, x^4, \dots con una relazione

non lineare input e output, gestiamo la macchina di learning (Least Square solution) sviluppandola nel seguente modo:

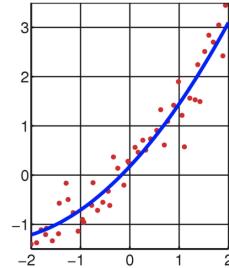
$$h_w(x) = w_0 + w_1x + w_2x^2 + \cdots + w_Mx^M = \sum_{j=0}^M w_jx^j$$

Detta **polynomial regression**.

Esempio 12.5.1. Facciamo un esempio non-lineare.

Il risultato dell'adattamento di una funzione quadratica (in blu) $M = 2$, passante un insieme di punti dati (in rosso).

Nei minimi quadrati lineari la funzione non deve essere lineare in argomento (variabili di input) ma solo nei parametri (w) che sono determinati per dare il best fit.



12.6 Linear basis expansion

Una generalizzazione (che mostriamo per la regressione) è la **LBE** o **linear basis expansion**, che definiamo come

$$h_w(x) = \sum_{k=0}^K w_k \phi_k(x)$$

L'argomento di input è un vettore con variabili aggiuntive, le quali sono trasformazioni di x secondo una funzione ϕ ($\phi_k : R^n \rightarrow R$).

Esempio 12.6.1. Alcuni esempi sono:

- Rappresentazione polinomiale $x : \phi(x) = x_j^2$ o $\phi(x) = x_j x_i \dots$
- Trasformazione non lineare di un singolo input $\phi(x) = \log(x_k), \phi(x) = \text{root}(x), \dots$
- Trasformazione non lineare di input multipli: $\phi(x) = \|x\|$
- Splines, ...

Il numero di parametri $K > n$. Il modello è lineare nei parametri (anche in ϕ , non in x): possiamo usare lo stesso algoritmo di learning di prima.

Esempio 12.6.2 (1-dim x). $\phi_j(x) = x^j$.

$$h(x) = w_0 + w_1x + w_2x^2 + \cdots + w_Mx^M = \sum_{j=0}^M w_jx^j$$

1-dim regressione polinomiale ($K = M$). Per il numero di termini nel polinomio con D dim. input vedremo più avanti.

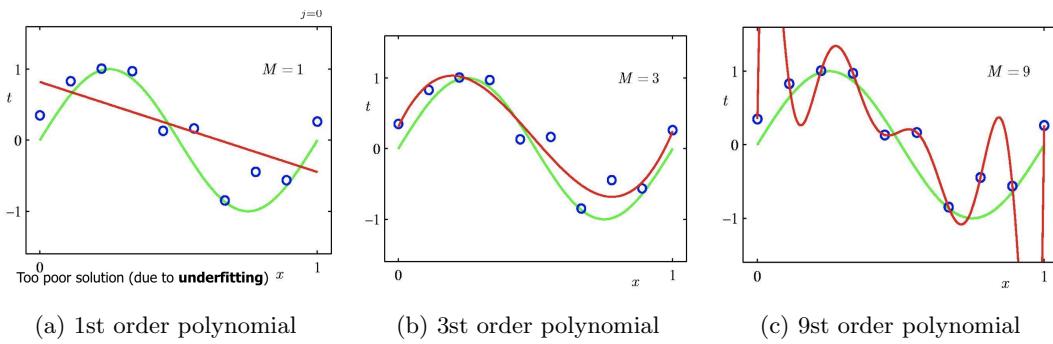
O "qualsiasi altro" $\phi(x) = \phi([x_1, x_2, x_3])$

$$h(x) = w_0 + w_1x_1 + w_2x_2 + w_3\log(x_2) + w_4\log(x_3) + w_5(x_2x_3) + w_6$$

Vediamo ora le criticità (sia positive che negative) del basis expansion, per poi arrivare ai così detti approcci **"dizionari"**.

- **PROS:** È più espressivo: può modellare in modo più complicato relazioni (rispetto a quelle lineari).
- **CONS:** Con un'ampia base di funzioni, abbiamo bisogno di metodi per controllare la complessità del modello.

$$h_w(x) = w_0 + w_1x + w_2x^2 + \cdots + w_Mx^M = \sum_{j=0}^M w_jx^j$$



Notiamo che l'ultimo caso è molto più flessibile, ma potrebbe essere eccessivo. $E(w) = 0$ nei dati di training, diventa un modello troppo complesso, scarsa rappresentazione della funzione vera (verde) (a causa del sovradattamento).

Capiamo dunque che il tradeoff della complessità è che:

- Un modello troppo semplice: non fitta i dati in modo corretto **underfitting**
- Un modello troppo complesso: Altamente sensibile alle leggere perturbazioni dei dati **overfitting**

Vuoi scegliere la regolarizzazione per bilanciare i due casi attraverso il controllo della complessità del modello (intesa come flessibilità). Considerando che la complessità non è dovuta al costo computazionale ma a misura della flessibilità del modello per adattarsi ai dati.

12.7 Regularization

Con la **regularization** possiamo controllare l'overfitting andando a penalizzare la complessità delle funzioni con valori dei pesi grandi (w) (per es. verso il coefficiente restrimento) o il numero dei parametri liberi (w non uguale a zero). Tutto questo mentre si va a mantenere la flessibilità dello spazio delle ipotesi.

Occam (Ockham) razor: "La spiegazione più semplice è più probabilmente quella corretta" o "Preferiscono l'ipotesi più semplice che si adatta ai dati".

Questo concetto fondamentale nel ML verrà ritrovato e lo "razzionalizzeremo" alla fine.

Definizione 12.7.1. *Ridge regression o tikhonov regularization.* La possibilità di aggiungere limitazioni alla somma di valori di $|w_j|$ favorendo modelli "sparsi" ess. con meno termini dovuti ai pesi $w_j = 0$ (o vicino allo 0) (questo vuol dire modelli meno complessi).

$$\text{Loss}(h_w) = \sum_{p=1}^l (y_p - h_w(x_p))^2 + \lambda \|w\|^2 \quad \|w\|^2 = \sum_i w_i^2$$

Termine di errore dei dati + termine di regolarizzazione/penalità.

Abbiamo che λ è chiamato coefficiente di regolarizzazione (un parametro costante o un iperparametro). L'effetto che abbiamo da questa regolarizzazione è il **decadimento del peso** (praticamente si aggiunge $2\lambda w$ al gradiente della perdita).

$$w_{new} = w + \eta \cdot \Delta w - 2\lambda w$$

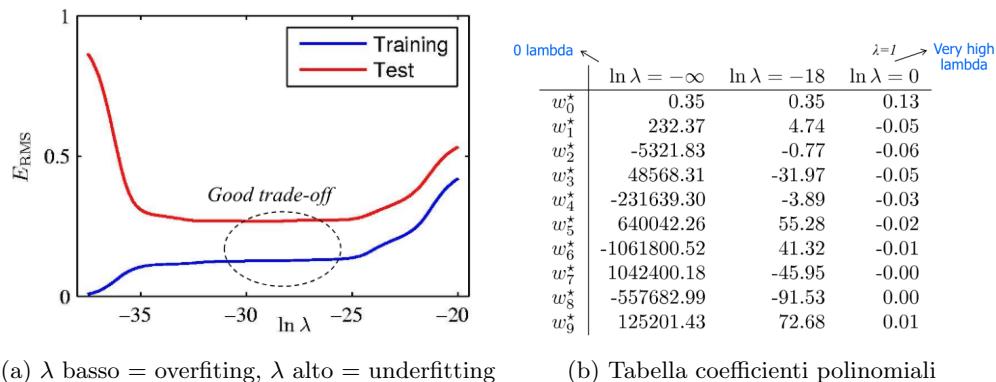
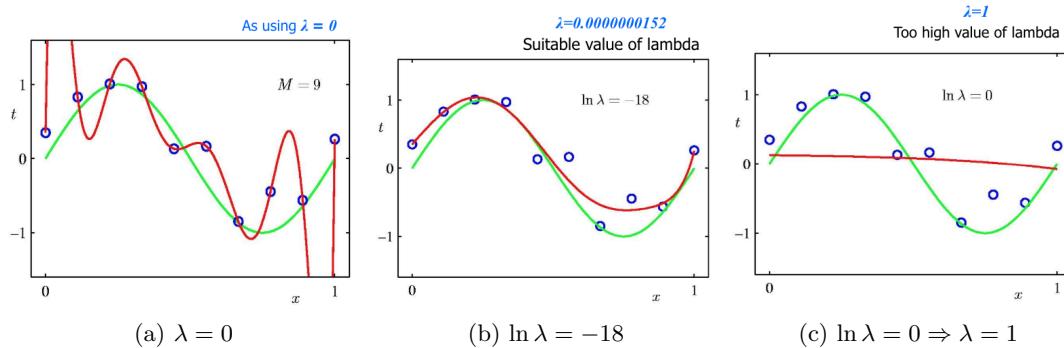
Per esempio con gradiente uguale a zero, esso decrementa il valore di ogni w con una frazione della vecchia w .

Applicabilità generale (non sono per i polinomi): esempio possiamo controllare la complessità del modello salmente usando λ , senza conoscere M come per i polinomi, o anche quando la complessità del modello non è conosciuta.

Notiamo il bilanciamento (compromesso) tra i due termini. Il termine relativo ai dati di errore di piccole dimensioni (minimizzare solo l'errore di addestramento) non è sufficiente per noi, vogliamo anche controllare la complessità del modello per contrallare l'**overfitting**, quindi introduciamo il secondo termine della minimizzazione.

Dall'altro canto possiamo eccedere perché troppo peso al secondo termine (alto valore λ) potrebbe focalizzare solo la minimizzazione (o principalmente) sulla regolarizzazione, quindi anche l'errore dei dati (primo termine) potrebbe aumentare molto, ovvero passare all'**underfitting**. Il compromesso è regolato dal valore di λ .

Vediamo ora l'esempio fatto prima con un polinomio di nono grado applicando un λ



12.7.1 Limitazioni di una funzione a base fissa

Avere funzioni di base lungo ci spieghiamo che ci sono molti modelli possibili. Per esempio, un polinomio generale di ordine 3 usa tutte le combinazioni delle variabili di input dovute per produrre $x_1, x_2, x_1x_2, x_3, \dots, x_1x_2x_3$ etc $\rightarrow n^3$ (approssimazione al crescere di n).

I dati diventano sparsi in questo alto volume, quindi l'allomanno dei dati necessari per supportare il risultato spesso cresce esponenzialmente con la dimensione. ϕ viene fissata prima di osservare i dati di allenamento.

In altri modelli, vedremo come possiamo farla franca con meno funzioni di base, andando a sceglierne queste usando i dati di training: ϕ (in uno strato computazionale nascosto) dipende. In altri modelli il calcolo di insulsione (trasformazione dell'input) è realizzato implicitamente attraverso le funzioni del kernel e controllando la complessità del modello. Per esempio SVM.

12.8 Classification

Andiamo a ri-usare i modelli lineari, infatti gli stessi modelli possono essere usati per la **classificazione**, ci sono delle categorie target per esempio 0/1 o $-1/+1$. In questo caso usiamo un iperpiano (wx) assumendo valori negativi o positivi. Sfruttiamo tali modelli per decidere se un punto x appartiene alla zona positiva o negativa dell'iperpiano (per classificarlo). Quindi vogliamo impostare w (nel learning) per ottenere una buona precisione di classificazione.

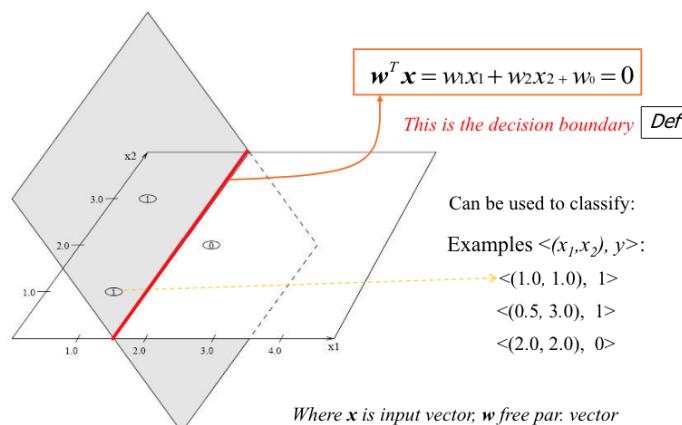
Per le notazioni di un input multidimensionale è come spiegato prima quindi, assumiamo una colonna vettore per x e w , numero di dati l , dimensione del vettore di input n , y_p che è d_p o t_p (targets 0/1 o $-1/+1$ è la cosa nuova).

$$w^T x + w_0 = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

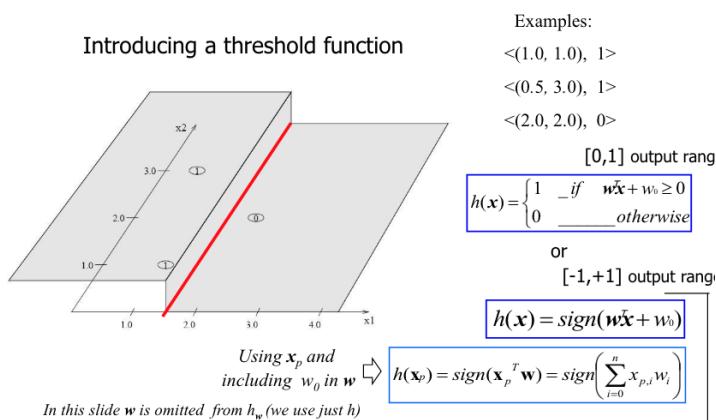
Con w_0 che è un intercept, threshold, bias, offset ... Genericamente è conveniente includere la costante $x_0 = 1$ così si può riscrivere il tutto come

$$w^T x = x^T w \quad x^T = [1, x_1, x_2, \dots, x_n] \quad w^T = [w_0, w_1, w_2, \dots, w_n]$$

Vediamo ora la vista geometrica dell'iperpiano. Visto che $w^T x$ definisce un iperpiano abbiamo che

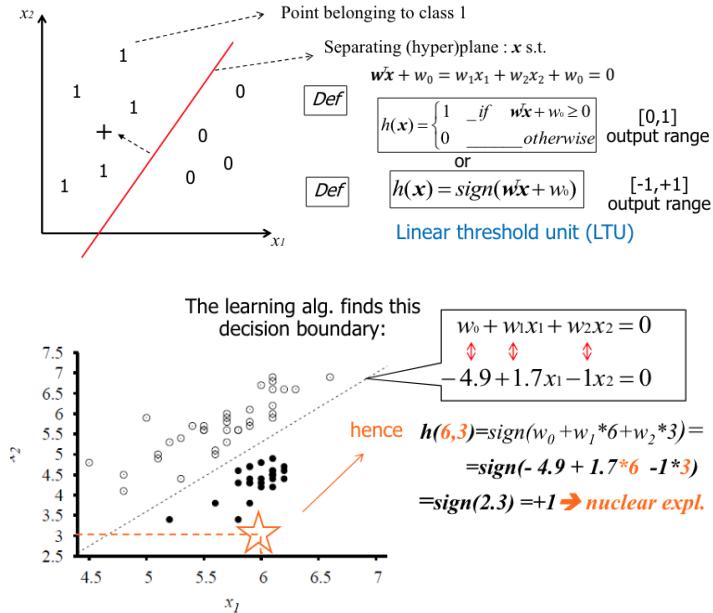


Da questo possiamo anche vedere la vista geometrica della classificazione. La classificazione può essere



vista come un'allocazione dello spazio di input nella regione di decisioni (es. 0/1). 2-dim spazio di input $x = (x_1, x_2) \in R^2$, $f(x) = 0/1$ (o $-1/+1$). Notiamo che, dato una bias w_0 , nella LTU diciamo che $h(x) = w^T x + w_0 \geq 0$ che è equivalente nel dire $h(x) = w^T x \geq -w_0$ con $-w_0$ abbiamo il valore "soglia".

Le due forme identificano la stessa zona positiva della classificazione, la seconda in particolare enfatizza il ruolo del bias come un valore soglia per "attivare" il +1 output del classifier.



Esempio 12.8.1. Trovare h tale che dato (x_1, x_2) restituisce 0/−1 per terremoti e +1 per una esplosione nucleare. Vediamo che l'algoritmo di learning trova questo confine di decisione: Dati sismici: x_1 magnitudo dell'onda corporea, x_2 magnitudo delle onde superficiali terremoti (bianco), esplosioni nucleari (nero).

Esempio 12.8.2. Un altro esempio è quello dello spam. Trovare $h(mail) + 1$ per spam, e −1 per non-spam. Le caratteristiche sono che $\phi(mail) = \text{parole}[0/1]$ o frasi ("free money") [0/1] o lunghezza [intero]. Per esempio $\phi_k(x) = \text{conain}(word_k)$. Il peso w contributo delle caratteristiche di input per la previsione, per esempio peso positivo per "denaro gratuito", negativo per ".edu". $x^T w$ è la combinazione dei pesi. $h_w(x)$ fornisce la soglia per decidere se spam o non spam.

$$h_w(x) = \text{sign}(\sum_k w_k \phi_k(x)) > 0 \rightarrow +1 = \text{Spam}$$

Vediamo ora Δw come correzione degli errori, regola di apprendimento.

$$\Delta w_i = \sum_{p=1}^l (y_p - x_p^T w) \cdot x_{p,i}$$

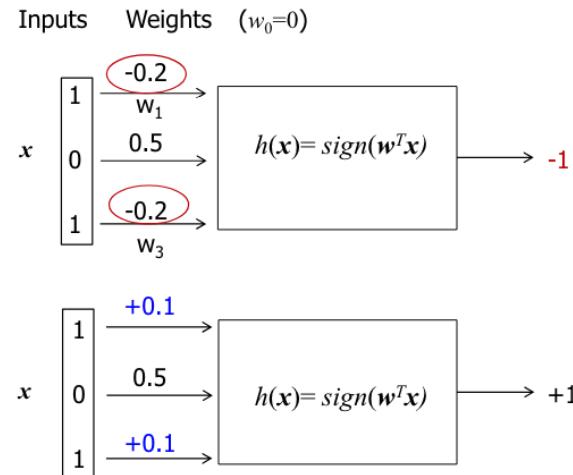
Se classificato erroneamente (Perché il bersaglio è +1) → (1-(??)) → delta positivo elevato per w_1 e w_3 → verranno aumenti proporzionalmente (con età) a il delta, quindi valore positivo in questo caso (errore regola di correzione). Per esempio la parte rossa in figura.

12.9 Learning algorithm

Come per la regressione si possono applicare anche l'espansione in base lineare e Tikhonov, notare che (algoritmo di apprendimento) converte asintoticamente al MinLS e non necessariamente al numero minimo di errori di classificazione. [(#ML)] se w non vengono modificati per ottenere output corretti ($h_w(x_p) = d_p$) otteniamo la regola di aggiornamento per per **percettore**

Esempio 12.9.1. Vediamo l'esempio della congiunzione. Possiamo andarla a rappresentare tramite un modello lineare per esempio:

$$h(x) = \begin{cases} 1 & \text{if } w \cdot x + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

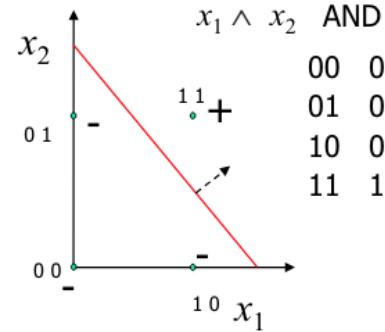


La **congiunzione** avviene nel seguente modo:

- 4 var $x_1 \wedge x_2 \wedge x_3 \wedge x_4 \Leftrightarrow y$
- $1x_1 + 1x_2 + 0x_3 + 1x_4 > 2$
 - $1x_1 + 1x_2 + 0x_3 + 1x_4 \geq 2.5$

- 2 var: $x_1 \wedge x_2 \Leftrightarrow y$

- $1x_1 + 1x_2 > 1$
- $1x_1 + 1x_2 \geq 1.5$

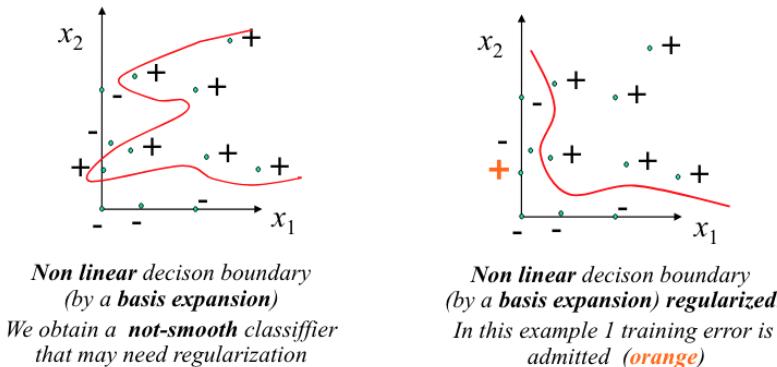


In geometrica due insiemi di punti in un grafico bidimensionale sono **linearmente separabili** quando gli insiemi di punti possono essere completamente separati da un'unica linea.

Definizione 12.9.1. In generale due gruppi sono detti **linearmente separabili** in uno spazio n -dimensionale se possono essere separati da un iperpiano di dimensione $(n-1)$.

Il confine decisionale lineare può fornire solo soluzioni esatte per insiemi di punti linearmente separabili.

Esempio 12.9.2. Esempi di limiti decisionali non lineari. Ricordiamo che anche l'**espansione in base lineare** e la **regolaizzazione di Tikhonov** può anche essere applicato.



Ci sono anche altri modelli lineari per la classificazione:

- Analisi discriminante lineare (anche multiclass)

- Regressione logica. Nel quale $P(y|x)$ parte dalla modellazione della densità di classe come densità nota. Soglia leggera (continua, differenziabile) con una funzione logica.
- Reti neurali (NN) e SVM menzionate in precedenza: **modelli flessibili** includono l'approssimazione non lineare per entrambe le regressioni e classificazioni. [#ML].
 - Le NN utilizzano molte unità (simili a LTU) all'interno di diversi strati
 - Apprendimento della rappresentazione delle funzionalità in ogni livello (concetto di deep learning)
 - Approccio di discesa del gradiente per l'apprendimento.

In conclusione i modelli linari sono degli aprocci di base ben fondati sia per la regressione che per la classificazione. Sono un modo molto compatto per rappresentare la conoscenza, ma con un forte presupporto sulla relazione tra i dati.

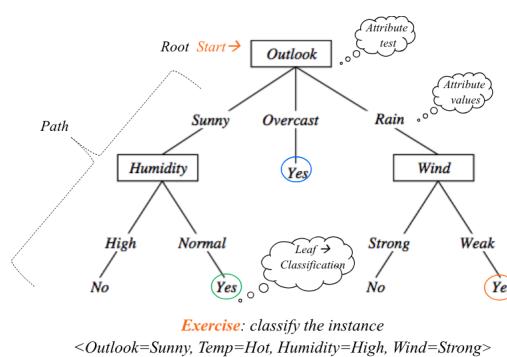
Un algoritmo iterativo correzione degli errori (LMS) che esegue la ricerca in spazi di ipotesi continui (questa è à base per molti altri approcci ML).

Una visione dei limiti degli approcci lineari e delle necessità di modelli ML più **flessibili** e relativi problemi sono un'estensione del modello lineare per compiti non lineari e un'introduzione al **controllo della complessità (regolaizzazione)**.

13 Alberi decisionali

Prendiamo ora un esempio di training set di partite di tennis e costruiamo l'albero decisionale.

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



Un albero decisionale rappresenta una disgiunzione di congiunzioni di vincoli sul valore degli attributi.

1. (verde) (*Outlook* = Sunny \wedge *Humidity* = Normal) \wedge
2. (blu) (*Outlook* = Overcast) \wedge
3. (rosso) (*Outlook* = Rain \wedge *Wind* = Weak)

H di DT capace di esprimere qualsiasi funzione finita a valori discreti (proposizionale).

13.1 ID3 algorithm

ID3 è un algoritmo base per l'apprendimento sui decision tree. Dato un training set di esempi, l'algoritmo per costruire un decision tree esegue una ricerca nello spazio dei decision tree. La costruzione dell'albero è di tipo **top-down** e l'algoritmo esegue una ricerca di tipo **greedy**.

La domanda fondamentale è "quale attributo dovrebbe essere testato successivamente? Quale domanda ci dà più informazioni"

- Selezionare il **miglior attributo**
- Viene quindi creato un nodo discendente per ogni possibile valore di questo attributo e gli esempi sono suddivisi in base a questo valore.
- Il processo viene ripetuto per ogni nodo successore finché tutti gli esempi sono classificati correttamente o non ci sono più attributi.

```

// X: esempi di training
// T: attributo target
// Attrs: altri attributi, inizialmente tutti gli attributi
ID3(X, T, Attr)
    Create Root Node
    if all X are +: return Root with class +
    if all X are -: return Root with class -
    if Attrs is empty: return Root with class most common value of T in X
    else
        A <- best attribute; decision attribute for Root <- A
        for each possible value v_i of A:
            add a new branch below Root, for test A = v_i
            X_i <- subset of X con A = v_i
            if X_i is empty then
                add a new leaf with class the most common value of T in X
            else then
                add the subtree generated by ID3(X_i, T, Attr - {A})
        return Root

```

Andiamo ad usare la nozione di **entropia**, comunemente usata nella teoria dell'informazione. L'entropia misura l'impurità di una collezione di esempi. Essa dipende dalla distribuzione di una variabile casuale p

- S è una collezione di esempi di training.
- p_+ la proporzioni di esempi positivi in S .
- p_- la proporzione di esempi negativi in S .

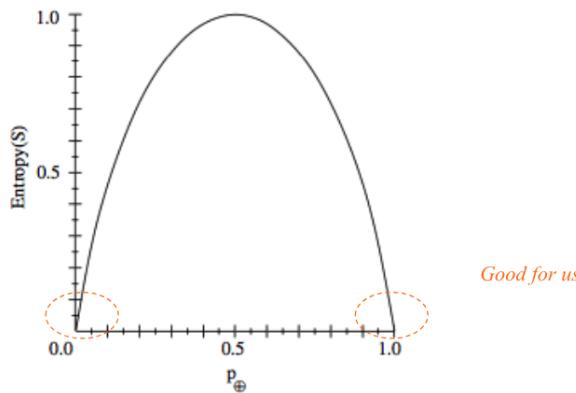
Definizione 13.1.1. Definiamo matematicamente **entropia** come:

$$\text{Entropy}(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_- \quad [\text{assumento } 0 \log_2 0 = 0]$$

Esempio 13.1.1. Alcuni esempi possono essere i seguenti:

- $\text{Entropy}([14+, 0-]) = -14/14 \log_2(14/14) - 0 \log_2(0) = 0$
- $\text{Entropy}([9+, 5-]) = -9/14 \log_2(9/14) - 5/14 \log_2(5/14) = 0.94$
- $\text{Entropy}([7+, 7-]) = -7/14 \log_2(7/14) - 7/14 \log_2(7/14) = 1/2 + 1/2 = 1$ questo è un caso con alta impurità

Note 13.1.1. Notare che $0 \leq p \leq 1, 0 \leq \text{entropy} \leq 1$



Se abbiamo tutti i +1 o tutti i -1 allora abbiamo 0 entropia cioè "no informazioni". Il massimo per S lo abbiamo invece con esempi 1/2+ e 1/2-.

Il guadagno di informazioni è la riduzione prevista dell'entropia causata partizionando gli esempi su un attributo.

Definizione 13.1.2. Definiamo la **Riduzione attesa dell'entropia conoscendo A**

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

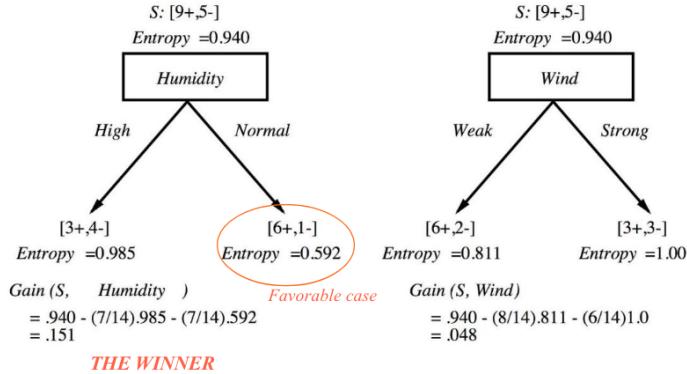
Dove $\text{values}(A)$ sono i possibili valori per A , mentre S_v è un sottoinsieme di esempi di S per i quali A ha valore v (somma di pesi)

Maggiore è il guadagno di informazione, più efficace è l'attributo classificazione dei dati di addestramento (maggior variazione nell'omogeneità dei distribuzioni delle classi nei sottoinsiemi, massima separazione delle classi). **Omogeneo**, poiché $[14+, 0-]$ o $[0+, 7-]$ ci consentono una classificazione "chiara".

L'entropia misura l'omogeneità (anzi impurità) della classe del sottoinsieme di esempi, quindi: **Selezionare A che massimizzi**

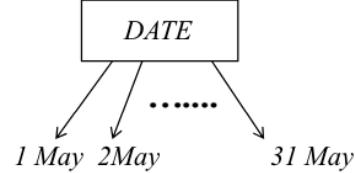
$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

Dopo la suddivisione, valori più bassi (obiettivi più omogenei) in ciascun sottoinsieme porta ad un guadagno più elevato. Lo scopo è quello di separare gli esempi in base al target, trovare l'attributo che discrimina gli esempi a cui appartengono diverse classi target.



Il guadagno di informazioni favorisce attributi con molti valori possibili. Consideriamo l'attributo "Date" nell'esempio "PlayTennis".

"Date" ha il massimo guadagno di informazioni. Ogni giorno corrisponde ad un sottoinsieme differente che è puro: $[1+, 0-]$ o $[0+, 1-] \rightarrow 0$ entropia. Abbiamo un adattamento perfetto (separazione) dei dati di training, ma non è significativo: non serve per i casi invisibili.



Chiediamoci ora come evitare la generazione di molti piccoli sottoinsiemi?

Definizione 13.1.3. Definiamo quello che è chiamato **gain ratio**

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInInformation}(S, A)}$$

Dove abbiamo che:

$$\text{SplitInInformation}(S, A) = - \sum_{i=1}^C \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Abbiamo che S_i sono degli insiemi ottenuti partizionando sul valore v_i di A fino al valore c . SplitInInformation misura l'entropia di S rispetto ai valori di A . Più i dati sono distribuiti uniformemente, più sono alti.

Il GainRatio penalizza gli attributi che dividono gli esempi in molte piccole classi come per Date. Sia $|S| = n$ la divisione degli esempi di date in n sottoinsiemi.

$$\text{SplitInInformation}(S, \text{Date}) = -[(1/n \log_2 1/n) + \dots + (1/2 \log_2 1/n)] = -\log_2 1/n = \log_2 n$$

Il quale è > 1 per $n > 2$ quindi riduciamo il GainRatio.

Confronto con un A che divide i dati in due classi pari (con $(n/2)/n = 1/2$):

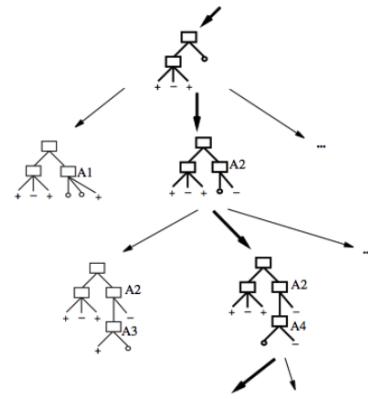
$$\text{SplitInInformation}(S, A) = -[(1/2 \log_2 1/2) + (1/2 \log_2 1/2)] = -[-1/2 - 1/2] = 1 \rightarrow \text{no reduction}$$

Abbiamo un problema: $\text{SplitInInformation}(S, A)$ può essere zero o molto piccolo quando $|S_i| \approx |S|$ per qualche valore i [$\log 1 = 0$]. Per mitigare questo effetto, vengono usate le seguenti euristiche:

1. Calcolare *Gain* per ogni attributo.
2. Applicare *GainRatio* solo per gli attributi con *Gain* sopra la media.

La ricerca nello spazio delle ipotesi attraverso ID3 consiste nel cercare (hill-climbing) attraverso lo spazio delle possibili DT dal più semplice al sempre più complesso. W.r.t. per l'algoritmo precedente:

- Lo spazio delle ipotesi è completo (rappresenta tutti i valori discreti funzioni).
- Nessun passo indietro; nessuna garanzia di ottimalità (ottimale locale).
- Utilizza tutti gli esempi disponibili (non incrementale)
- Può terminare prima, accettando classi "rumorose"



Parliamo ora degli **bias induttivi** nell'apprendimento per i decision-tree. Quali sono?

1. Gli alberi più piccoli sono preferibili rispetto a quelli più grossi. A causa della ricerca da semplice a complessa, incrementalmente. Questo però non è sufficiente questo sarebbe il bias che esibirebbe un semplice algoritmo breadth first generando tutti i DT e selezionando quello più breve e coerente.
2. Preferiamo alberi che posizionano l'informazion gain più alto più vicina alla root.

Note 13.1.2. DT non sono limitate alla rappresentazione tutte le possibili funzioni, la restizione non riguarda lo spazio delle ipotesi, ma la strategia di ricerca.

Definizione 13.1.4. *Definiamo le preferenze o **search bias** (dovute alla strategia di ricerca).*

ID3 ricerca uno spazio completo di ipotesi, la strategia di ricerca è incompleta.

Definizione 13.1.5. *Definiamo le restrizioni o **language bias** (a causa dell'insieme di ipotesi esprimibile o considerato).*

Visto nel candidate-elimination dove sono ricercati in uno spazio di ipotesi incompleto, la strategia di ricerca è completa (tutte le ipotesi consistenti, nel version space)

Perché il search bias viene preferito rispetto al language bias? Nel ML tipicamente si usano approcci **flessibili** (capacità universale dei modelli) senza escludere a priori la funzione bersaglio sconosciuta. Ovviamente la flessibilità può portare a problemi di overfitting.

Perché preferire ipotesi più brevi? Come disse Ockham nel 1300 con il razoio di Occam "La spiegazione più semplice è più probabilmente quella corretta" o "Prefisco l'ipotesi più semplice che si adatta ai dati" Il termine rasoio si riferisce all'atto di radere via inutili ipotesi per arrivare alla spiegazione più semplice. Viene anche detta "Legge di parsimonia" o "legge di economia".

Ricorda il "controllo della complessità del modello" mediante la regolarizzazione per modelli lineari. Ancora, questo concetto fondamentale nel ML verrà ritrovato e "razzionalizzato" più avanti.

13.2 Risolvere problemi ID3

Ci sono diverse problematiche nell'apprendimento per gli alberi di decisioni fra questi abbiamo:

- Overfitting. Che si divide a sua volta in: arresto anticipato, ridotta potatura degli errori, regola post-potatura.
- Estensioni (specifico per i DT): attributi a valore continuo, gestire esempi di formazione con valori di attributo mancanti, gestire attributi con costi diversi, miglioramento dell'efficienza computazionale.

13.2.1 Overfitting

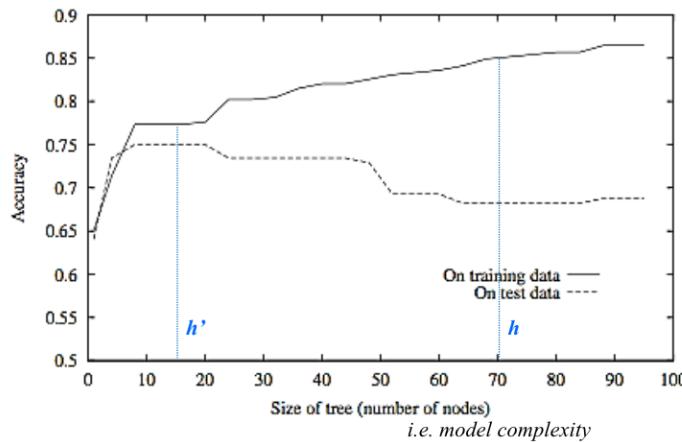
Partiamo dal problema dell'overfitting. Costruire alberi che "si adattano troppo" agli esempi formativi può portare a un "overfitting". Consideriamo l'errore su un ipotesi h su: dati di training: $\text{error}_D(h)$, intera distribuzione X di dati: $\text{error}_X(h)$

Definizione 13.2.1. Si dice che l'ipotesi h overfits i dati di training se c'è un'ipotesi alternativa $h' \in H$ tale che:

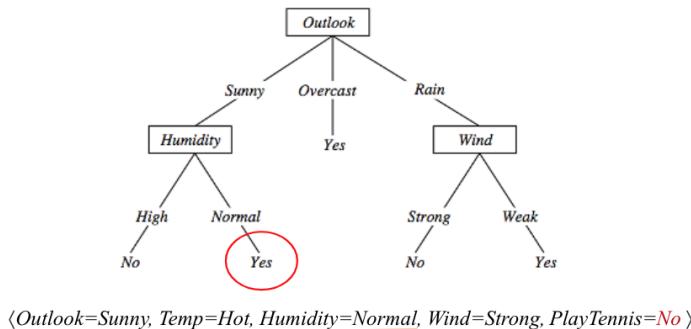
$$\text{error}_D(h) < \text{error}_D(h') \quad e \quad \text{error}_X(h') < \text{error}_X(h)$$

Cioè h' si comporta peggio con i dati TR, meglio con i dati invisibili.

Gli approcci flessibili possono facilmente incontrare un overfitting (se utilizzati senza cura speciale)



Esempio 13.2.1. Vediamo un esempio con dati "noisy".



$\langle \text{Outlook}=\text{Sunny}, \text{Temp}=\text{Hot}, \text{Humidity}=\text{Normal}, \text{Wind}=\text{Strong}, \text{PlayTennis}=\text{No} \rangle$

Immagina che questo nuovo esempio rumoroso causi la divisione del secondo nodo foglia, allora DT cresce (la sua complessità cresce). Adattandosi anche al rumore, il nuovo DT è perfetto per i dati TR, non per i nuovi dati.

Per "evitare" l'overfitting nei DT, possiamo usare una di queste due strategie:

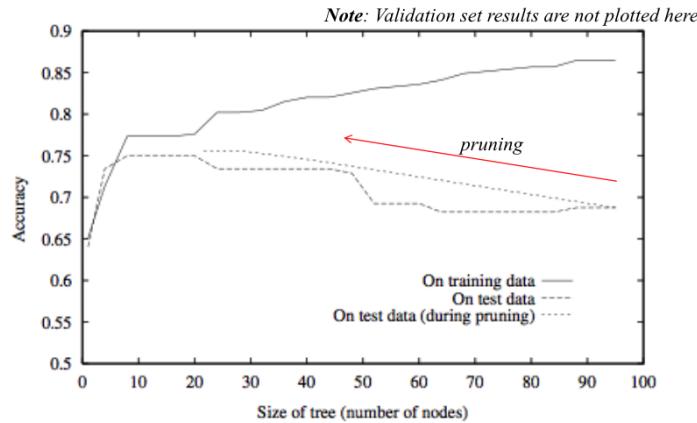
- Smettere di far crescere l'albero prima, prima della classificazione perfetta ("arresto anticipato") vedere la trama prima)
- Consentire all'albero di adattarsi eccessivamente ai dati, quindi potare l'albero.

Come possiamo valutare l'effetto? Un primo modo può essere quello di usare un set di training e validazione, andando a dividere il training set in due parti (training set e validation set) e utilizzare il set di validazione per valutare l'utilità di 1 e 2.

Altri approcci possono essere utilizzare un test statistico per stimare l'effetto dell'espansione o della potatura. **Principio della lunghezza minima:** utilizza una misura di complessità per codificare il DT e gli esempi (classificati erroneamente) e interrompe la crescita dell'albero quando questa dimensione di codifica è minima.

13.2.2 Reduced-error pruning

Ogni nodo è un candidato per la potatura, la potatura consiste nel rimuovere un sottoalbero radicato in un nodo: il nodo diventa una foglia e gli viene assegnata la classificazione più comune, i nodi vengono rimossi solo se l'albero risultante non ha prestazioni peggiori. I nodi vengono potati iterativamente: ad ogni iterazione il nodo di cui la rimozione aumenta maggiormente la precisione del set di convalida e viene ridotta. La potatura si interrompe quando nessuna potatura aumenta la precisione.



13.2.3 Regole post-potatura

Abbiamo alcune regole che si applicano post-potatura:

1. Create il decision tree dal training set.
2. Convertire l'albero in un set di regole equivalenti dove:
 - Ogni **path** corrisponde ad una regola.
 - Ogni **nodo** lungo un cammino corrisponde ad una pre-condition.
 - Ogni **foglia** classificata come post-condition.

Esempio: $(Outlook = Sunny) \wedge (Humidity = High) \Rightarrow (PlayTennis = No)$

3. Potare (generalizza) ogni regola rimuovendo tali precondizioni la cui rimozione migliora la precisione oltre il set di validazione e oltre il training con una misura, statisticamente ispirata, pessimistica.
4. Ordina le regole in ordine di precisione stimato e considerale come una sequenza durante la classificazione di nuove istanze.

Perché convertire in regole?

Ogni percorso distinto produce una regola diversa: una condizione di rimozione può basarsi su un criterio locale (contestuale), l'eliminazione delle precondizioni è specifica della regola (percorso), l'eliminazione dei nodi è globale ed interessa tutte le regole (sottoalbero), la conversione in regole migliora la leggibilità per gli essere umani (assumento un numero limitato di regole).

13.2.4 Attributi a valori continui

Fino ad ora abbiamo avuto valori discreti per attributi e risultati. Dato un attributo A a valore continuo, creare dinamicamente un nuovo valore A_c

$$A_c = \text{True if } A < c \text{ else False}$$

Come si determina la soglia del valore c ?

Esempio 13.2.2. La temperatura nell'esempio di PlayTennis. Ordinare gli esempi in base alla temperatura.

<i>Temperature</i>	40	48	60	72	80	90	
<i>PlayTennis</i>	No	No	54	Yes	Yes	85	No

Determinare le soglie dei candidati calcolando la media dei valori consecutivi dove c'è un cambio di classificazione: $(48+60)/2 = 54$ e $(80+90)/2 = 85$.

Valutare le soglie dei candidati (attributi) in base a guadagno di informazioni. La temperatura migliore è 54. Quindi, il nuovo attributo compete con gli altri.

13.2.5 Gestire dati di training incompleti

Come possiamo affrontare il problema che il valore di alcuni attributi potrebbe **mancare**? Per esempio risultato dell'esame del sangue in un problema di diagnosi medica.

La strategia: usa altri esempi per indovinare l'attributo (entro dati di addestramento), imputazioni:

1. La cosa più comune è quella di assegnare il valore più comune tra tutti gli esempi di training al nodo o nodi nella stessa classe.
2. Assegnare una probabilità p_i ad ogni valore v_i , in base alle frequenze, ed assegnare valori all'attributo mancante, in base a questa distribuzione di probabilità (aggiungendo più esempi ponderati dalla probabilità, cioè assegnare la frazione p_i dell'esempio a ciascun albero discendente)
3. Classificare il nuovo esempio allo stesso modo (ponderazione): e viene scelta la classificazione più probabile.

13.2.6 Gestire attributi con costi differenti

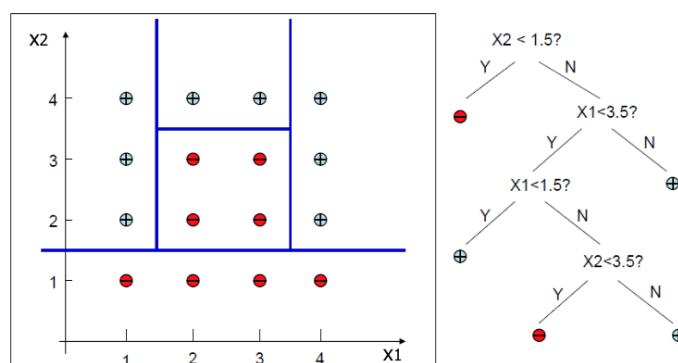
Gli attributi dell'istanza possono avere un costo associato: lo faremo preferire alberi decisionali che utilizzano attributi a basso costo. L'ID3 può essere modificato per tenere conto dei costi. Una prima versione di Tan e Schlimmer (1990)

$$\frac{Gain^2(S, A)}{Const(A)}$$

Una seconda versione di Nunez (1988)

$$\frac{2^{Gain(S,A)} - 1}{(Cost(A) + 1)^w} \quad \text{con } w \in [0, 1]$$

Una vista geometrica. Confini decisionali che possono essere prodotti da un DT: gli alberi decisionali dividono lo spazio di input in rettangoli ed etichette paralleli agli assi, ogni rettangolo con una delle classi K (foglie dell'albero).



14 Validation

Definizione 14.0.1. Definiamo **selezione del modello** la stima delle prestazioni (generalizzazione errore) di diversi modelli di apprendimento al fine di scegliere il migliore (per generalizzare).

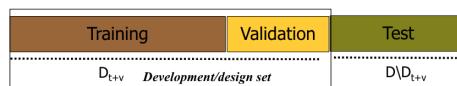
Questo include la ricerca dei migliori iperparametri del tuo modello (es. ordine polinomiale, lambda di regressione di cresta, ...)

Definizione 14.0.2. Definiamo come **valutazione del modello** quando dopo aver scelto uno modello finale, si va a stimare/valutare il proprio errore/rischio di previsione (generalizzazione errore) su nuovi dati di test (misura della qualità/prestazione del modello scelto alla fine).

14.1 Hold out

Una regola molto importante è quello di mantenere la separazione tra gli obiettivi e utilizzare set di dati separati in ogni fase.

Se la dimensione del set di dati è sufficiente, per esempio 50% TR, 25% VL, 25% set disgiunti.



Vediamo ora quello che chiamiamo **hold out**:

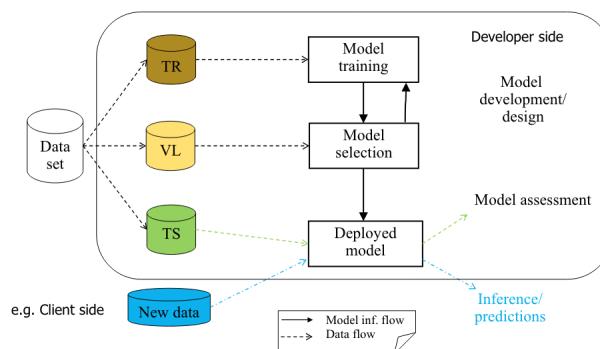
- **TR:** training set è usato per adattarsi [training]
- **VL:** Validation set (o selection set) è usato per selezionare il miglior modello (fra differenti modelli e/o configurazioni di iperparametri) [model selection]
- **TR+VL** alcune volte sono insieme e vengono chiamate **insieme di development/design**, usate per costruire il modello finale.
- **TS:** test set è usato per stimare l'errore di generalizzazione (del modello finale) [valutazione del modello]

Note 14.1.1. Notare che la stima fatta per la selezione del modello (sul set VL) [è a scopo di selezione del modello], non è una buona stima per la dase di valutazione/test di rischio. Inoltre i risultati del set di test non possono essere utilizzati per la selezione del modello (o chiamarlo set di convalida).

E cosa succederebbe se si usasse il test set in un ciclo di progettazione (ripetuto)?

Stiamo effettuando una selezione del modello e una valutazione non affidabile (stima di errore di generalizzazione previsto) e non saremmo in grado di farlo su esempi futuri, concetto di **set di test ciechi** (ad esempio nelle competizioni di ML). In tal caso, l'errore del set di test fornisce una valutazione eccessivamente ottimistica per il vero errore del test (vedremo come è facile ottenere una classification accuracy molto alta un un compito casuale anche utilizzando solo il set di test implicito).

Gold rule: Mantenere la separazione tra gli obiettivi e utilizza set separati in ogni fase (TR per la formazione, VL per la selezione del modello, TS per la stima del rischio)



14.2 Un semplice meta-algoritmo

Vediamo i vari passi di semplice meta algoritmo:

1. Separare gli insiemi **TR** (training), **VL** (validation) e **TS** (test).
2. Cercare il migliore $h_{w,\lambda}()$ modificando gli iperparametri del modello λ (es. l'ordine del polinomio, il valore del λ per la regressione)
3. Per ogni diverso valore di λ (ricerca in griglia)
 - Cerca il migliore $h_{w,\lambda}()$ che minimizzi l'errore/perdita empirica (adattandosi al set TR), trovare i migliori parametri w dove migliore = errore minimo sul set TR (es. $\operatorname{argmin}_w Loss(w) \in L_2$)

Questo è un doppio ciclo, la ricerca migliore può essere un for loop, per ogni valore λ si addestra un modello $h_{w,\lambda}$ (nel ciclo interno, ad esempio il ciclo di discesa del gradiente) e quindi si calcolano i risultati (accuracy) sul set VL. Quindi prendi il valore miglior di λ ovvero il modello con errore VL minimo o precisione VL massima, ecc.

4. Selezionare il miglior $h_{w,\lambda}()$: dove migliore = errore minimo sul set VL.
5. (Opzionale) Ora è anche possibile adattare $h_{w,\lambda}(x)$ su TR+VL con il miglior modello λ
6. Valutare il finale $h_{w,\lambda}$ nel TS.

Esempio 14.2.1. Vediamo per esempio la ricerca su una griglia con 2 iper-parametri.

Trovare il valore degli iperparametri (ovvero i parametri che non sono direttamente appresi, che non vengono modificati dalla formazione). Il miglior iperparametro di ricerca può essere un <<FOR>> su una griglia di valori candidati. Per ogni modello di training $h_{w,\lambda}$ calcolare i risultati (accuracy) sul VL impostato. Quindi prendi quello con l'errore minimo o la precisione massima.

Hyper-param.	Lambda 0.1	Lambda 0.01	Lambda 0.001
Degree 1	Res1	Res4	Res7
Degree 2	Res2	Res5	Res8
Degree 4	Res3	Res6	Res9

E.g. "Res1" is computed on the VL set,
by the model with and Polynomial-Degree=1
and Lambda=0.1 trained on the TR set

Example: The best one is Res3 → (Degree 4, lambda=0.1) is the winner

Possiamo automatizzarlo. La parallelizzazione è semplice (indipendenza delle prove), esistono alternative per ridurre i costi o automatizzare la ricerca.

Esempio 14.2.2. Vediamo ora un controesempio (separare TR, VR e TL). Abbiamo circa 20-30 esempi, 1000 variabili di input random. un **random** target 0/1. Scelgo 1 modello con una sola variabile/feature che in indovina 'per caso' al 99% sul dataset e poi su qualsiasi split successivo in training, validation e test set.

Abbiamo che il valore perduto 99% non è una buona stima dell'errore di test (quello corretta è 50%)

1. Errore stimato su training o validation per model selection NON è utile per stima del rischio!
Dati di TR o VL non vanno usati per scopi di test.
2. Usare tutto il data set per feature/model selection lede la correttezza della stima (risultati biased - <<Feature Selection bias>>). Test se è stato usato implicitamente dall'inizio. Test deve essere separato prima, prima di qualsiasi model selection o design del modello (incluso selezione di features)

Un test set esterno fornisce invece la stima corretta del 50% (random coin result). È la correttezza della stima che è in giudizio, non la possibilità di risolvere la task.

Pattern	Input variable value										Target
	1	2	...	26	27	28	...	1000	...		
1	1	1	1	1	
2				0	0	0				0	
3				1	1	1				1	
4				0	0	0				0	
...				0	0	0				0	
...				1	1	1				1	
...				0	0	0				0	
20	1	1	1	1	
TS1				1	0	0				1	
TS2				0	1	0				0	
TS3				1	1	1				1	
Accuracy				100%	33%	66%					

14.3 K-fold

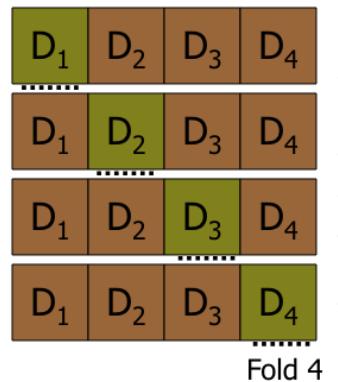
Mantere il CV più fare un uso insufficiente dei dati.

Definizione 14.3.1. Definiamo la *convalida incrociata k-fold* come:

- Suddividere il set di dati D in k sottoinsiemi mutualmente esclusivi D_1, D_2, \dots, D_k
- Addestrare l'algoritmo di apprendimento su $D \setminus D_i$ e testarlo su D_i
- Riassumere la media di tutti i risultati D_i (diagonale)
- Utilizza tutti i dati per la formazione, la convalida o il test.

Note 14.3.1. Nota che questa tecnica può essere utilizzata sia per il validation set che per il set di prova.

Ci sono alcuni problemi di questa tecnica fra cui: decidere quanti "fold", spesso computazionalmente molto costosa, abbinabile al set di validazione, doppio k-fold CV, ...

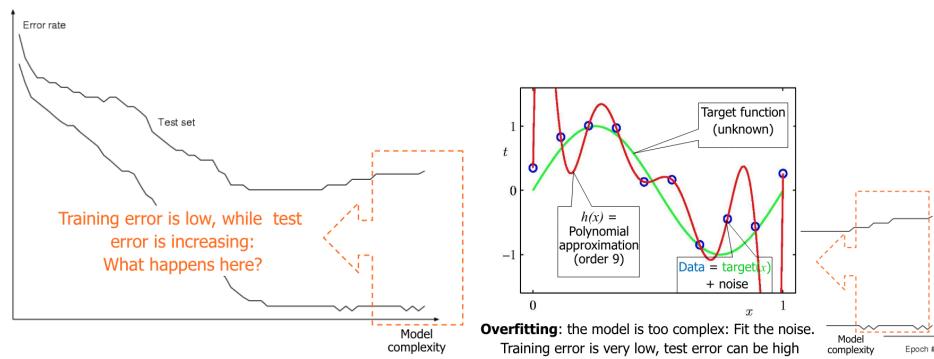
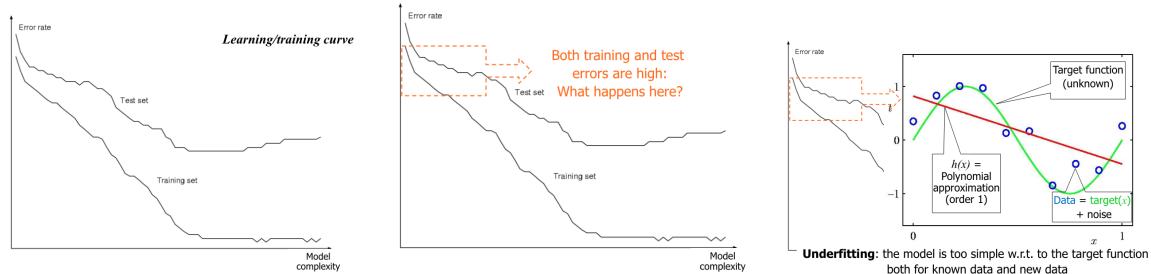


Esempio 14.3.1. Un esempio di selezione del modello e valutazione (con K-fold CV).

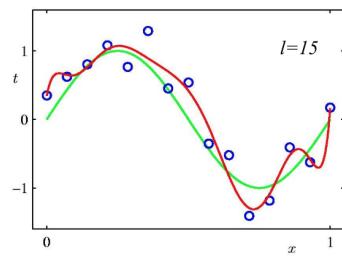
- Andiamo a dividere i dati in **TR** e **Test set** (qui semplice hold-out o k-fold CV)
- [Selezione del modello] Utilizzare k-fold CV (internal) sul set TR, ottenendone di nuovi TR e VL impostati in ogni split, per trovare i migliori iperparametri del tuo modello (es. ordine polinomiale, lambda della ridge regression). Questo come?
Andiamo ad applicare una ricerca su gliglia con molti valori possibili dell'iper-parametro. Per esempio: un k-fold CV per $\lambda = 0.1$, un k-fold-CV per $\lambda = 0.01, \dots$ per poi prendere il migliore λ (controllando gli errori medi calcolati sul set di validazione ottenuti da tutte le folds per ogni k-folds CV, il risultato è sulla diagonale dell'immagine).

- Allenamento su tutto il TR impostato sul modello finale.
- [Valutazione del modello] valutarlo sul l'insieme di test esterno.

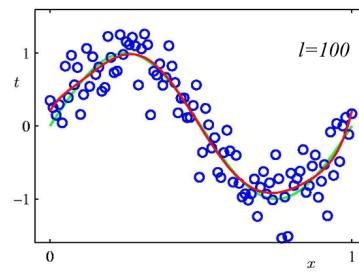
Vediamo un tipico comportamento nell'apprendimento.



9th Order Polynomial (changing the of number of data)



9th Order Polynomial (even more data)



14.4 SLT

Andiamo ora a mettere tutto insieme: la capacità di generalizzazione (misurata come rischio o errore del test) di un modello rispetto all'errore di formazione o rispetto alle zone di overfitting e underfitting. Poi mettiamo anche insieme il ruolo della complessità del modello, ed il ruolo del numero di dati. Quello che otteniamo è la **Teoria di apprendimento statistico** (SLT) che è una teoria generale relativa questi argomenti.

Definizione 14.4.1. Una funzione di approssimazione sconosciuta $f(x)$, d (o y , o t) è il target ($d = \text{true } f + \text{noise}$). Minimizzare la funzione di rischio:

$$R = \int L(d, h(x)) dP(x, d)$$

Dati i valori dall'apprendimento (d), la probabilità di distribuzione $P(x, d)$ e una funzione di perdita, ess: $L(h(x), d) = (d - h(x))^2$.

Cercare $h \in H$ che minimizzi R . Però non abbiamo solamente l'insieme finito dei dati $TR = (x_p, d_p) p = 1 \dots l$. Per cercare h andiamo a minimizzare il rischio empirico (errore di training E) frontando il migliori valori per il modello dei parametri liberi.

$$R_{emp} = \frac{1}{2} \sum_{p=1}^l (d_p - h(x_p))^2$$

Questo si chiama **principio induttivo della minimizzazione del rischio empirico (ERM)**

Possiamo usare R_{emp} per approssimare R . La risposta è SI ed è stato studiato da Vapnik-Chervonenkis.

Definizione 14.4.2. Definiamo la **VC-dim(VC)** come una misura di complessità dello spazio delle ipotesi

Definizione 14.4.3. Definiamo **VC-bounds** come un limite superiore al rischio. E può essere gestito con una probabilità di $1 - \delta$

$$R \leq R_{emp} + \epsilon(1/l, VC, 1/\delta)$$

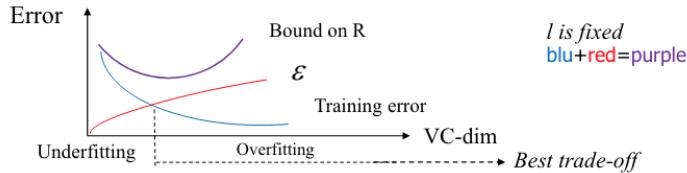
Mentre ϵ si chiama **VC-confidence**.

Una prima spiegazione è la seguente:

- ϵ è una funzione che cresce con VC (VC-dim) e decresce con (più alto) l e δ .
- Sappiamo che R_{emp} decremente usando un modello comlesso (con alta VC-dim) (esempio il grado del polinomio)
- δ è la confidenza, regola la probabilità che il bounds varia, roba statistica.

L'intuizione è che se abbiamo tanti dati l la VC-confidence si abbassa (tende a 0) ed il rischio empirico R_{emp} va verso R . Se invece ho un modello troppo semplice o rigido (bassa VC-dim) potrebbe essere non sufficiente dovuto al R_{emp} troppo alto (**underfittig**). Mentre un alto VC-dim, quindi grande complessità del modello, diminuisce il R_{emp} ma la VC-conf, e quindi R , potrebbe aumentare (**overfittig**)

Per avere la minimizzazione del rischio strutturale bisogna minimizzare il limite!



Concetto di controllo della complessità del modello (flessibilità): compromesso tra la TR accuracy (fitting) e la complessità del modello (VC-dim).

In sintesi possiamo dire che la SLT permette inquadramento formale del problema della generalizzazione e underfittig/overfittig, fornendo limitazioni superiori analitiche e quantitative al rischio R di predizione su tutti i dati, indipendentemente dal tipo di learning algorithm o dettagli del modello.

Il ML è ben fondato:

- Il rischio del learning (e l'errore di generalizzazione) può essere analiticamente limitato, e solo pochi concetti sono fondamentali.
- Si può trovare una buona approssimazione dell' f target da esempi, pure di avere un buon numero di dati e un'adeguata complessità del modello (misurabile formalmente con la VC-dim)

Il STL porta a nuovi modelli (SVM) (e altri metodi che direttamente considerino il controllo della complessità nella costruzione del modello). Fonda uno dei principi induttivi sul controllo della complessità.

Nei modelli lineari la complessità sembra correlata al numero di parametri liberi w : input dimensione /dim dell'espansione della base (ad esempio grado polinomio). Parametro lambda per la versione refolazzata (utilizzando il modello selezione/validazione tecnica per trovare corretto di lambda). Per il DT invece abbiamo il numero di nodi.

15 SVM

SVM sta per support vector machine ed è un classificatore derivato dalla teoria dell'apprendimento di Vapnik. Dopo anni di sviluppi teorici, SVM divenne famosa quando, utilizzando le immagini come input, ha fornito una precisione paragonabile a reti neurali SotA (negli anni 90) con funzionalità progettate a mano in un compito di riconoscimento della grafica.

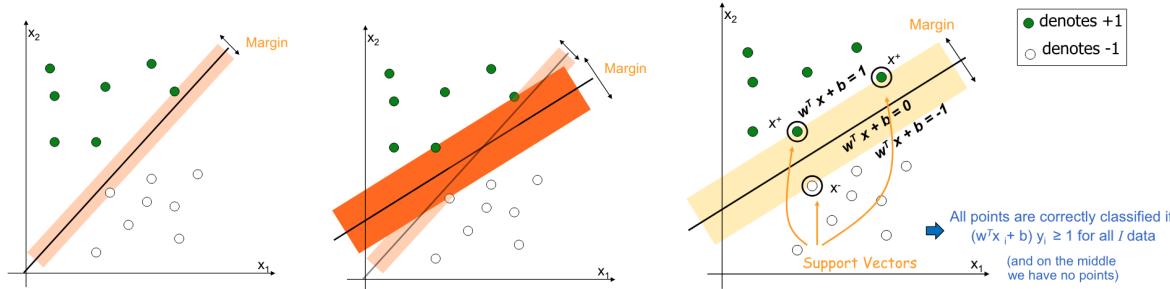
Attualmente SVM è ampiamente utilizzato in tutti i campi di applicazione di apprendimento supervisionato, utilizzato anche per la regressione. Anche se le attuali deep-neural-network spesso lo superano in molti ambiti (ad esempio analisi delle immagini).

Partiamo dal modello lineare per la classificazione, ed ancora, siamo interessati anche ad arricchirlo per problemi non lineari.

15.1 Maximum margin classifier

Problema di classificazione binaria. Cominciamo ad assumere un problema lineare separabile e assumiamo anche l'assenza di rumore (margin rigido del SVM).

Esempio 15.1.1. Facciamo un esempio di margini. Infatti non tutti gli iperpiani che risolvono il compito sono uguali, variando l'iperpiano di separazione varia anche il margine.



Definizione 15.1.1. Il **margine** è (il doppio della) distanza tra l'iperpiano di separazione e i punti dati più vicini (esempi di input).

Definizione 15.1.2. Definiamo anche **vettori di supporto**

$$x_p \text{ t.c. } |w^T x_p + b| = 1 \quad b = w_0$$

Consideriamo il problema di apprendere un modello lineare per la classificazione binaria, cioè una funzione nella forma $h : \mathbb{R}^n \rightarrow \{-1, 1\}$, come per esempio $h(x) = \text{sign}(wx + b)$, sulla base di esempi (x_p, y_p) nel TR set.

Definizione 15.1.3. Definiamo **training problem** come trovare (w, b) tale che tutti i punti siano classificati correttamente e il margine è massimizzato.

Un vincolo è che:

$$(w^T x_p + b)y_p \geq 1 \forall p$$

Che vuol dire che **tutti i punti sono correttamente classificati**. Si noti inoltre che, a differenza della soluzione LMS, per i casi separabili linearmente qui abbiamo 0 errori.

16 Riassunto modelli

Riassunto di tutti i modelli visti durante il la parte di ML.

Categoria	Nome	Descrizioni
Concept learning	Find-S	Ricerca in spazio di ipotesi partendo dalla più specifica
Concept learning	List-then-eliminate	Utlizza il concetto di version space rimuovendo per ogni esempio
Concept learning	Candidate Elimination	utilizza sempre VS ma con confine generale e
Modelli lineari	LMS + Gradiant descent	Si cerca w che minimizzi la Loss, per farlo di fa una ricerca dei
Modelli lineari	Linear basis expansion	si usa non solo per aumentare il numero di parametri ma
Modelli lineari	LMS + regularization	permette di limiare fenomeni di overfitting andando a penalizzare la
Alberi decisionali	ID3	
Modelli lineari	SVM	
Non lineare	K-NN	
Non supervisionato	K-means	