

Fondamenti di Intelligenza Artificiale



Fondamenti di Intelligenza Artificiale

Capitolo 1 – Agenti Intelligenti

1.1 Introduzione

L'**intelligenza artificiale** studia i fondamenti teorici, le metodologie e le tecniche che consentono di progettare sistemi hardware e software atti a fornire all'elaboratore prestazioni che, a un osservatore comune, sembrerebbero essere appartenere esclusivamente all'intelligenza umana. Lo scopo dell'AI non è quello di replicare tale intelligenza ma di riprodurne o emularne alcune funzioni.

La definizione di **Intelligenza Artificiale** comprende diversi aspetti, ci sono:

- elementi rivolti ai processi di pensiero e al ragionamento: misurano il successo in base alla somiglianza ad un'esecuzione umana;
- elementi rivolti rivolte al comportamento: usano come metro di paragone il concetto ideale di intelligenza, che noi chiamiamo razionalità. Un sistema è razionale se, date le sue conoscenze, “fa la cosa giusta”.

Pensare umanamente: il tentativo dei computer che arrivano a pensare. Quando diciamo che un determinato programma ragiona come un essere umano, dobbiamo prima di tutto capire quali sono i meccanismi interni del cervello umano. **Questo approccio è associato alle scienze cognitive**. Ci sono 3 modi di *modellare* i meccanismi interni del cervello umano:

Introspezione: consiste nell'osservazione e nell'analisi dell'interiorità rappresentata da pensieri, pulsioni, desideri e stimoli prodotti dal pensiero stesso. La modellazione consiste nel catturare “al volo” i nostri pensieri mentre scorrono.

Sperimentazione psicologica: è la branca della psicologia che mira ad applicare il metodo sperimentale di Newton nell'indagine dei processi cognitivi del cervello umano. La modellazione consiste nell'osservazione dei pensieri, pulsioni, desideri e stimoli di una persona in azione.

Imaging cerebrale: rappresenta l'utilizzo di tecniche per la mappatura diretta e/o indiretta della struttura, della funzione e della farmacologia del sistema nervoso. La modellazione, in questo caso, consiste nell'osservazione del cervello in azione, così da poter intuirne i meccanismi nervosi interni e trarne conclusioni.

Agire umanamente: il risultato dell'operazione compiuta dal sistema intelligente non è distinguibile da quella svolta da un umano. Un esempio è il **test di Turing** noto anche come “*The imitation game*”. Al test ci sono 3 partecipanti: una macchina (A), una donna (B) e un interrogante (C). C sarà isolato in una stanza, mentre A e B condivideranno l'altra. L'identità di A e B non è nota a C e, infatti, quest'ultimo li conoscerà come due persone ignote X e Y.

Obiettivo di C è di determinare quale delle due entità sia la macchina e quale la donna attraverso una serie di domande. Obiettivo di A è di ingannare C mentre B lo aiuterà.

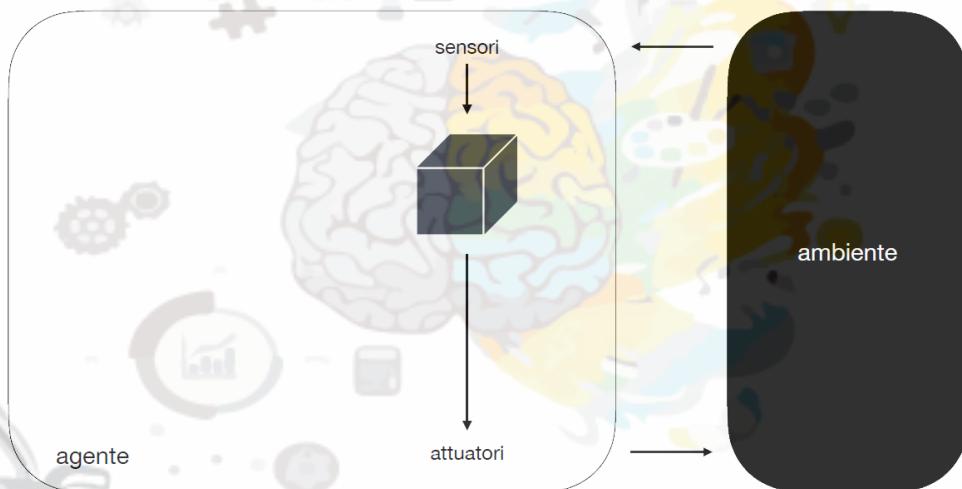
Secondo Turing, l'intelligenza può essere misurata guardando il comportamento esterno, non il ragionamento che ha portato a quel comportamento.

Pensare razionalmente: il processo che porta il sistema intelligente a risolvere un problema è un procedimento formale che si rifà alla logica.

Agire razionalmente: l'Intelligenza computazionale è lo studio della progettazione di **agenti intelligenti**. Un agente è qualcosa che agisce, che opera autonomamente, è in grado di percepire l'ambiente e raggiungere obiettivi. Un agente razionale agisce in modo da ottenere il miglior risultato o, in condizione di incertezza, il miglior risultato atteso. In alcune situazioni non si può dimostrare l'esistenza di una azione "giusta" da fare e, tuttavia, qualcosa va fatto. Quindi bisogna agire razionalmente cioè di prendere le giuste decisioni: questa è la vera sfida dell'Intelligenza Artificiale moderna. In altri termini, la razionalità propone l'idea di "adattarsi" al contesto riuscendo ad agire nel miglior modo possibile sulla base delle informazioni disponibili.

1.2 Agenti Intelligenti

Un **agente** è un sistema che percepisce il suo ambiente attraverso dei **sensori** ed agisce su di esso tramite degli **attuatori**.



Useremo il termine **percezione** per indicare gli input percettivi dell'agente in un certo istante.

La **Sequenza percettiva** è tutto ciò che l'agente ha percepito nella sua esistenza. In generale, la scelta di un'azione da parte dell'agente può dipendere dalla sequenza percettiva osservata fino a quel momento, ma non da qualcosa che non ha mai percepito. Per ogni possibile sequenza percettiva abbiamo descritto l'agente in modo completo.

In termini matematici, il comportamento di un agente è descritto dalla sua **funzione agente**, che descrive una corrispondenza tra una sequenza percettiva ed una specifica azione. Inoltre, la funzione agente può essere espressa tramite una **tavella**. Da un punto di vista pratico, la *funzione agente* è implementata tramite un **programma agente**. In sintesi:

- il **programma agente** prende in input la percezione corrente;
- la **funzione agente** prende in input l'intera storia percettiva.

Quando un agente viene inserito in un ambiente, genera una sequenza di azioni in base alle percezioni che riceve. Questa sequenza di azioni porta l'ambiente ad attraversare una sequenza di stati: se tale sequenza è desiderabile, significa che l'agente si è comportato bene. Questa nozione di desiderabilità è catturata da una **misura di prestazione** che valuta una sequenza di stati dell'ambiente.

Per stabilire se un agente è buono o cattivo, intelligente o stupido, abbiamo 4 parametri:

- **prestazione:** cioè la capacità di compiere un obiettivo per cui l'agente è stato progettato;
- **conoscenza pregressa** che influenza il comportamento dell'agente;
- le **azioni** che l'agente può compiere;
- **sequenza percettiva** fino all'istante corrente.

Infatti, la **razionalità** di un agente dipende da questi 4 fattori. Un **agente razionale** è un agente che dovrebbe scegliere un'azione che massimizzi il valore atteso della sua **misura di prestazione** sulla base della sua sequenza percettiva.

La **misura di prestazione** deve basarsi su come l'agente opererà sull'ambiente e non su come dovrebbe comportarsi l'agente. Altrimenti, un agente otterrebbe una razionalità perfetta "illudendosi" di fare la cosa giusta.

Ricordiamo però che la razionalità non implica l'onniscienza.

Un **agente onnisciente** conosce il risultato delle sue azioni e può agire di conseguenza. Sfortunatamente, nel mondo reale l'onniscienza è impossibile.

Un **agente razionale**, invece, fa la cosa "giusta" sulla base del contesto in cui opera. Infatti, razionalità non significa perfezione ma piuttosto massimizzazione del risultato atteso.

Per massimizzare il risultato atteso, è possibile intraprendere azioni di **information gathering**, ovvero azioni che hanno lo scopo di raccogliere informazioni sull'ambiente circostante e che consentono di modificare le percezioni future.

Un esempio di *information gathering* è detto **esplorazione**, ovvero l'azione necessaria per "conoscere" un ambiente sconosciuto. L'esplorazione è importante per gli **algoritmi di ricerca**.

Oltre all'esplorazione, la razionalità prevede anche il concetto di **apprendimento**, ovvero la capacità di imparare coppie di *percezioni-azioni* sulla base delle azioni e dei risultati ottenuti in precedenza. Un agente che intraprende azioni sulla base della sola conoscenza inserita in fase di progettazione e non delle sue percezioni manca di autonomia. Il concetto di **apprendimento** è alla base degli **algoritmi di machine learning**.

1.3 Ambiente

Un **ambiente** è un'istanza di un problema di cui gli agenti razionali rappresentano le soluzioni. Un **ambiente** viene descritto tramite la formulazione **PEAS** (Performance, Environment, Actuators, Sensors).

- **Performance**: misura di prestazione usata per valutare l'operato di un agente.
- **Environment**: descrizione degli elementi che formano l'ambiente.
- **Actuators**: gli attuatori disponibili all'agente per intraprendere le azioni.
- **Sensors**: i sensori attraverso i quali l'agente riceve gli input percettivi.

Esempio:

Tipo di ambiente	Misura di prestazione	Ambiente	Attuatori	Sensori
Sistema di diagnosi medica.	Paziente sano, minimizzare i costi e le denunce.	Paziente, ospedale, staff medico.	Schermo per visualizzare domande, test, diagnosi, trattamenti.	Tastiera per l'inserimento dei sintomi, dei risultati, delle risposte del paziente.

Proprietà degli ambienti

La varietà degli ambienti operativi nell'IA è molto vasta. È comunque possibile identificare un numero relativamente piccolo di dimensioni in base a cui suddividerli in categorie. Inoltre, essere capaci di identificare le proprietà dell'ambiente ci permette di scegliere il giusto algoritmo.

- **Completemente osservabili** (o parzialmente osservabili): se i sensori di un agente gli danno accesso allo stato completo dell'ambiente in ogni momento, allora diciamo che l'ambiente operativo è completamente osservabile.
Un ambiente potrebbe essere **parzialmente osservabile** a causa di sensori inaccurati o per la presenza di rumore. Se l'agente non dispone di sensori, l'ambiente è **inosservabile**.
- **Deterministico** (o stocastico): se lo stato successivo dell'ambiente è determinato sia dallo stato corrente e che dall'azione eseguita dall'agente, allora l'ambiente è deterministico; altrimenti è **stocastico**.
- **Episodico** (o sequenziale): l'esperienza dell'agente è divisa in “*episodi*” atomici, dove ciascun episodio consiste nell'eseguire una sola azione. La scelta dell'azione dipende dall'episodio stesso. Se invece un'azione dipende da tutte quelle precedenti avremo un ambiente **sequenziale**.
- **Statico** (o dinamico): l'ambiente è invariato mentre un agente sta lavorando, altrimenti se cambia è **dinamico**. L'ambiente si dice **semi-dinamico** se esso non cambia con il passare del tempo ma il punteggio delle prestazioni dell'agente lo fa.
- **Discreto** (o continuo): l'ambiente fornisce un numero limitato di percezioni e azioni distinte e definite. La dama è un esempio di ambiente discreto, mentre l'auto a guida autonoma è un esempio di ambiente continuo.
- **Singolo** (o multi-agente): l'ambiente consente la presenza di un unico agente. Se ci sono più agenti, gli scenari cambiano, come negli algoritmi di ricerca e in particolare nella teoria dei giochi.

A questo proposito, vale la pena fornire una seconda classificazione:

- **Competitivo**. L'ambiente è competitivo se, in un ambiente *multi-agente*, il comportamento di un agente A massimizza/minimizza la misura di prestazione del comportamento di B (e viceversa).
- **Cooperativo**. L'ambiente è cooperativo se, in un ambiente *multi-agente*, due agenti A e B puntano a massimizzare/minimizzare la stessa misura di prestazione.

Esempio:

Tipo di ambiente	Osservabile	Agenti	Deterministico	Episodico	Statico	Discreto
Cruciverba	Completemente	Singolo	Deterministico	Sequenziale	Statico	Discreto
Scacchi con clock	Completemente	Multi	Deterministico	Sequenziale	Semi	Discreto

1.4 Tipologie di Agente

Il compito dell'intelligenza artificiale è di progettare il **programma agente** attraverso l'implementazione della **funziona agente**. Per progettare un **programma agente** dobbiamo, innanzitutto, definire l'agente come l'unione dell'architettura, cioè un computer dotato di sensori e attuatori, con il programma (*agente = architettura + programma*). I *programmi agente* prenderanno in input la percezione corrente dei sensori e restituiranno come output un'azione agli attuatori.

È importante notare la differenza tra **programma** e **funzione agente**:

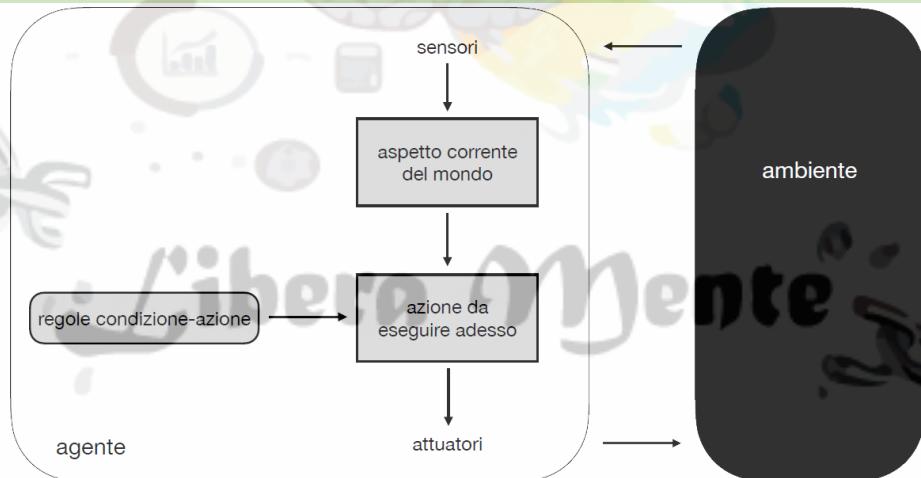
- il primo prende in input solo la percezione corrente;
- la seconda l'intera storia percettiva.

In generale, la sfida principale dell'intelligenza artificiale è trovare il modo per scrivere programmi che producano un comportamento razionale con una piccola quantità di codice invece che con la rappresentazione di tutti gli input percettivi possibili. Quindi, usare le tabelle per esprimere una funzione agente è svantaggioso.

Tutti gli agenti che vedremo sono classificati come **learning agents**, ovvero agenti capaci di migliorare le loro prestazioni ed eseguire azioni migliori attraverso l'apprendimento.

➤ Agenti Reattivi Semplici

Il tipo più semplice è l'**agente reattivo semplice**. Questi agenti eseguono le azioni sulla base della percezione corrente, ignorando la storia percettiva pregressa. Pertanto, non hanno memoria.



Il vantaggio degli **agenti reattivi semplici** riguarda la rappresentazione delle percezioni perché in questo caso non dovremmo memorizzare nulla.

Un *agente reattivo semplice* fa uso delle regole **condizione-azione** per decidere quale sarà la prossima azione da compiere. Le regole sono banalmente espresse come degli **if-then**.

Esempio: **if** la-macchina-davanti-frena **then** inizia-a-frenare.

Un *agente reattivo semplice* funziona bene solo se l'ambiente è **completamente osservabile** altrimenti non potremmo specificare tutte le regole condizione-azione per cui avremmo dei **dati mancanti**. Quindi, in casi in cui un agente reattivo semplice opera in **ambienti parzialmente osservabili** (quindi con dati mancanti) l'agente si troverà in **loop infiniti**.

Una soluzione a questo problema è quella di definire una **componente casuale**, la quale verrà invocata nel caso di dati mancanti per poter compiere un'azione casuale e, quindi evitare cicli infiniti.

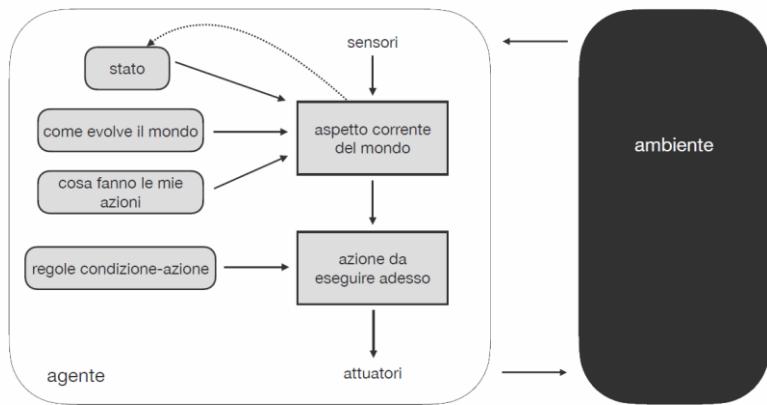
Bisogna fare attenzione perché una *componente casuale* porta solitamente ad un comportamento razionale, sfortunatamente però non è questo il caso, infatti, la *componente casuale* serve solo ad un *agente reattivo semplice* a far evitare cicli infiniti, ma non "crea" intelligenza.

➤ Agenti Reattivi basati su Modello

Un **agente reattivo basato su modello** è un'agente con due tipi di conoscenza:

1. come evolve il mondo, indipendentemente dal suo stato;
2. informazioni sull'impatto delle sue azioni sull'ambiente.

In base a questi due tipi di conoscenza, tale tipo di agente opererà sempre tenendo in considerazione le regole *condizione-azione*.



Questa conoscenza sul “*funzionamento del mondo*”, implementata attraverso circuiti logici, viene chiamata **modello del mondo**. Un agente che si appoggia a un simile modello prende il nome di **agente basato su modello**.

```
function AGENTE-REATTIVO-BASATO-SU-MODELLO(percezione) returns un'azione
    persistent: stato, la concezione corrente dello stato del mondo da parte dell'agente;
                modello, una descrizione della dipendenza dello stato successivo dallo stato
                corrente e dall'azione;
                regole, un insieme di regole condizione-azione;
                azione, l'azione più recente, inizialmente nessuna;

    stato <- AGGIORNA-STATO(stato, azione, percezione, modello)
    regola <- REGOLA-CORRISPONDENTE(stato, regole)
    azione <- regola.AZIONE

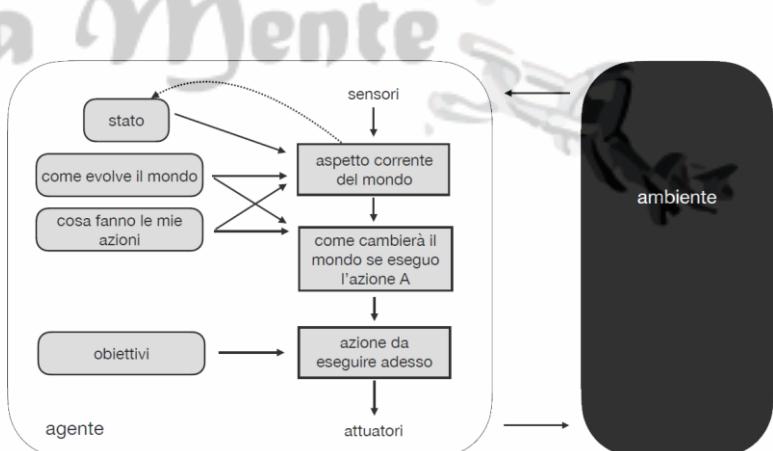
    return azione
```

La parte interessante è la funzione **AGGIORNA-STATO()**, responsabile della creazione del nuovo stato interno.

➤ Agenti basati su Obiettivi

L'**agente basato su obiettivo**, oltre che della descrizione dello stato corrente, ha bisogno di qualche tipo di informazione riguardante il suo obiettivo (goal), che descriva situazioni desiderabili.

Il *programma agente* può unire quest'informazione al modello per scegliere le azioni che portano al soddisfacimento dell'obiettivo.



Talvolta, scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere raggiunto in un solo passo. Altre volte è più difficile, ad esempio quando l'agente deve considerare lunghe sequenze di azioni per trovare il “cammino” che porta al risultato desiderato.

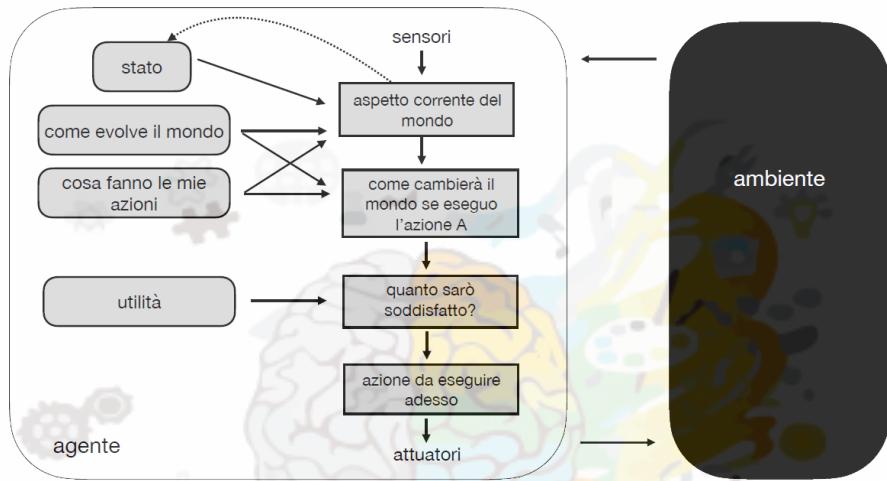
Bisogna notare che questo tipo di decisioni non ha nulla a che vedere con le regole *condizione – azione*, perché ora bisogna prendere in considerazione il futuro sotto due aspetti: “cosa accadrà se faccio così o così?” e anche “se faccio questo, sarò soddisfatto?”.

Ovvero, le decisioni prese dall'agente non sono necessariamente deterministiche e basate su esplicite regole ma dipendono da come cambierà l'ambiente in funzione delle azioni e degli obiettivi da raggiungere.

➤ Agenti basati sull'Utilità

Nella maggior parte degli ambienti, gli obiettivi da soli non bastano a generare un comportamento di alta qualità poiché i forniscono solamente una distinzione binaria tra stati “contenti” e “scontenti”; in situazioni reali, esistono situazioni “desiderabili” e non.

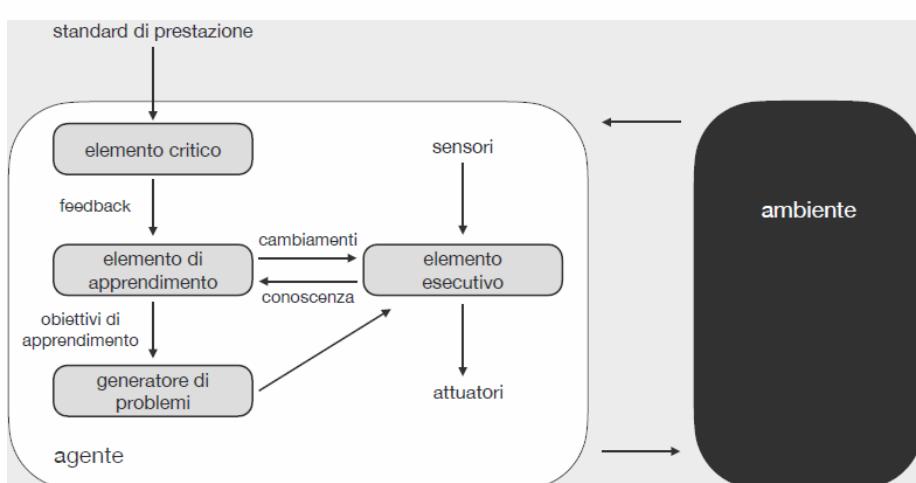
Una **funzione di utilità** di un agente assegna ad uno stato un numero reale che quantifica il grado di “desiderabilità” ad esso associato: quando ci sono più obiettivi tra di loro contrastanti, una funzione di utilità consente di determinare quale obiettivo preferire in base al punteggio ad esso assegnato dalla *funzione utilità*.



➤ Agenti capaci di apprendere

L'**apprendimento** ha il vantaggio di permettere agli agenti di operare in ambienti inizialmente sconosciuti diventando col tempo più competenti. Un **agente capace di apprendere** può essere diviso in quattro componenti astratti:

1. **Elemento di apprendimento**. L'elemento responsabile del miglioramento interno.
2. **Elemento esecutivo**. L'elemento responsabile della selezione delle azioni esterne. Questo elemento è l'agente: prende in input le percezioni e decide le azioni.
3. **Elemento critico**. L'elemento responsabile di fornire feedback sulle prestazioni correnti dell'agente, così che l'*elemento di apprendimento* possa determinare se e come modificare l'elemento esecutivo affinché si comporti meglio in futuro.
4. **Generatore di problemi**. L'elemento responsabile di suggerire azioni che portino ad esperienze nuove e che portino l'agente ad apprendere nuove conoscenze da sfruttare poi per migliorare le sue azioni.





ALGORITMI DI RICERCA

Capitolo 2 – Formulazione di problemi

2.1 Agenti risolutori di problemi

Gli **algoritmi di ricerca** lavorano con rappresentazioni atomiche cioè hanno un livello di astrazione maggiore perché non considerano la struttura interna di uno stato

In questo ambito parleremo degli **agenti risolutori di problemi**, i quali si pongono uno o più obiettivi da raggiungere ed eseguono azioni che massimizzano le chance del loro raggiungimento. Utilizzano, pertanto, **rappresentazioni atomiche del mondo**, i cui stati verranno usati per trovare una soluzione accettabile al problema trattato.

Esempio:

Supponiamo che un agente X sia in vacanza ad Arad, in Romania. La sua **misura di prestazione** sarà composta da diversi fattori: migliorare il suo rumeno, ammirare i panorami più belli, passare notti divertenti e così via. di partire da Arad e dover andare a Bucarest.

In un contesto del genere, l'obiettivo dell'agente risulta essere complesso in quanto dovrà trovare una serie di compromessi che bilancino i vari obiettivi.

Supponiamo adesso che l'agente abbia un biglietto aereo non rimborsabile per la partenza da Bucarest il giorno successivo. Tutte le azioni che non consentiranno all'agente di raggiungere Bucarest in tempo possono essere scartate. In termini più pratici, questo implica che lo **spazio delle soluzioni** potrà essere sostanzialmente ridotto, velocizzando il raggiungimento di una soluzione.

La **formulazione del problema** è il processo che porta a decidere, dato un obiettivo, quali **azioni** e quali **stati** considerare.:

- **Stati**: ogni stato corrisponderà a trovarsi in una particolare città.
- **Azioni**: gli spostamenti da una città all'altra.

L'**obiettivo** è rappresentato da un insieme ammissibile di stati del mondo. Più in dettaglio, è rappresentato da tutti e soli quegli stati in cui l'obiettivo è soddisfatto.

Ovviamente, gli spostamenti da una città all'altra comportano un cambiamento di stato, perché se ci troviamo ad Arad vuol dire che ci troviamo nello stato in cui siamo ad Arad.

Compire un'azione ci fa cambiare stato, perché dopo il compimento di una determinata azione, ci troveremo nello stato in cui saremo in un'altra città.

La soluzione di questo problema è trovare un percorso che ci porti a Bucharest; quindi, l'agente dovrà scegliere una **giusta granularità**, ovvero un certo livello di astrazione per giungere alla soluzione.

Inoltre, un agente che ha a disposizione diverse opzioni immediate di **valore sconosciuto** può decidere cosa fare esaminando le azioni future che alla fine porteranno a stati di **valore conosciuto**.

Pertanto, la soluzione per noi sarà la sequenza di azioni che ci porta dallo stato in cui siamo ad Arad allo stato in cui saremo a Bucharest.

Questo tipo di problema è **deterministico e completamente osservabile**, e inoltre è a **singolo stato**. Infatti, l'agente sa esattamente in quale stato si trova, perché conosce la sequenza di stati.

2.2 Algoritmo di ricerca

La soluzione di qualsiasi problema è rappresentata da una **sequenza fissata di azioni**. In altri termini, una soluzione potrebbe essere implementata come una strategia ramificata che raccomandi azioni future diverse in base alle percezioni avute. Il processo che cerca una sequenza di azioni che raggiunge l'obiettivo è detto **ricerca**.

La progettazione dell'agente consiste quindi di tre fasi: **Formulazione, Ricerca, Esecuzione**.



La funzione prende in input la percezione corrente dell'agente.

```

function SEMPLICE-AGENTE-RISOLTORE-PROBLEMI(percezione)
    persistent: seq, una sequenza di azioni, inizialmente vuota;
                stato, una descrizione dello stato corrente del mondo;
                goal, un obiettivo, inizialmente null;
                problema, una formulazione di problema;

    stato <- AGGIORNA-STATO (stato, percezione)
    if seq è vuota then
        goal <- FORMULA-OBIETTIVO(stato)
        problema <- FORMULA-PROBLEMA(stato, goal)
        seq <- RICERCA(problema)
        if seq = fallimento then return un'azione nulla
    while seq non è vuota
        azione <- PRIMO-ELEMENTO(seq)
        seq <- CODA(seq)
    
```

- /* Nella prima fase, le variabili necessarie alla risoluzione del problema sono instanziate */
- /* Aggiornato lo stato, dato lo stato precedente e la percezione attuale, l'agente formerà un obiettivo e un problema. Dopodiché, avvierà l'algoritmo di ricerca */
- /* Se l'algoritmo fallisce, l'agente non compirà alcuna azione */
- /* Altrimenti, l'agente inizierà ad eseguire le azioni raccomandate, una alla volta, fino alla fine della sequenza */

Mentre l'agente esegue la sequenza di azioni ignora le proprie percezioni, perché le conosce in anticipo. Un agente di questo tipo, che porta avanti le proprie azioni ad “occhi chiusi”, deve essere certo di quello che accade.

Nella teoria del controllo, si parla di **sistema a ciclo aperto**, perché ignorando le percezioni si rompe il ciclo tra agente e ambiente.

In quest'altro pseudocodice *dell'agente semplice risolutore di problemi*, una volta eseguita la prima azione suggerita dall'algoritmo di ricerca, questa verrebbe restituita.

Una volta richiamata la funzione, dopo aver eseguito la prima azione, la percezione cambierebbe e di conseguenza anche obiettivo, problema e ricerca sarebbero modificati. Così facendo:

- **non ci troveremmo più in un ciclo aperto;**
- **l'agente ricercherebbe una nuova sequenza di azioni ad ogni stato.**

```

function SEMPLICE-AGENTE-RISOLTORE-PROBLEMI(percezione) returns un'azione
    persistent: seq, una sequenza di azioni, inizialmente vuota;
                stato, una descrizione dello stato corrente del mondo;
                goal, un obiettivo, inizialmente null;
                problema, una formulazione di problema;

    stato <- AGGIORNA-STATO (stato, percezione)
    if seq è vuota then
        goal <- FORMULA-OBIETTIVO(stato)
        problema <- FORMULA-PROBLEMA(stato, goal)
        seq <- RICERCA(problema)
        if seq = fallimento then return un'azione nulla
    while seq non è vuota
        azione <- PRIMO-ELEMENTO(seq)
        seq <- CODA(seq)
    
```

```

return azione
    
```

2.3 Problemi ben definiti e soluzioni

Per formulare i **problemi di ricerca**, abbiamo bisogno di cinque componenti:

1. **Stato iniziale dell'agente**: rappresenta il punto di partenza dell'agente. Nell'esempio precedente, lo stato iniziale potrebbe essere descritto come **in (Arad)**.
2. **Azioni**: l'insieme Azioni include la descrizione delle possibili operazioni attuabili dall'agente. Più formalmente, dato uno stato s , **Azioni (s)** restituisce l'insieme di azioni che possono essere eseguite in s , anche dette **azioni applicabili** in s .
Nell'esempio precedente, le azioni possibili sono {Go (Sibiu), Go (Timisoara), Go (Zerind)}.
3. **Modello di transizione**: descrive il risultato di ogni azione attuabile dall'agente in s . È specificato da una funzione **Risultato (s, a)** che restituisce lo stato risultante dall'esecuzione dell'azione a nello stato s . Utilizziamo anche il termine **successore** per indicare qualsiasi stato raggiungibile da uno stato mediante una singola azione.
Ad esempio, potremmo avere che: Risultato (in (Arad), Go (Zerind)) = In (Zerid).
4. **Test obiettivo**: determina se un particolare stato è uno stato obiettivo. Può essere implicito (quando lo stato obiettivo è solo uno) o esplicito (elenco tutti gli stati obiettivo).
Nell'esempio, il test obiettivo sarà dato dal singleton {in (Bucarest)}.
5. **Costo di cammino**: funzione che determina il costo numerico a ogni cammino. Nell'esempio, il costo potrebbe essere rappresentato dai chilometri da percorrere per arrivare a Bucarest.

Stato iniziale, azioni, e modello di transizione definiscono lo **stato degli spazi** del problema, cioè l'insieme di tutti gli stati raggiungibili da quello iniziale mediante qualsiasi sequenza di azioni.

Questo può essere rappresentato sotto forma di **grafo**, in cui i **nodi** rappresentano gli **stati** e gli **archi** (pesati) rappresentano l'**azione** che porta da un nodo ad un altro.

L'azione possiede un costo di passo $c(s, a, s')$, ovvero il costo dell'azione a che porta dallo stato s a quello s' .

2.4 Formulazione di problemi

Quindi, ora sappiamo come è definito un problema, ovvero abbiamo descritto le sue 5 componenti. Ora vediamo come formularlo in maniera efficace.



Per formulare un problema dobbiamo rimuovere tutti quei dettagli che sono irrilevanti al fine di raggiungere il nostro obiettivo. Il processo di rimozione dei dettagli da una rappresentazione è detta **astrazione**.

Oltre ad astrarre le **descrizioni di stato**, dobbiamo astrarre anche le **azioni**. In altri termini, possiamo pensare ad uno stato non come un elemento individuale, ma come un insieme di **stati dettagliati**; allo stesso modo, un'azione è in realtà un insieme di **azioni dettagliate**.

- Un'astrazione si definisce **valida** se possiamo espandere ogni soluzione astratta in una soluzione nel mondo più dettagliata.
- Un'astrazione si definisce **utile** se eseguire ogni azione nella soluzione è più facile che nel problema originale.

La scelta di una buona astrazione (o granularità dell'astrazione) prevede che si rimuova quanto più dettaglio possibile mantenendo la validità e assicurandosi che le azioni astratte siano più facile da eseguire.

Senza la capacità di astrazione, gli agenti intelligenti sarebbero completamente sopraffatti dalla complessità del mondo reale.



Capitolo 3 – Algoritmi di Ricerca Non Informata

3.1 Algoritmi di ricerca ad Albero

Come primo step abbiamo visto la formulazione di un problema, ora vedremo come poter ricercare una soluzione.

Una soluzione è una **sequenza di azioni**, per questo, gli **algoritmi di ricerca** operano varie possibili sequenze.

Tali sequenze di azioni formano un **albero di ricerca**:

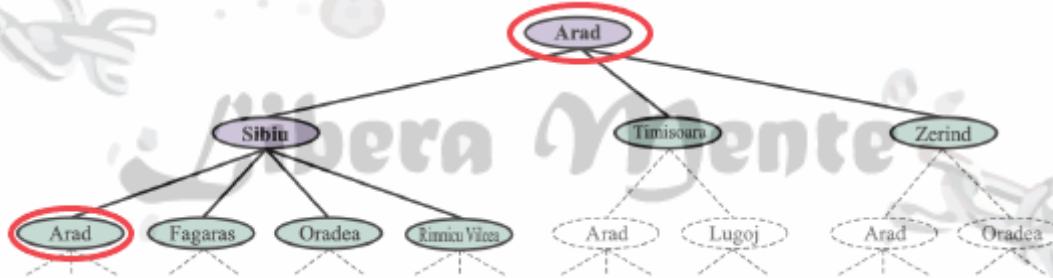
- lo stato iniziale (che coincide con la radice dell'albero) rappresenta il problema;
- i **rami** sono le azioni;
- i **nodi** corrispondono agli stati che il problema può avere.

Dato lo stato iniziale, il primo passo sarà verificare che l'obiettivo non sia stato raggiunto; se non sarà così, si considereranno le varie azioni, espandendo lo stato corrente e generando così un nuovo insieme di stati. Questa è l'essenza della ricerca: cioè **approfondire** un'opzione e mettere da parte le altre, per poi riprenderle nel caso la prima non porti ad una soluzione.

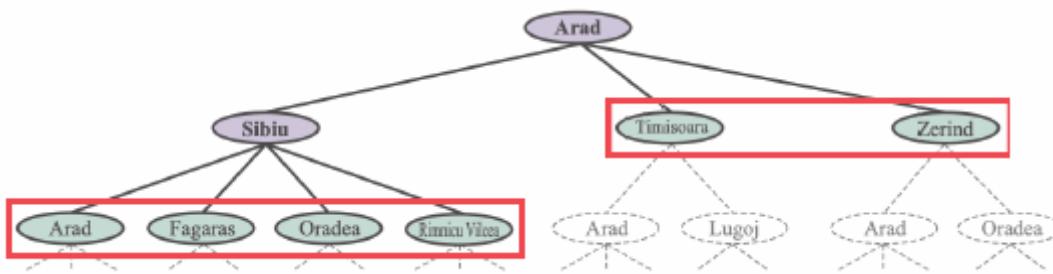
Nell'*albero di ricerca* uno stato può essere ripetuto più volte (es. Arad): in questo caso parliamo di **cammino ciclico**.

In alcuni casi, i cicli possono causare il fallimento di alcuni algoritmi, poiché un ciclo può essere percorso all'infinito.

Tuttavia, i cammini sono **additivi**, cioè tutti i cammini portano ad incrementare un costo che non è negativo. Quindi, non conviene mai intraprendere un *cammino ciclico*.



I nodi terminali di un albero sono detti **nodi foglia**, ovvero nodi privi di figli in un determinato momento della ricerca. L'insieme di tutti i nodi foglia che possono essere espansi in un dato punto è detto **frontiera**.



Nell'esempio solo **Arad** è colorato, questo perché in quello specifico momento è stato considerato solo lo stato iniziale; nel momento in cui, si è verificato che **Arad** non è l'obiettivo della **ricerca**, l'**algoritmo di ricerca** andrà a considerare le tre alternative: **Sibiu, Timisoara e Zerind**.

Pseudocodice di una ricerca nell'albero

L'algoritmo prende in considerazione anche una **strategia** per poter espandere una **frontiera**. La strategia definisce la politica che l'algoritmo utilizzerà per decidere come procedere con la ricerca.

```
function RICERCA-ALBERO(problema, strategia) returns una soluzione o un fallimento
    inizializza la frontiera usando lo stato iniziale di problema

    loop do
        if la frontiera è vuota then return fallimento
        scegli un nodo foglia in base a strategia e rimuovilo dalla frontiera

        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        espandi il nodo scelto, aggiungendo i nodi risultati alla frontiera
```

3.2 Cammini ciclici e cammini ridondanti

I **cammini ciclici** sono un caso particolare dei **cammini ridondanti**.

Un **cammino ridondante** è un cammino in cui esistono due o più modi per passare da uno stato all'altro. I **cammini ridondanti**, inoltre, possono causare il fallimento degli *algoritmi di ricerca* portando alla non risoluzione del problema. In alcuni casi, tuttavia, non è possibile evitare i **cammini ridondanti**.

Per evitare di addentrarsi in *cammini ridondanti*, è necessario ricordare dove si è passati. Possiamo, quindi, migliorare l'algoritmo di ricerca introducendo una struttura dati chiamata **insieme esplorato** (o lista chiusa) in cui terremo traccia dei nodi esplorati.

```
function RICERCA-GRAFO(problema, strategia) returns una soluzione o un fallimento
    inizializza la frontiera usando lo stato iniziale di problema
    inizializza a vuoto l'insieme esplorato

    loop do
        if la frontiera è vuota then return fallimento
        scegli un nodo foglia in base a strategia e rimuovilo dalla frontiera

        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        aggiungi il nodo all'insieme esplorato
        espandi il nodo scelto, aggiungendo i nodi risultati alla frontiera solo se non è nella
                    frontiera o nell'insieme esplorato
```

L'albero di ricerca costruito con questo algoritmo contiene al più una copia di ciascuno stato; perciò, possiamo pensare che faccia crescere un albero come un **grafo**.

L'algoritmo ha un'altra proprietà: la frontiera separa **il grafo degli stati nella regione esplorata** e in quella **inesplorata**, in modo che ogni cammino che vado dallo stato iniziale ad uno stato inesplorato deve passare attraverso uno stato della frontiera.

3.3 Struttura dati per algoritmi di ricerca

Gli *algoritmi di ricerca* richiedono una **struttura dati** per la costruzione dell'albero di ricerca. Per ogni nodo n dell'albero abbiamo una struttura che contiene i seguenti 4 componenti:

1. $n.\text{stato}$: lo stato dello spazio degli stati a cui corrisponde il nodo;
2. $n.\text{padre}$: il nodo dell'albero di ricerca che ha generato il nodo corrente;
3. $n.\text{azione}$: l'azione applicata al padre per generare il nodo;
4. $n.\text{costo-cammino}$: il costo $g(n)$ del cammino che va dallo stato iniziale al nodo;

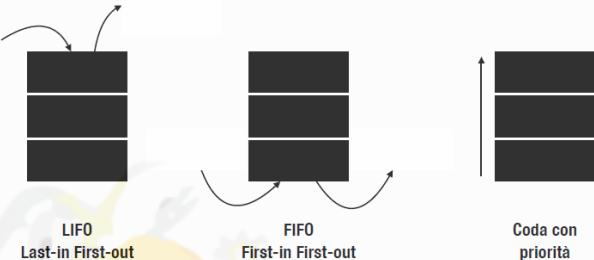
Date le componenti di nodo padre, è facile calcolare le componenti necessari per generare un nodo figlio:

```
function NODO-FIGLIO(problema, padre, azione) returns un nodo
    returns un nodo con:
        STATO = problema.RISULTATO(padre.STATO, azione)
        PADRE = padre, AZIONE = azione
        COSTO-DI-CAMMINO = padre.COSTO-DI-CAMMINO + problema.COSTO-DI-PASSO(padre.STATO, azione, STATO)
```

Per memorizzare, invece, tutti i nodi generati dall'albero di ricerca sarà necessaria una **struttura dati**.

La frontiera deve essere memorizzata in modo da consentire all'algoritmo di ricerca di scegliere facilmente il nodo successivo da espandere in base alla propria strategia.

Quindi, la struttura dati più adatta è una **coda**.



3.4 Valutare un algoritmo di ricerca

Una strategia di ricerca è scelta secondo una **modalità di espansione dei nodi**, ovvero scegliendo l'ordine in cui i nodi sono espansi. Tali strategie vengono valutate in base a 4 indicatori:

1. **Completezza**: se l'algoritmo garantisce di trovare una soluzione;
2. **Ottimalità**: se l'algoritmo garantisce di trovare la soluzione ottima;
3. **Complessità temporale**: quanto tempo impiega l'algoritmo per trovare una soluzione;
4. **Complessità spaziale**: quanta memoria ha bisogno l'algoritmo per trovare una soluzione.

A differenza dell'informatica teorica, dove la complessità spaziale e temporale sono tipicamente misurate in termini di dimensione del grafo (ovvero, in termini di $|V|$ e $|E|$), in **Intelligenza Artificiale** il grafo è rappresentato dallo **stato iniziale**, dalle **azioni** e dal **modello di transizione**: per questa ragione, è spesso **indefinito**.

Ne consegue che la complessità temporale e quella spaziale sono misurate in termini di:

- **b** (branching factor): massimo fattore di ramificazione dell'albero di ricerca;
- **d**: profondità della soluzione a costo minore;
- **m**: massima profondità dello spazio degli stati (può essere ∞).

3.5 Strategie di ricerca non informata

Gli **algoritmi di ricerca non informata** usano strategie di ricerca che non dispongono di informazioni aggiuntive sugli stati oltre a quella fornita nella definizione del problema: tutto ciò che conoscono è dove si trovano, qual è l'obiettivo e cosa possono fare quando si trovano in un certo stato.

La differenza tra le varie strategie di ricerca consiste nell'ordine in cui vengono espansi i nodi.

Gli **algoritmi di ricerca non informata** non sanno stimare quanto un nodo non obiettivo sia "promettente" per la risoluzione del problema, a differenza degli **algoritmi di ricerca informata** ed **euristica**, i quali fanno uso di informazioni riguardante la distanza stimata dalla soluzione.

Ricerca in ampiezza

La **ricerca in ampiezza** è una semplice strategia che consiste di:

1. Espandere il nodo radice;
2. Espandere i nodi generati dalla radice;
3. Espandere i loro successori e così via.

La **ricerca in ampiezza** è una strategia di ricerca in cui tutti i nodi di profondità d sono espansi prima di quelli di profondità $d + 1$.

La ricerca in ampiezza è una strategia **sistematica** di ricerca, ovvero è in grado di trovare sempre una soluzione, se esiste; nel caso pessimo andremo ad espandere tutti i nodi. Inoltre, trova sempre la soluzione con il **cammino più breve** il quale non è necessariamente quello con il costo minore.

La *ricerca in ampiezza* è un'istanza dell'algoritmo di ricerca su grafo in cui viene scelto per l'espansione il nodo non espanso più vicino alla radice.

Da un punto di vista pratico, questa strategia può essere implementata utilizzando una semplice **coda FIFO** per la *frontiera*. Di conseguenza, i nuovi nodi (che sono quelli via via più lontani dalla radice) vanno in fondo alla coda e i nodi vecchi (che sono quelli più vicini) vengono espansi per primi.

```
function RICERCA-IN-AMPIEZZA(problema, strategia)
    returns una soluzione o un fallimento

    nodo <- un nodo con stato = problema.STATO-INIZIALE e COSTO-DI-CAMMINO=0
    if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)

    frontiera <- una coda FIFO con nodo come unico elemento
    esplorati <- un insieme vuoto

    loop do
        if la frontiera è vuota then return fallimento
        nodo <- POP(frontiera)
        aggiungi nodo.STATO a esplorati

        for each azione in problema.AZIONI(nodo.STATO) do
            figlio <- NODO-FIGLIO(problema, stato, azione)
            if figlio.STATO non è in esplorati e figlio non è in frontiera then
                if problema.TEST-OBIETTIVO(figlio.STATO) then return SOLUZIONE(figlio)
                frontiera <- INSERISCI(figlio, frontiera)
```

- All'inizio assegniamo la radice del grafo alla variabile nodo e controlliamo se la soluzione è la radice.
- Se la soluzione non è nella radice, allora inizializziamo la frontiera (con la radice, che è l'unico nodo esplorato finora) e la lista dei nodi esplorati.
- Se durante la ricerca la frontiera dovesse essere vuota e la soluzione non dovesse essere trovata allora viene restituito un fallimento.
- Altrimenti, viene chiamata la funzione `POP` di una coda che estrae il primo elemento della frontiera; il nuovo nodo verrà aggiunto a quelli esplorati.
- La ricerca va avanti: dal nodo considerato si procede ad esplorare i figli (solo se non già esplorati) e valutare se questi rappresentano delle soluzioni.

Prestazioni – Ricerca in ampiezza

Analizziamo adesso le **prestazioni** della ricerca in ampiezza, considerando i 4 indicatori. Denotiamo con:

- **b** il fattore di **diramazione** (ovvero, il numero massimo di successori);
 - **d** la **profondità** della soluzione minima.
- **Completezza:** sì, perché visitiamo comunque tutto l'albero, però il fattore di **diramazione (b)** deve essere finito.
 - **Ottimalità:** se suppongo un costo di cammino unitario, ho che la prima soluzione trovata è la migliore, in particolare se il costo di cammino è una funzione monotona non decrescente. Se il costo di cammino non è unitario e costante, allora non sono sicuro che la prima soluzione trovata sia la migliore.
 - **Complessità temporale:** supponiamo di dover ricercare una soluzione in un albero in cui ogni nodo ha **b successori**. La radice genererà **b** nodi al primo livello, ognuno dei quali genererà altri **b** nodi, per un totale di b^2 al secondo livello, arrivando a b^3 al terzo livello, e così via. Nel caso pessimo, la soluzione si troverà ad una profondità **d**.
Quindi: $b + b^2 + b^3 + \dots + b^d = O(b^d)$ avremo una **complessità esponenziale**.
 - **Complessità spaziale:** l'algoritmo memorizza ogni nodo espanso nell'insieme dei nodi esplorati: per questa ragione, la complessità sarà sempre una funzione di **b**.
In particolare, la **ricerca in ampiezza conserva ogni nodo generato in memoria**: avremo così $O(b^{d-1})$ nodi **nell'insieme esplorato** e $O(b^d)$ nodi nella **frontiera**.
Quindi, la **complessità spaziale** sarà di $O(b^d)$, uguale alla *complessità temporale*.

Ricerca a costo uniforme

La **ricerca a costo uniforme** espande il nodo **n** con il costo di cammino minimo $g(n)$ memorizzando la **frontiera** in una **coda a priorità ordinata secondo g** (in cima alla coda ci saranno i nodi di costo minore).

Oltre all'ordinamento della coda, ci sono due differenze implementative rispetto alla *ricerca per ampiezza*:

1. il test obiettivo è applicato ad un nodo non appena è selezionato per l'espansione e non quando è generato per la prima volta.
2. si aggiunge un test obiettivo nel caso in cui sia trovato un cammino migliore per raggiungere un nodo che attualmente si trova sulla frontiera.

```
function RICERCA-IN-AMPIEZZA RICERCA-COSTO-UNIFORME(problema, strategia)
    returns una soluzione o un fallimento

    nodo <- un nodo con stato = problema.STATO-INIZIALE e COSTO-DI-CAMMINO=0
    frontiera <- una coda a priorità ordinata per COSTO-CAMMINO, con nodo unico elemento
    esplorati <- un insieme vuoto

    if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)

    frontiera <- una coda FIFO con nodo come unico elemento
    esplorati <- un insieme vuoto

    loop do

        if la frontiera è vuota then return fallimento
        nodo <- POP(frontiera)
        if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
        aggiungi nodo.STATO a esplorati

        for each azione in problema.AZIONI(nodo.STATO) do
            figlio <- NODO-FIGLIO(problema, stato, azione)
            if figlio.STATO non è in esplorati e figlio non è in frontiera then
                if problema.TEST-OBIETTIVO(figlio.STATO) then return SOLUZIONE(figlio)
                frontiera <- INSERISCI(figlio, frontiera)
            else if figlio.STATO è in frontiera con COSTO-CAMMINO più alto then
                sostituisce quel nodo frontiera con figlio
```

Prestazioni – Ricerca a costo uniforme

- **Completezza e ottimalità:** l'algoritmo è completo (a meno di operazioni NoOp) e anche ottimo.
- **Complessità temporale e spaziale:** la *ricerca a costo uniforme* è guidata dal costo del **cammino** e non dalla loro profondità, per questo consideriamo $C^* =$ il costo della soluzione ottima e assumiamo che ogni azione costi almeno ϵ .
La complessità temporale e spaziale sarà uguale a $O(b^{1+|C^*/\epsilon|-1})$.

La spiegazione è molto semplice: C^*/ϵ rappresenta il costo di ogni passo fatto dall'algoritmo; la complessità è dominata dal numero di passi effettuati, con l'aggravante che non si arresta una volta raggiunto l'obiettivo, ma esamina tutti i nodi al livello di profondità dell'obiettivo per verificare l'esistenza di una soluzione con costo minore.

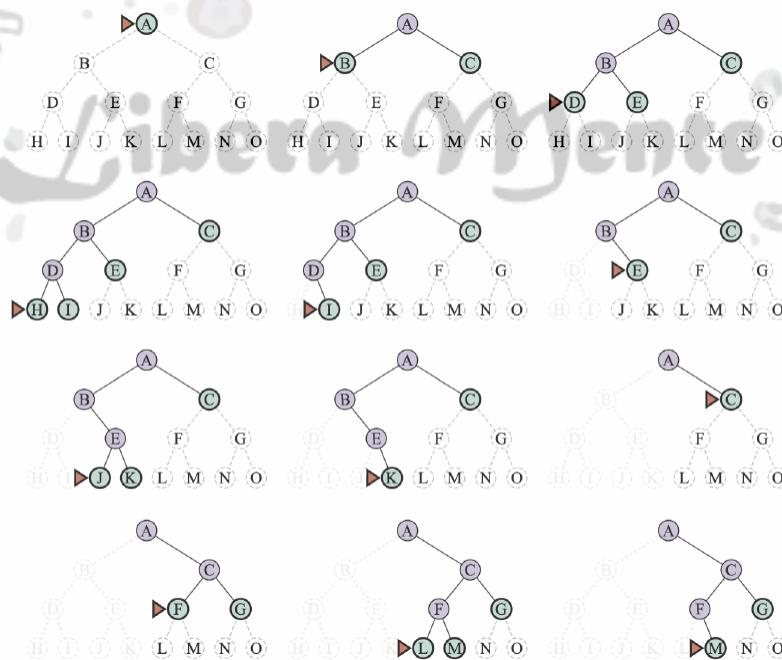
Quindi, la **ricerca a costo uniforme** è, in generale, più efficace di quella in *ampiezza* ma potrebbe essere addirittura meno efficiente.

Ricerca in profondità

Questo tipo di ricerca è l'opposto della *ricerca in ampiezza*; infatti, viene sempre espanso prima il nodo più profondo nella frontiera corrente dell'albero di ricerca.

Da un punto di vista pratico, la **ricerca in profondità** raggiunge il livello più profondo dell'albero, dove i nodi non hanno successori. L'espansione di tali nodi li rimuove dalla frontiera, per cui la ricerca "torna indietro" (**backtracking**) per riconsiderare il nodo più profondo che ha successori non ancora espansi.

Questo tipo di ricerca fa uso di una **coda LIFO**, poiché verrà sempre espanso l'ultimo nodo generato.



In casi in cui l'albero di ricerca ha uno spazio degli stati con profondità infinita o con cicli, l'algoritmo non terminerà. Quindi, possiamo sostenere che l'algoritmo di ricerca in profondità **non è completo**.

Allo stesso tempo, se abbiamo due nodi obiettivo e il primo si trova nel primo ramo l'algoritmo lo restituisce a prescindere dalla soluzione. Quindi, l'algoritmo **non è ottimo**.

Sebbene questo algoritmo abbia dei vantaggi in termini di **complessità spaziale** rispetto agli algoritmi precedenti, la **complessità temporale** resta **esponenziale**.

Possiamo però migliorare l'algoritmo imponendo un limite massimo alla profondità dei cammini.

Ricerca in profondità limitata

Opera in maniera simile alla strategia di *ricerca in profondità* ma impone un limite di profondità per evitare che l'algoritmo non termini.

Con questa strategia, un nodo viene espanso solo se la lunghezza del cammino corrispondente è minore rispetto al massimo stabilito.

Da un lato, l'algoritmo evita la non terminazione; dall'altro, però, introduce una nuova fonte di incompletezza nel caso in cui la soluzione si trovi ad una profondità maggiore di quella stabilita.

Rispetto ai precedenti algoritmi, questo tipo di ricerca può terminare in due modi diversi:

1. per via di un fallimento (ovvero, nessuna soluzione);
2. tramite il valore del **taglio** (ovvero, nessuna soluzione entro il limite stabilito).

Il problema di questa strategia consiste nel trovare la soglia di taglio ideale. Questa soglia è molto difficile da rilevare e deriva dall'esperienza del designer o da dati empirici.

```
function RICERCA-PROFONDITÀ-LIMITATA(problema, strategia)
    returns una soluzione o il fallimento/taglio

returns RPL-RICORSIVA(CREA-NODO(problema.STATO-INIZIALE), problema, limite)

function RPL-RICORSIVA(nodo, problema, limite)
    returns una soluzione o il fallimento/taglio

    if problema.TEST-OBIETTIVO(nodo.STATO) then returns SOLUZIONE(nodo)
    else if limite = 0 then return taglio

    else
        avvenuto_taglio <- false
        for each azione in problema.AZIONI(nodo.STATO) do
            figlio <- NODO-FIGLIO(problema, nodo, azione)
            risultato <- RPL-RICORSIVA(figlio, problema, limite-1)
            if risultato = taglio then avvenuto_taglio <- true
            else if risultato != fallimento then return risultato

        if avvenuto_taglio then return taglio
        else return fallimento
```

Pseudocodice

La prima funziona richama la seconda funzione RPL-Ricorsiva. Questa seconda funziona avrà una variabile `limite` che servirà a terminare la ricerca ed evitare casi di incompletezza.

Il processo di ricerca è identico all'algoritmo generale di ricerca, con l'eccezione che la funzione sarà ricorsiva.

Performance – Ricerca in profondità limitata

- **Completezza:** l'algoritmo non è completo nei casi in cui la soluzione si trova ad una profondità maggiore del limite stabilito.
- **Complessità temporale:** la complessità è limitata dalla dimensione dello spazio degli stati. Sia k il limite prestabilito, nel caso pessimo la ricerca genererà $O(b^k)$ nodi.
- **Ottimalità:** non può essere ottimo nei casi in cui il limite stabilito sia minore della profondità massima (quindi sempre, altrimenti non sarebbe limitata).
- **Complessità spaziale:** la ricerca deve memorizzare un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi. Una volta che un nodo è stato espanso, può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati. La complessità sarà di $O(bk)$ dove b è la **ramificazione** e k la **profondità**.

Ricerca ad approfondimento iterativo

Questo tipo di ricerca ha l'obiettivo di identificare un **limite** per la ricerca a profondità limitata, in questo modo il limite viene incrementato progressivamente, finché un nodo obiettivo non è identificato.

```
function RICERCA-APPROFONDIMENTO-ITERATIVO(problema)
    returns una soluzione o un fallimento

    for profondità = 0 to ∞ do
        risultato ← RICERCA-PROFONDITÀ-LIMITATA(problema, profondità)
        if risultato != fallimento then return risultato
```

Oltre a migliorare la ricerca in profondità, la **ricerca ad approfondimento iterativo** è analoga a quella in ampiezza, poiché ad ogni iterazione esplora completamente un livello di nodi prima di prendere in considerazione il successivo.

In generale, questo tipo di ricerca è il metodo di **ricerca non informata** da preferire quando lo spazio di ricerca è grande e la profondità della soluzione non è nota.

Performance – Ricerca ad approfondimento iterativo

- **Completezza:** l'algoritmo è completo poiché aumentando progressivamente la profondità e muovendosi a pendolo, troverà il nodo obiettivo.
- **Complessità temporale:** nel caso pessimo, saranno generati $O(b^d)$ nodi.
- **Ottimalità:** se il costo di cammino è una funzione monotona non decrescente, allora la ricerca è ottima. In generale, però, l'algoritmo è ottimale perché restituisce la soluzione più vicina, indipendentemente dal costo.
- **Complessità spaziale:** la ricerca deve memorizzare un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi per ciascun nodo sul cammino. Una volta che un nodo è stato espanso, può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente. Per cui, la complessità sarà di $O(bd)$.

Ricerca bidirezionale

La **ricerca bidirezionale** effettua due ricerche in parallelo:

- la prima guidata dai dati;
- la seconda dall'obiettivo.

Questa strategia di ricerca esegue una prima ricerca in avanti dallo stato iniziale e l'altra all'indietro dallo stato obiettivo.

Questo tipo di ricerca è implementata sostituendo il test obiettivo con un **controllo** affinché le frontiere delle due ricerche si intersechino: in tal caso, è stata trovata una soluzione.

Il problema di questa ricerca è di saper fare **backtracking**. I *predecessori* di uno stato x sono tutti gli stati che hanno x come successore. Definire i predecessori non è sempre facile.

Capitolo 4 – Algoritmi di Ricerca Informata

4.1 Algoritmi di Ricerca Informata

Mentre gli *algoritmi di ricerca non informata* hanno l'obiettivo di trovare soluzioni senza avere a disposizione informazioni aggiuntive sul problema, gli **algoritmi di ricerca informata** sfruttano conoscenze specifiche del problema, così da essere più efficienti.

Questo tipo di ricerca è anche chiamato **ricerca best-first**: in cui il nodo da espandere viene scelto tramite una stima sul nodo mediante una funzione di valutazione $f(n)$.

In generale, l'implementazione della **ricerca best-first** è identica a quella della ricerca a costo uniforme: l'unica differenza è che la coda a priorità è ordinata in base ad $f(n)$ piuttosto che $g(n)$.

Molti algoritmi di *ricerca best-first* fanno riferimento ad una **funzione euristica** chiamata $h(n)$. Le funzioni euristiche sono il modo più comune per inserire una conoscenza aggiuntiva in un algoritmo di ricerca.

```
function RICERCA-COSTO-UNIFORME RICERCA-BEST-FIRST(problema, strategia)
    returns una soluzione o un fallimento

    nodo <- un nodo con stato = problema.STATO-INIZIALE e COSTO-DI-CAMMINO=0
    frontiera <- una coda a priorità ordinata per COSTO-CAMMINO, con nodo unico elemento
    frontiera <- una coda a priorità ordinata per COSTO-STIMATO, con nodo unico elemento
    esplorati <- un insieme vuoto

    loop do
        if la frontiera è vuota then return fallimento
        nodo <- POP(frontiera)
        if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
        aggiungi nodo.STATO a esplorati

        for each azione in problema.AZIONI(nodo.STATO) do
            figlio <- NODO-FIGLIO(problema, stato, azione)
            if figlio.STATO non è in esplorati e figlio non è in frontiera then
                frontiera <- INSERISCI(figlio, frontiera)
            else if figlio.STATO è in frontiera con COSTO-CAMMINO più alto then
                else if figlio.STATO è in frontiera con COSTO-STIMATO più alto then
                    sostituisci quel nodo frontiera con figlio
```

4.2 Ricerca Best-First Greedy

Gli **algoritmi greedy** sono tra i più utilizzati per problemi di ottimizzazione. L'algoritmo cerca sempre di espandere il nodo più vicino alla soluzione.

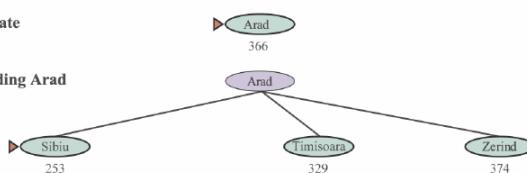
Inoltre, più è bassa la **funzione euristica** $h(n)$ più ci si sta avvicinando al nodo obiettivo; quindi, il prossimo nodo della frontiera da espandere verrà scelto fra quelli con $h(n)$ più basso.

Esempio:

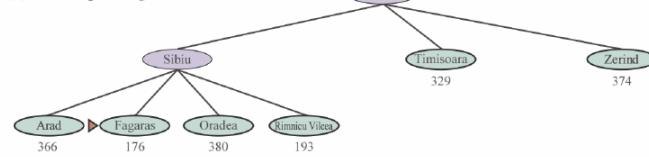
(a) The initial state



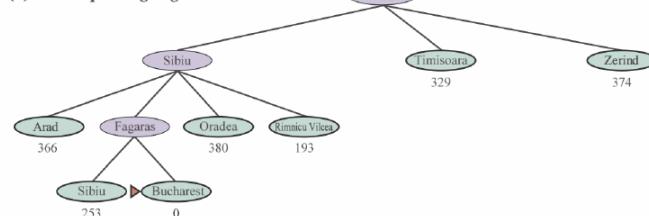
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



La soluzione non è ottima: esiste un cammino che va da Sibiu a Vilcea e poi a Pitesti che farebbe risparmiare 32km rispetto al tragitto identificato in maniera greedy.

Ecco perché l'algoritmo viene detto "goloso": ad ogni passo cerca sempre di arrivare più vicino all'obiettivo, non considerando cammini alternativi con stima di costo maggiore.

L'algoritmo greedy non è **ottimo né completo**.

- **Complessità temporale:** nel caso pessimo, l'algoritmo ha complessità $O(b^m)$, dove m è la **profondità massima** dello spazio di ricerca. Tuttavia, con una buona euristica, è possibile ridurre drasticamente tale complessità.
- **Complessità spaziale:** è $O(b^m)$, poiché l'algoritmo conserva tutti i nodi in memoria.

4.3 Ricerca A*

La forma più diffusa di *ricerca best-first* è la **ricerca A***, la quale viene eseguita combinando $g(n)$, ovvero il costo per raggiungere il nodo, con $h(n)$, ovvero il costo dal nodo n all'obiettivo.

Il costo della soluzione più conveniente che passa per n che si usa in A* è $f(n) = g(n) + h(n)$, con:

- $h(n)$ = stima per andare da n fino al nodo goal;
- $g(n)$ = costo per arrivare in n ;

Quindi:

- Se $h(n) = 0$, allora $f(n) = g(n) \rightarrow$ si ha una **Ricerca Uniforme**;
- Se $g(n) = 0$, allora $f(n) = h(n) \rightarrow$ si ha una ricerca **Best First Greedy**.

Performance

- **Ottimalità:** l'ottimalità di A* dipende da 2 fattori:
 - la prima è che $h(n)$ sia un'**euristica ammissibile**, ovvero che non sbaglia mai la stima del costo per arrivare all'obiettivo.
 - la seconda è che $h(n)$ sia un'**euristica consistente**, ovvero per ogni nodo n e ogni successore n' generato da un'azione a , il costo stimato per raggiungere l'obiettivo partendo da n è inferiore al costo per arrivare a n' sommato al costo stimato per andare da lì all'obiettivo.

Questa è una forma di **disuguaglianza triangolare** cioè ogni lato di un triangolo non può mai essere più lungo della somma degli altri due.

- **Completezza:** richiede un numero finito di nodi di costo $\leq C * (\text{la soluzione ottima})$, questo è vero se tutti i costi dei passi superano un valore finito ϵ e se b è finito.
- **Complessità temporale:** l'algoritmo ha complessità $O(b^\epsilon)$, il che lo rende più efficiente degli altri algoritmi *best-first*.
- **Complessità spaziale:** l'algoritmo tiene in memoria tutti i nodi generati, quindi sarà $O(b^m)$.

Quindi, con **l'algoritmo A*** riusciamo a generare soluzioni con una complessità temporale migliore, non riuscendo a migliorare quella spaziale.

4.4 Migliorare l'occupazione di memoria

Beam Search

La **Beam Search** è una variante della *ricerca best-first* che non conserva in memoria tutti i nodi generati, ma tiene ad ogni passo solo i k nodi più promettenti, con k definito come ampiezza del raggio.

L'ampiezza del raggio è definita in base ad una *euristica*: solo i nodi nel raggio vengono conservati e considerati come candidati per la soluzione.

La **Beam Search** è, quindi, a tutti gli effetti una soluzione greedy alla risoluzione dei problemi, poiché espanderà sempre il nodo che ritiene essere più utile al raggiungimento della soluzione.

Sebbene migliori l'occupazione di memoria l'algoritmo non è **completo** né **ottimale**.

Iterative Deepening A* (IDA*)

Funziona nella stessa maniera della **ricerca iterativa a profondità limitata**, la differenza principale sta nel valore del taglio, che non è più basato sulla profondità ma sul costo f , ovvero su $g + h$.

Essendo un'estensione di A*, ne mantiene le proprietà di completezza e ottimalità, a patto che ci siano le condizioni di **ammissibilità** e **consistenza**.

Il vantaggio si ottiene in termini di **occupazione di memoria**, poiché la ricerca dovrà memorizzare un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi. Una volta che un nodo è stato espanso, può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente. Per cui, la complessità sarà di $O(bd)$.

Ricerca Best-First ricorsiva

Questa ricerca è una variante della *ricerca best-first classica*, ma usa solo uno **spazio lineare**. Ad ogni iterazione, l'algoritmo tiene traccia del miglior percorso alternativo. Invece di fare *backtracking* in caso di fallimento, interrompe l'esplorazione quando trova un nodo meno promettente (definito sulla base di f).

```
function RICERCA-BEST-FIRST-RICORSIVA(problema)
    returns una soluzione o il fallimento

    returns RBFS(problema, CREA-NODO(problema.STATO-INIZIALE), ∞)

function RBFS(problema, nodo, f_limite)
    returns una soluzione o il fallimento e un nuovo limite all'f-costo

    if problema.TEST-OBIETTIVO(nodo.STATO) then returns SOLUZIONE(nodo)
    successori ← [ ]

    for each azione in problema.AZIONI(nodo.STATO) do
        aggiungi NODO.FIGLIO(problema, nodo, azione) a successori

    if successori è vuoto then returns fallimento, ∞

    for each s in successori do
        s.f ← max(s.g + s.h, nodo.f)
    loop do
        migliore ← il nodo con f-valore minimo in successori
        if migliore.f > f_limite then returns fallimento, migliore.f
        alternativa ← il secondo nodo con f-valore minimo in successori
        risultato, migliore.f ← RBFS(problema, migliore, min(f_limite, alternativa))
        if risultato ≠ fallimento then return risultato
```

La ricerca **Best-First ricorsiva** è **ottimale** se la funzione euristica $h(n)$ è ammissibile.

La **complessità spaziale** è lineare $O(bd)$.

La **complessità temporale** è però difficile da definire. Questa dipende sia dalla funzione euristica sia dalla frequenza dei cambiamenti del cammino ottimo durante l'espansione dei nodi. Nel caso pessimo, però, sarà **esponenziale**.

Il problema della ricerca *Best-First ricorsiva*, così come di *IDA**, consiste nel fatto che utilizzano **troppa memoria** portando a dei loop.

Simplified Memory Bounded A* (SMA*)

L'idea alla base è quella di utilizzare meglio la memoria disponibile. SMA* procede come A* fino all'esaurimento della memoria disponibile. Quando questo accade, allora l'algoritmo libera il nodo peggiore, ovvero quello con l' f -valore più alto.

Come RBFS memorizza nel nodo padre il valore del nodo dimenticato. Questo meccanismo consente all'algoritmo di tenere traccia della radice di un sottoalbero dimenticato, avendo quindi a disposizione l'informazione sul cammino migliore di quel sotto-albero.

SMA* rigenera un sotto-albero dimenticato solo quando tutti gli altri cammini promettono di comportarsi peggio di quello.

- SMA* è **completo** se la soluzione è raggiungibile;
- è **ottimale** se c'è una soluzione ottima raggiungibile.

Il problema di questo algoritmo consiste nel fatto che, per problemi difficili, la ricerca sarà obbligata a passare continuamente dall'uno all'altro tra molti cammini candidati di cui sarà possibile memorizzare solo un piccolo sottoinsieme. Quando questo accade, il problema potrebbe diventare intrattabile, infatti, la **complessità temporale** potrebbe esplodere, rendendo il problema non risolvibile nella pratica.

4.5 Funzione euristica

Molti problemi dell'Intelligenza Artificiale sono di complessità esponenziale: tuttavia, c'è esponenziale ed esponenziale.

Una **buona euristica** può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca.

Per la risoluzione di problemi, spesso si dovrebbe "inventare" delle euristiche seguendo questi punti:

- **Rilassamento dei vincoli:** pensare ad un problema con meno vincoli;
- **Massimizzazione di euristiche:** se si avessero una serie di euristiche, senza dominanza tra queste, allora potrebbe convenire massimizzare la funzione: $h(n) = \max(h_1(n), h_2(n), \dots, h_k(n))$.
- **Apprendere dall'esperienza:** usare dati passati per apprendere un'euristica; quindi utilizzare tecniche di apprendimento per la risoluzione di problemi di ricerca;
- **Combinare euristiche:** quando diverse caratteristiche influenzano la bontà di uno stato, allora si può usare una combinazione lineare del tipo: $h(n) = c_1 \cdot h_1(n) + c_2 \cdot h_2(n) + \dots + c_k \cdot h_k(n)$.

Capitolo 5 – Algoritmi di Ricerca Locale

5.1 Algoritmi di ricerca locale

Gli algoritmi studiati finora sono progettati per esplorare lo spazio degli stati: per farlo, tengono in memoria uno o più cammini e registrano quali alternative sono state esplorate in ogni punto del cammino.

Quando viene raggiunto uno stato obiettivo, il cammino verso quello stato costituisce una soluzione del problema.

Tuttavia, in molti casi il cammino che porta alla soluzione è **irrilevante** (ad esempio, nel problema delle N regine ciò che conta è la disposizione finale delle regine sulla scacchiera, non l'ordine con cui sono state aggiunte).

Infatti, in molti **problemi di ottimizzazione** lo stato obiettivo è la soluzione al problema indipendentemente da come ci si è arrivati.

Quindi, dal punto di vista algoritmico, dato lo stato iniziale, si fa una ricerca negli stati successori, e ci si sposta nel migliore degli stati successori e si continua in quella direzione. Si chiama **ricerca locale** perché quando ci si muove nello stato successore ci si dimentica del passato, quindi ora non interessa più costruire un albero di ricerca perché non si ha bisogno di ricordare.

L'unica cosa che interessa in questo tipo di problemi è:

- lo **stato attuale**;
- una **funzione** che indica quanto si sta andando bene se ci si sposta in una direzione piuttosto che in un'altra;
- **muoversi**.

La struttura dati che viene data all'algoritmo è sempre un grafo dello spazio degli stati, ma la sola struttura che poi serve durante l'algoritmo è il **nodo**.

Algoritmi di <u>ricerca “tradizionali”</u>	Algoritmi di <u>ricerca locale</u>
<p>Considerano interi cammini dallo stato iniziale a quello obiettivo.</p> <p>Hanno una complessità temporale e spaziale generalmente esponenziale.</p> <p>Non possono essere applicati in problemi con spazio degli stati grandi/infiniti.</p>	<p>Considerano lo stato corrente con l'obiettivo di migliorarlo.</p> <p>Usano poca memoria, molto spesso avendo una complessità costante.</p> <p>Possono trovare soluzioni ragionevoli (sub-ottimali) a problemi complessi.</p>

5.1.1 La funzione obiettivo

Gli algoritmi di **ricerca locale** non hanno un test obiettivo, ma affidano le loro azioni ad una **funzione obiettivo** che indica quanto la ricerca sta migliorando nelle iterazioni. Questo li rende adatti alla risoluzione di classici **problemi di ottimizzazione**.

La scelta della funzione obiettivo è fondamentale per l'efficienza del sistema e definisce l'insieme delle soluzioni che possono essere raggiunte da una soluzione s in un singolo passo di ricerca dell'algoritmo.

Inoltre, la **funzione obiettivo** è definita attraverso le possibili mosse che l'algoritmo può effettuare. In altri termini la *ricerca locale* si basa sull'**esplorazione iterativa** delle “soluzioni vicine” che possono migliorare la soluzione corrente mediante **modifiche locali**.

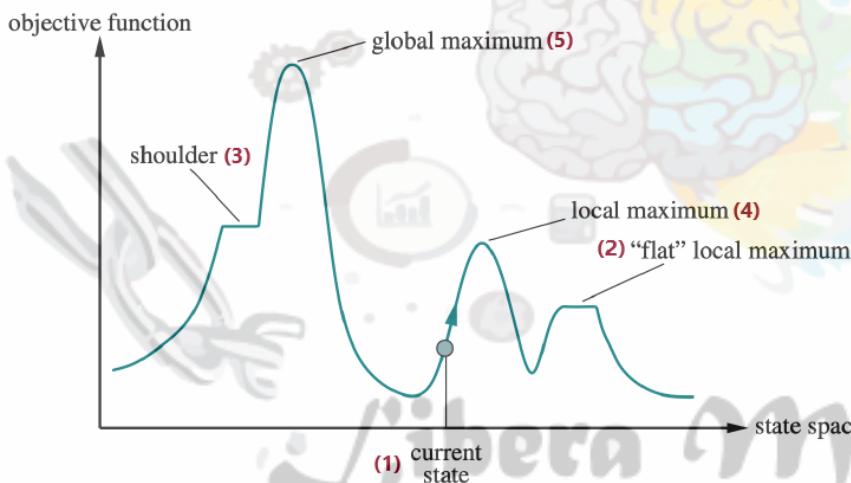
Una **struttura dei vicini** (neighborhood) è una funzione F che assegna ad ogni soluzione s , dell'insieme di soluzioni S , un insieme di soluzioni $N(s)$ che è sottoinsieme di S .

È importante notare che la soluzione trovata, da un *algoritmo di ricerca locale*, non è detto che sia **globalmente ottima**, ma può essere ottima rispetto ai **cambiamenti locali**.

5.1.2 Spazio degli stati

Per poter comprendere il funzionamento degli *algoritmi di ricerca locale*, possiamo utilizzare un piano cartesiano in cui:

- sull'**asse delle x** avremo lo **spazio degli stati**, cioè i vari stati raggiungibili dalla ricerca;
- sull'**asse delle y** avremo una **funzione obiettivo** associata a ciascuno stato;



La figura riporta il caso in cui abbiamo un problema di massimizzazione del risultato.

Chiaramente, se volessimo minimizzare allora andremmo alla ricerca del *global minimum*.

1. Lo **stato corrente** (current state) indica la posizione in cui si trova la ricerca rispetto al panorama degli stati;
2. Un **punto piatto** (flat) dello spazio degli stati indica una regione dello spazio dove gli stati vicini avranno lo stesso valore (anche detto plateau).
3. Una **spalla** (shoulder) è un *plateau* che presenta uno spigolo in salita.
4. Un **massimo locale** (local maximum) è una soluzione sub-ottima ad un problema di ricerca.
5. L'**ottimo globale** (global maximum) è la soluzione migliore in assoluto al problema.

5.1.3 Proprietà

Un *algoritmo di ricerca locale completo* trova sempre un obiettivo, se questo esiste.

Un *algoritmo di ricerca locale ottimo* trova sempre un minimo/massimo globale.

Più formalmente, un massimo o minimo locale è una soluzione s tale che, data una **funzione di fitness** (di valutazione) f , vale la seguente equazione: $\forall s' \in N(s), f(s) \geq f(s')$.

Quando risolviamo un problema di massimizzazione (minimizzazione) cerchiamo un massimo (minimo) globale, s_{opt} , cioè una soluzione tale che $\forall s, f(s_{opt}) \geq f(s)$.

Banalmente, più è largo il **neighborhood** più è probabile che un massimo/minimo locale sia anche globale e quindi migliore sarà la qualità della soluzione.

5.2 Algoritmo Hill-Climbing

L'algoritmo **Hill-Climbing** è il più semplice tra gli algoritmi di *ricerca locale* e ha l'obiettivo di “**scalare**” lo spazio di ricerca per trovare un **massimo/minimo globale**.

```
function HILL-CLIMBING(problema) returns uno stato che è un massimo
    nodo_corrente ← CREA-NODO(problema.STATO-INIZIALE)
    loop do
        vicino ← il successore di nodo_corrente di valore più alto
        if vicino.VALORE <= nodo_corrente.VALORE then return nodo_corrente.STATO
        nodo_corrente ← vicino
```

Ad ogni passo, il **nodo corrente** viene rimpiazzato dal **vicino** migliore (lo pseudocodice fa riferimento ad un problema di massimizzazione). L'algoritmo termina quando raggiunge un picco che non ha vicini di valore più alto.

È da notare che l'algoritmo non mantiene un albero di ricerca, per cui la struttura dati deve solo memorizzare lo stato e il valore della funzione obiettivo del nodo corrente.

Svantaggi

L'algoritmo viene a volte chiamato **ricerca locale greedy**, poiché segue una strategia simile agli algoritmi di *ricerca best-first greedy*, poiché sceglie uno stato “buono” senza pensare a come andrà avanti.

Sfortunatamente, l'*algoritmo Hill-Climbing* spesso non riesce ad identificare la soluzione migliore a livello globale per varie ragioni:

- gli *algoritmi Hill-Climbing* che raggiungono la vicinanza di un **massimo locale**, saranno attirati verso il picco, ma rimarranno bloccati lì senza poter andare altrove.
- gli *algoritmi Hill-Climbing* che raggiungono un **plateau** rischiano di bloccarsi poiché non riusciranno a migliorare la soluzione corrente e saranno quindi costretti a restituire una soluzione sub-ottimale.
- I cosiddetti “**ridge**” (creste) rappresentano delle porzioni dello spazio di ricerca che presentano una brusca variazione. In questi casi, l'algoritmo analizzerà una serie di massimi locali da cui è difficile uscire.

Quindi, massimi locali, plateau e creste possono deteriorare le prestazioni degli algoritmi Hill-Climbing. In ognuno di questi casi, infatti, questi algoritmi raggiungeranno un punto dal quale non riusciranno a fare ulteriori progressi.

5.2.1 Miglioramenti Hill-Climbing

Per poter migliorare questo tipo di algoritmi potremmo stabilire un **limite massimo** al numero di determinate azioni che l'algoritmo può eseguire. Questo chiaramente aumenta il tasso di successo dell'algoritmo, impattando però la sua efficienza.

Esistono altre varianti dell'algoritmo di base dell'Hill-Climbing, le quali vanno a lavorare sulla probabilità che l'algoritmo riesce ad uscire da un massimo locale ed esplorare lo spazio di ricerca in maniera più efficace. Anche in questi casi, però, l'efficienza degrada. La scelta dell'algoritmo da utilizzare dipende dal tipo di problema da affrontare.

5.2.2 Varianti algoritmo Hill-Climbing

L'**Hill-Climbing stocastico** sceglie in maniera casuale il successore; infatti, a differenza dell'algoritmo di base, non sceglie immediatamente il successore più conveniente, aumentando le chance di navigare lo spazio di ricerca in maniera più estesa.

L'**Hill-Climbing con prima scelta** può generare i successori a caso fino a trovarne uno migliore dello stato corrente. Questa strategia è molto buona quando uno stato ha molti (migliaia) successori, poiché l'algoritmo riuscirà a navigare meglio lo spazio di ricerca.

L'**Hill-Climbing con riavvio casuale** può consentire alla ricerca di ripartire da un punto a caso dello spazio di ricerca (è da notare che generare uno stato casuale in uno spazio degli stati può essere di per sé un problema difficile).

Il *riavvio casuale* consente all'algoritmo di avviare una serie di ricerche hill-climbing. Inoltre, questo algoritmo può essere completo perché prima o poi dovrà generare, come stato iniziale, proprio un obiettivo. In generale, il successo di questo algoritmo dipende dalla forma del panorama dello spazio degli stati: se ci sono pochi massimi locali e plateau, allora troverà una soluzione molto velocemente. Ma purtroppo, non è sempre così (i problemi *NP-hard* ne sono un esempio).

5.3 Algoritmo Simulated Annealing

Un modo per uscire dai massimi locali è scegliere degli stati che nel momento attuale non sembrano migliori. Quindi, l'algoritmo **Simulated Annealing** combina una ricerca hill-climbing con un'esplorazione casuale, per migliorare l'efficienza.

All'inizio, l'algoritmo "scuoterà" molto la ricerca, variando molto nel panorama degli stati, per poi man mano ridurre l'intensità di questo meccanismo, concentrandosi quindi su un punto specifico del panorama e cercando una soluzione ottima in quel contesto.

```
function SIMULATED_ANNEALING(problema, velocità_raffreddamento)
    returns uno stato soluzione

    nodo_corrente <- CREA-NODO(problema.STATO-INIZIALE)

    for t <- 1 to ∞
        T <- velocità_raffreddamento[t]
        if T = 0 then return nodo_corrente
        successivo <- un successore scelto a caso di nodo_corrente
        ΔE <- successivo.VALORE - nodo_corrente.VALORE

        if ΔE > 0 then nodo_corrente <- successivo
        else nodo_corrente <- successivo solo con probabilità e^ΔE/T
```

Il parametro `velocità_raffreddamento` è un'analogia che sta ad indicare il tempo. La velocità di raffreddamento indicherà quanto T venga ridotto ad ogni passo dell'algoritmo.

Il ciclo interno è simile a quello dell'Hill-Climbing, ma questa volta il nodo successore sarà scelto a caso. Se il successore ha un valore migliore del nodo corrente, allora la mossa verrà sempre accettata e l'algoritmo procederà ad esplorare lo spazio intorno a quello stato.

In alternativa, il successore potrebbe essere accettato ugualmente, ma con una probabilità inferiore ad 1.

Inoltre, mosse "cattive" verranno accettate più facilmente all'inizio poiché T sarà alto e diventeranno sempre meno probabili a mano a mano che T si abbassa. Se la temperatura si abbassa abbastanza lentamente, l'algoritmo troverà un ottimo globale con una probabilità tendente ad 1.

5.4 Algoritmo Local Beam

L'algoritmo di ricerca **local beam** tiene traccia di k stati anziché uno. All'inizio comincia con k stati generati casualmente: ad ogni passo, sono poi generati i successori di tutti i k stati. Se uno di questi è un obiettivo, la ricerca termina. Altrimenti, scegli i k successori migliori dalla lista e ricomincia.

Potrebbe sembrare simile all'*Hill-Climbing con riavvio casuale*, ma nella ricerca con riavvio casuale ogni ricerca è del tutto indipendente dalle altre.

Nella *local beam*, invece, la ricerca è “**informata**”, cioè le ricerche sono dipendenti e questo algoritmo andrà a generare dei cammini di ricerca non ridondanti e che migliorano l'*Hill-Climbing con riavvio casuale*.

Questo aumenta le chance di trovare stati più “promettenti” e quindi l'ottimo globale, però d'altro canto, la *local beam* potrebbe soffrire del **problema di diversificazione tra i k stati**: cioè i k stati, casualmente generati, si trovano in una certa porzione dello spazio di ricerca portando l'algoritmo a focalizzarsi su una piccola porzione dello spazio.

Un'alternativa è rappresentata dalla ricerca **local beam stocastica**, che è ispirata ai concetti dell'*Hill-Climbing stocastico*.

Invece di scegliere tra i candidati i k successori migliori, si scelgono **k successori a caso** e si assegna una probabilità maggiore di scelta.

5.5 Teoria dell'evoluzione e algoritmi genetici

La teoria di selezione naturale, introdotta da Charles Darwin, la si può riassumere in 4 punti fondamentali:

1. **Variation**: gli individui in una popolazione posseggono un patrimonio genetico diverso (genotipo), che implica molte variazioni nelle loro caratteristiche fisiche (fenotipo).
2. **Inheritance**: gli individui si riproducono e trasmettono parte del loro materiale genetico alla loro prole.
3. **Selection (and Adaptation)**: alcuni individui possiedono tratti ereditati che gli permettono di sopravvivere più a lungo in un ambiente e/o produrre ancora più prole. Di conseguenza, questi tratti tenderanno ad essere predominanti nell'intera popolazione.
4. **Time**: a lungo andare, la selezione può comportare la nascita di nuove specie (speciazione).

5.5.1 Ottimizzazione

Un problema di **ottimizzazione** consiste nel trovare la/le soluzione/i cioè il valore restituito dalla **funzione obiettivo** sia il massimo/minimo globale.

Quindi, gli **algoritmi di ottimizzazione** hanno il compito di esplorare lo **spazio di ricerca** con l'obiettivo di trovare il punto di ottimo globale.

Alcuni di questi algoritmi sono detti **completi**, cioè algoritmi in grado di trovare per certo l'ottimo globale a volte, però, con un tempo molto elevato. È necessario, perciò, accontentarsi di soluzioni approssimate, vicine all'ottimo globale, ma comunque accettabili. Quindi l'idea della **ricerca globale** è di trovare diverse **soluzioni candidate** premiando quelle più *promettenti* (vicino all'ottimo) e penalizzando quelle meno *promettenti* (più lontane).

Altri algoritmi, invece, sono detti **incompleti**.

5.5.2 Algoritmi Genetici

Gli **algoritmi evolutivi** sono algoritmi di ottimizzazione ispirati ai meccanismi della **teoria dell'evoluzione**.

Un **algoritmo genetico** è una procedura ad alto livello (metauristica) ispirata dalla genetica per definire un **algoritmo di ottimizzazione** capace di esplorare in maniera efficiente lo spazio di ricerca.

Non sono algoritmi già definiti pronti all'uso, bensì una sorta di **framework** che definiscono un nostro algoritmo di ottimizzazione.



In quanto algoritmi meta-euristiche, essi sono:

- **Slegati dallo specifico problema:** sono in grado di risolvere un'ampia gamma di problemi di ottimizzazione e di ricerca;
- **Efficienti:** esplorano in maniera rapida gran parte dello spazio di ricerca;
- **Approssimati:** forniscono soluzioni sub-ottimali.

In sintesi, un **algoritmo genetico** fa evolvere **iterativamente** una **popolazione di individui** (soluzioni candidate), producendo di volta in volta nuove **generazioni** di individui migliorati, rispetto ad una cosiddetta **misura di fitness**, finché uno o più criteri di arresto non sono soddisfatti.

Un tipico criterio di arresto è stabilito dal **tempo di esecuzione**: l'evoluzione termina dopo X secondi di esecuzione. Questo X prende il nome di **budget di ricerca**;

La generazione di nuovi individui avviene per mezzo di 3 operatori di ricerca:

1. **Selezione:** un sottoinsieme della popolazione viene candidato alla riproduzione e duplicato in una generazione intermedia (fase detta *mating pool*). Il criterio di selezione dovrebbe produrre individui con un'alta *fitness* (maggiore probabilità di essere selezionati).
2. **Crossover:** gli individui selezionati (parents) sono abbinati casualmente per generare della prole (offsprings) incrociando il loro corredo genetico (la codifica). I figli generati rimpiazzeranno i genitori nella generazione intermedia. Il crossover è anche noto come recombination.
3. **Mutazione:** alcuni geni (singoli elementi delle codifiche) presenti tra tutti gli individui sono mutati casualmente (bassa probabilità). La mutazione, di solito, non deve dare individui con codifiche fuori dal dominio del problema.

Il crossover e la mutazione permettono di **esplorare (explore)** efficientemente lo spazio di ricerca, andando a ridurre il rischio di bloccarsi in ottimi locali.

La selezione permette di **sfruttare (exploit)** le soluzioni più promettenti nella speranza che esse permettano la generazione di altre più promettenti.

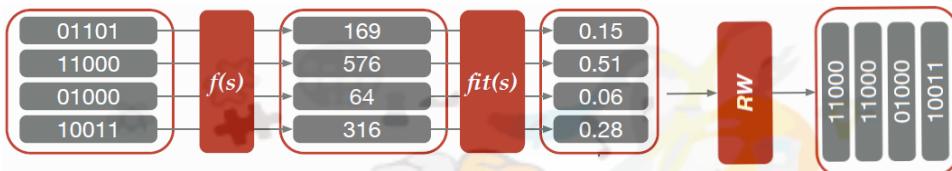
Adesso analizziamo alcuni criteri da applicare ai tre operatori di ricerca:

- **Roulette Wheel Selection:** ogni individuo riceve una porzione di una ruota proporzionata al valore di valutazione relativo al resto delle valutazioni. In questo caso, la *funzione di fitness* è un valore compreso tra 0 e 1:

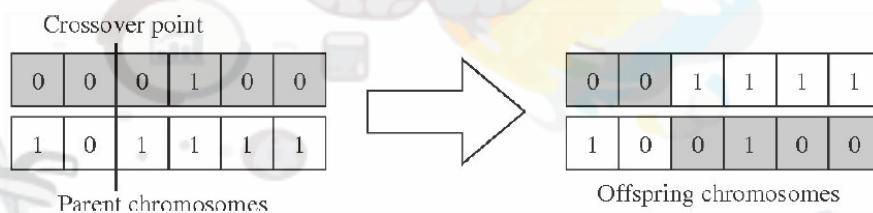
$$\text{fit}(s) = \frac{f(s)}{\sum f(s)}$$

La ruota viene girata un numero di volte pari al numero degli individui, e i vincitori sono ammessi al *mating pool*. Ne segue che un individuo può vincere più volte, quindi potrebbero esserci dei cloni nel *mating pool*.

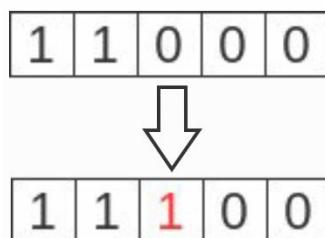
Esempio:



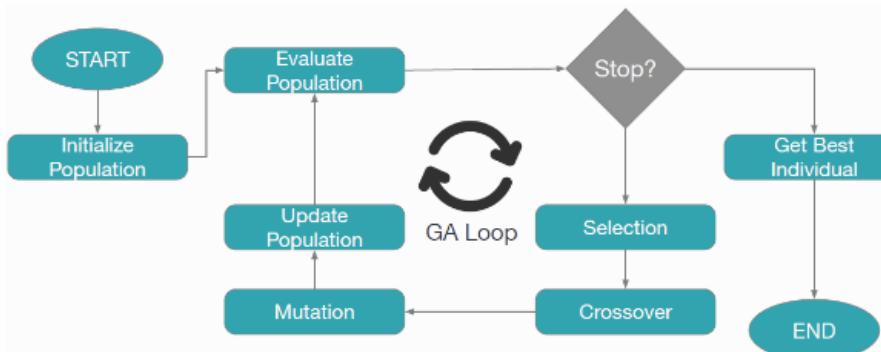
- **Single-point Crossover:** si sceglie casualmente un punto di taglio e si incrociano le due parti per dare vita a due *offspring*. Per quanto riguarda l'abbinamento dei genitori, si creano abbinamenti casuali di tutti gli individui e ogni individuo deve partecipare all'abbinamento una sola volta.



- **Bit-flip Mutation:** scegliamo una mutazione che segue una probabilità molto bassa (arbitraria e di solito molto bassa), poi si mettono in fila i bit e li si scorrono. Se la probabilità è rispettata, allora si muta un bit. Avendo codifiche binarie, possiamo optare per un'operazione di **bit flip**, ovvero l'inversione dello 0 con 1, e viceversa.



Algoritmo Genetico



Caratteristiche degli Algoritmi Genetici

Goldberg identifica 4 caratteristiche fondamentali dei GAs:

1. **Encoding** (codifica): un GA necessita di codificare gli individui di solito in stringhe binarie di lunghezza finita.
2. **Global Search** (ricerca globale): un GA usa un insieme di soluzioni candidate per esplorare da più punti lo spazio di ricerca. Questo riduce il rischio di “bloccarsi in ottimi locali”.
3. **Blindness** (cieco): un GA non ha bisogno di conoscere internamente la funzione da ottimizzare: è necessario che sia soltanto accessibile in maniera **black-box**. Qualora non dovesse esserlo, si possono adottare delle approssimazioni.
4. **Probabilistic**: un GA include componenti di casualità che servono a guidare la ricerca (senza degenerare in un algoritmo random search).

Esempio:

Consideriamo una scatola nera con 5 interruttori. Di questa scatola sappiamo solo che il suo comportamento è modellato dalla funzione $f(s)$, dove s è una configurazione dei 5 interruttori. La funzione restituisce un valore intero, il nostro obiettivo è massimizzarla, ovvero trovare la configurazione che dia il massimo di $f(s)$.

Cosa facciamo a stabilire se un individuo sia “migliore” di un altro?

Abbiamo bisogno di una **funzione di valutazione**, ovvero una funzione che prende in input la codifica di un individuo e restituisce un valore che rappresenta la bontà di tale individuo.

Quale scegliamo?

Dato che la funzione obiettivo $f(s)$ è accessibile e non richiede molte risorse per essere calcolata, possiamo riusarla come **funzione di valutazione**. Questo riuso avviene nella maggior parte dei casi, ma in altri casi la funzione obiettivo potrebbe essere complessa da calcolare o addirittura impossibile. Quindi, conviene definire la funzione di valutazione come un'approssimazione della vera funzione obiettivo. Infatti, è molto importante che la funzione di valutazione sia veloce, visto che sarà necessario eseguirla un numero elevatissimo di volte ad ogni iterazione.

Quale criterio di arresto scegliamo?

In questo esempio, adottiamo il seguente criterio: “se il miglior individuo non ha una valutazione superiore al migliore della generazione precedente, allora l’algoritmo termina”.

Pseudocodice

```
function SIMPLE-GA(fitness_function, stopping_condition)
    returns an individual
    P <- init_population()
    while stopping_condition is false
        evaluate(fitness_function, P)
        P_1 <- selection(P)
        P_2 <- crossover(P_1)
        P_3 <- mutation(P_2)
        P <- P_3
        b <- get_best_individual(fitness_function, P)
    return b
```

Questa è la forma più semplice di un *algoritmo genetico* di tipo **generazionale** noto come **Simple Genetic Algorithm (SGA)**: ad ogni iterazione si crea una nuova generazione.

Da un punto di vista operativo è sempre bene conservare la storia dell'intera evoluzione invece che rimpiazzare del tutto quella precedente. Quindi conviene mantenere una struttura dati che conservi lo storico di tutte (o almeno le ultime X) le generazioni.

Vantaggi dei GAs

Flessibili

I GAs possono essere visti come dei *framework* in grado di risolvere un'ampia gamma di problemi. Due sono i punti cruciali da affrontare: codifica delle soluzioni candidate, e modellazione della funzione obiettivo. Se questi due task sono risolti facilmente, allora i GA sono una buona scelta.

Esplorazione rapida

La loro capacità di compiere ricerca globale permette una rapida esplorazione dello spazio di ricerca, evitando in poco tempo aree lontane dall'ottimo. Sono in grado di restituire risultati accettabili in poco tempo.

Continuo e discreto

I GAs sono in grado di ottimizzare sia *funzioni nel continuo* che nel *discreto*, e si prestano bene quando abbiamo funzioni che accettano input con molte variabili o comunque strutturati in modo complesso, dove gli altri algoritmi di ottimizzazioni falliscono.

Svantaggi dei GAs

Parametri

Ci sono tantissimi componenti e parametri da decidere, e non esiste la combinazione ottimale nota a priori e generale (**No Free Lunch Theorem**). Quindi, è necessario fare valutazioni empiriche delle combinazioni che si reputano più promettenti. Conoscere al meglio il problema modellato permette di scartare alcune combinazioni poco promettenti.

5.5.3 Modellazione del problema

La vera forza dei GA risiede nell'avere molte componenti (Selezione, Crossover, Mutazione, Codifica, etc.): invece di adattare un algoritmo di ottimizzazione per risolvere un problema di ottimizzazione, conviene adattare il problema alla meta-euristica.

Il problema specifico, infatti, potrebbe avere tante particolarità difficili da affrontare, mentre le componenti di un GA, pur essendo tante, sono semplici e soprattutto ricorrenti tra tanti problemi.

Esistono molti algoritmi di selezione, crossover e mutazione ben noti e già pronti all'uso però la difficoltà maggiore risiede soltanto in 2 aspetti:

1. stabilire la **funzione obiettivo**;
2. scegliere un'opportuna **codifica degli individui**.

Componenti dei GAs

➤ Codifica degli Individui

La **codifica degli individui** deve essere sempre una stringa binaria poiché si presta bene agli operatori che abbiamo descritto. Ha, però, un grande problema: **non scala bene**. Per rappresentare soluzioni “piccole” abbiamo bisogno di grandi codifiche. Questo problema si accentua considerando che facciamo evolvere centinaia/migliaia di individui per centinaia/migliaia di iterazioni.

Quindi, bisogna trovare codifiche quanto più compatte possibili, codificando soltanto le parti essenziali. Qualora non fosse possibile, si opta per altri tipi di codifiche, ad esempio:

- **Stringhe di numeri reali** (real-coded GA);
- **Strutture dati** (e.g., alberi, grafi);
- **altre strutture complesse** (e.g., reti neurali, programmi).

La **codifica reale** è molto gettonata in quanto più compatta e più vicina a problemi che prendono input numerici. In questo scenario i classici operatori di *crossover* e *mutazione* creano meno diversificazione rispetto al caso binario.

Inoltre, la scelta della **codifica** ha un impatto nella scelta di algoritmi di crossover e mutazione.

➤ Funzione Obiettivo

Ricordiamo la differenza tra **funzione obiettivo** e **funzione di valutazione**:

- **Funzione obiettivo**: è una funzione matematica da ottimizzare che modella il problema.
- **Funzione di valutazione**: è una funzione che assegna un valore di bontà agli individui.

La **funzione di valutazione** è un concetto legato ai GAs, mentre la **funzione obiettivo** è legata al problema da risolvere.

Le due spesso coincidono quando non possiamo (o non vogliamo) usare direttamente la *funzione obiettivo* per ragioni di performance o accessibilità. Dove è possibile, è bene valutare gli individui direttamente con la funzione obiettivo.

La funzione obiettivo possiamo modellarla noi, infatti:

- Se la *funzione obiettivo* che abbiamo modellato è eseguibile in tempi accettabili, allora possiamo anche riusarla come *funzione di valutazione*.
- Se la *funzione obiettivo* che abbiamo modellato NON è eseguibile in tempi accettabili, allora possiamo definire la nostra *funzione di valutazione* che approssima/stima la *funzione obiettivo*.

➤ Selezione

L'operatore di **selezione** realizza il meccanismo della selezione naturale in modo da **favorire gli individui più promettenti** (quelli con alto punteggio di valutazione) e **sfavorire gli altri** (quelli con basso punteggio).

La selezione non avviene usando direttamente il punteggio di valutazione, ma attraverso un **punteggio di fitness**, calcolato usando una **funzione di fitness**. La **Roulette Wheel Selection** adotta una *funzione di fitness* basata sulla valutazione di ciascun individuo rispetto agli altri (**fitness-proportionate**).

Una selezione di tipo **fitness-proportionate** porta con sé alcuni rischi. Infatti, con popolazioni piccole si rischia di favorire troppo i migliori individui, che verrebbero selezionati molte volte, andando via via a perdere la diversità nella popolazione.

Infatti, gli *algoritmi genetici* fanno della diversità un loro punto di forza: senza di essa non avremmo sufficiente “*forza di ricerca globale*”, rischiando di degenerare in una ricerca locale classica (come Hill Climbing).

La perdita di diversità, infatti, ha una conseguenza negativa. Quando gli individui in una popolazione iniziano a somigliarsi troppo, il **crossover** può dar luogo a figli geneticamente simili ai genitori, riducendo le probabilità di essere migliori di loro. Questo fenomeno prende il nome di **convergenza prematura**, che si può osservare quando l’evoluzione smette di dare miglioramenti.

Esistono diverse strategie per ridurre questo rischio. Le principali forniscono una giusta pressione di selezione favorendo gli individui forti senza trascurare quelli meno forti.

- **Rank Selection:** si valutano tutti gli individui e si ordinano rispetto al punteggio di valutazione ricevuto. A ciascuno viene assegnata una posizione nell’ordinamento chiamata **rango**. Dopodiché, la *funzione di fitness* assegna a ciascun individuo una probabilità di selezione proporzionale al rango, così calcolata: $fit(x) = (|P| - rank(x) + 1) / \sum rank(x)$

Questa soluzione è applicabile anche con **valutazioni negative** (impossibile con Roulette Wheel per via della somma presente al denominatore). In termini computazionali costa di più della *Roulette Wheel* per via dei continui ordinamenti da applicare alle varie generazioni.

- **K-way Tournament Selection:** fissato un valore $1 < K < |P|$ arbitrario, si selezionano casualmente K individui per formare un cosiddetto “**torneo**”, all’interno del quale il migliore (con valutazione massima) passerà la selezione. Si ripete per $M < |P|$ (valore arbitrario) tornei, quindi fino ad ottenere un totale di M individui selezionati.

Non richiede alcun ordinamento ed è applicabile anche con **valutazioni negative**. Incarna in maniera fedele il concetto della “*legge del più forte*”. Esistono delle varianti, ad esempio, invece di far vincere sempre il migliore si applica una *mini-Roulette Wheel*, oppure si impediscono vittorie multiple di uno stesso individuo.

Al di là dell’algoritmo di selezione, c’è sempre il rischio che la selezione vada a **sfavorire individui molto buoni**. Una soluzione è l’**Elitism** una sorta di “*fattore correttivo*” che salva un sottoinsieme di individui migliori (secondo i punteggi di valutazione) nella generazione successiva, **bypassando il processo di selezione**.

Questi individui prendono il nome di **elite**, che di solito non supera il 10% dell’intera popolazione. L’elite non deve partecipare né al crossover e né alla mutazione, ma ciò non ci vieta di verificare empiricamente i risultati.

A seconda dell’algoritmo di selezione, il mating pool può avere una dimensione diversa. Con algoritmi di **selezione con ricampionamento** (quelli in cui un individuo può passare più volte la selezione) il mating pool può essere grande quanto la popolazione.

Con algoritmi di **selezione senza ricampionamento** il mating pool deve essere necessariamente di dimensione inferiore. In tal caso avremmo una nuova generazione più piccola di quella precedente, andandosi via via a ridurre a mano a mano che l’algoritmo avanza.

Una soluzione sarebbe quella di effettuare una prima selezione di $|M| < |P|$ individui, generare gli $|M|$ figli, e rieffettuare una seconda selezione (anche con un diverso algoritmo) tra tutti i $2|M|$ individui. In questo scenario i figli non rimpiazzano automaticamente i genitori.

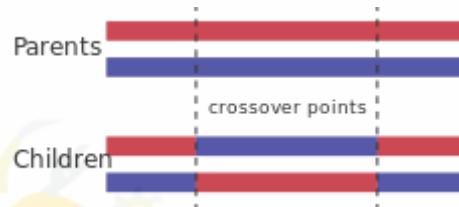
➤ Crossover

L'operatore di **crossover** rappresenta il primo passo per la creazione di nuove soluzioni per esplorare aree sconosciute dello spazio di ricerca.

Il **crossover** è l'operatore più sensibile alla codifica scelta per gli individui. Infatti, la decisione della codifica deve essere sempre valutata sulla base dell'esistenza di un operatore di *crossover sensato*. Qualora non dovesse esistere già un algoritmo di crossover noto, è comunque possibile definirlo ad-hoc per lo specifico problema.

Vediamo i principali operatori per individui codificati come array (di bit o di numeri):

- **Two-Point Crossover:** invece di scegliere un singolo punto di taglio se ne scelgono due. Questo crea una maggiore diversità, ed è possibile generalizzarlo a K punti. Un numero troppo alto non va bene perché potrebbe portare troppa diversificazione.
- **Uniform Crossover:** non si compie alcun taglio, ma il gene i -esimo di entrambi i figli sarà pari al gene i -esimo di uno dei due genitori. Dato che la scelta tra i due genitori è uniforme, si avranno figli con circa 50% del materiale del primo genitore e del secondo.



Il **crossover** non si applica ciecamente ad ogni coppia di genitori, ma bisogna sempre verificare una probabilità di crossover (p_c), decisa a priori. Questa scelta viene fatta per evitare di creare una popolazione di figli troppo diversa dai genitori, aumentando troppo l'*exploration* a scapito dell'*exploitation*.

Di solito questa probabilità è abbastanza alta, ad esempio, 0.8. Inoltre, se esiste un meccanismo che identifica la convergenza prematura, è possibile **modificarla dinamicamente** (Adaptive GA), aumentandola, per ridurne l'effetto.

Nella maggior parte dei casi il crossover avviene usando due genitori, ma esistono studi che hanno considerato il **multi-parent crossover**, che utilizza più di due individui per la generazione dei figli.

➤ Mutazione

L'operatore di **mutazione** è il secondo passo per la creazione di nuove soluzioni per esplorare aree sconosciute dello spazio di ricerca.

Sebbene in misura minore, la mutazione è sensibile alla codifica scelta per gli individui. Infatti, la decisione della codifica deve essere valutata anche in base dell'esistenza di un operatore di *mutazione sensato*. Qualora non dovesse esistere già una mutazione nota, è comunque possibile definirla ad-hoc per lo specifico problema.

La mutazione è considerata un “**meccanismo di sicurezza**” che permette al GA di sbloccarsi da situazioni di **convergenza prematura**. Il parametro di governo p_m indica la probabilità di mutazione di un gene ed è tendenzialmente basso (0.01) proprio per evitare di degenerare in una ricerca casuale. Come per il crossover, è possibile modificare questa probabilità dinamicamente in caso di **convergenza prematura**.

- **Random Resetting**: modificare casualmente un gene ad un altro valore.
- **Swap**: scambiare di posto due geni scelti casualmente.
- **Scramble**: selezionare una sottosequenza casuale di geni e la si permuta casualmente.
- **Inversion**: selezionare una sottosequenza casuale di geni e la si inverte.

➤ Crossover e Mutazione

Gli operatori di **crossover** e **mutazione** creano nuove soluzioni. I problemi di ottimizzazioni possono definire un insieme di vincoli, **constraints**, ovvero delle condizioni che le soluzioni devono necessariamente rispettare.

I vincoli, infatti, definiscono quel sottoinsieme del dominio che prende il nome di **regione ammissibile** (feasible region).

Esistono diversi metodi per rispettare i vincoli nonostante l'imprevedibilità del crossover e della mutazione:

- **Preservare Ammissibilità**: partendo da soluzioni ammissibili (feasible), si introducono degli accorgimenti affinché gli operatori di *crossover* e *mutazione* non creino mai soluzioni inammissibili (infeasible), quindi facendo in modo di mantenere sempre validi tutti i vincoli definiti.
- **Funzione di Penalità**: si modifica la funzione di valutazione aggiungendo una **componente di penalità**, che riduce il punteggio in base a “quanto si viola” un vincolo. Bisogna stabilire una misura che catturi il grado di violazione.
- **Favorire Soluzioni Ammissibili**: ipotizzando una *Tournament Selection*, le soluzioni ammissibili vincono automaticamente contro le inammissibili. Tra due soluzioni inammissibili vince quella con un numero minore di violazioni.

➤ Criteri di Arresto

I criteri di arresto sono tante condizioni booleane collegate tra loro in OR. Esistono diversi criteri di arresto:

- **Basate su Budget di Ricerca**: l'evoluzione termina se l'algoritmo spende tutte le **risorse** ad esso allocate. Le risorse possono essere:
 - **tempo di esecuzione**: l'evoluzione termina se l'algoritmo è in esecuzione da più di X secondi.
 - **funzione di Costo**: assegna a ciascun individuo un valore di costo che indica il “consumo” di quell'individuo rispetto ad una certa risorsa. Se si ottiene una popolazione la cui somma dei costi supera il costo massimo allocato X , l'evoluzione termina.
 - **numero di generazioni**: l'evoluzione termina se l'algoritmo ha creato più di X generazioni.
 - **numero Valutazioni**: l'evoluzione termina se l'algoritmo ha eseguito più di X volte la *funzione di valutazione*.
- **Basate su Criteri di Convergenza**: l'evoluzione termina se la popolazione ha raggiunto una situazione di **convergenza** da cui non è possibile uscirne. La convergenza può essere:

- **Fenotipica**: l'evoluzione termina se dopo X generazioni non si osservano miglioramenti rilevanti. Per valutare l'intera popolazione si può usare la valutazione media: i miglioramenti sono "rilevanti" se la differenza supera una soglia arbitraria ϵ .
- **Genotipica**: l'evoluzione termina se gli individui si somigliano per un $X\%$ (ovvero, $X\%$ dei loro geni sono perfettamente identici). Di solito X viene fissato a 90.
- **Test Ottimalità**: se è disponibile verifica il raggiungimento dell'ottimo, così da arrestarsi prima del tempo.

➤ Inizializzazione

Abbiamo visto empiricamente come la popolazione iniziale influenzi le prestazioni di ricerca di un GA. In particolare, quanto più la popolazione è diversificata tanto più è facile trovare buone soluzioni.

Spesso si crea la prima popolazione in maniera del tutto **casuale** ma esistono altre tecniche più sofisticate basate sull'uso di **metriche di diversità**, che fanno in modo che la popolazione iniziale inizia già con un set di individui ben diversi.

La metrica si può basare sulla **dispersione fenotipica** (ad esempio, deviazione standard delle valutazioni degli individui), altre sulla **dispersione genotipica** (ad esempio, il grado di somiglianza delle codifiche degli individui).

➤ Vincoli Popolazione

In genere un GA utilizza una **popolazione a taglia fissa**, determinata da un parametro arbitrario deciso a priori. Alcune implementazioni prevedono una **popolazione di taglia variabile** tra le generazioni ma sono più complesse da gestire.

Popolazioni troppo grandi danneggiano le prestazioni, mentre quelle piccole rischiano facilmente di convergere prematuramente.

Per evitare di danneggiare le prestazioni, si potrebbero imporre altri limiti alla popolazione oltre la taglia, ad esempio impostando un **limite di costo**.

5.5.4 Problemi Multi-Obiettivo

Per prendere le decisioni bisogna trovare un **trade-off**: cioè un compromesso, un equilibrio tra più obiettivi contrastanti.

La differenza tra **ottimizzazione mono-obiettivo** ed **ottimizzazione multi-obiettivo (MOO)** è nel numero di soluzioni ottime: i **MOO** prevedono un set di soluzioni ottimali non confrontabili tra loro.

Esistono due approcci per poter gestire i *problemi multi-obiettivo*:

1. Approccio Classico

Convertire il **problema multi-obiettivo** in un **problema mono-obiettivo** aggregando tutte le funzioni obiettivo in una singola tramite una **somma pesata**. Il vettore dei pesi viene stabilito su informazioni provenienti dal problema che si vuole risolvere. Questa è una tecnica semplice da realizzare poiché riusa gli ottimizzatori classici, ma:

- la scelta dei pesi è molto soggettiva;
- si perde il concetto di *trade-off*;
- le *funzioni obiettivo* possono essere su scala e ordini differenti.

2. Approccio Ideale

Si ottimizza il problema tenendo conto dei *trade-off* di tutte le funzioni obiettivo. L'ottimizzazione produce un **set di soluzioni ottimali**, dal quale, in un secondo momento, si identifica la soluzione che fa al caso nostro tramite informazioni provenienti dal problema.

Questa tecnica è più complessa da realizzare poiché richiede la definizione di nuovi tipi di ottimizzatori, ma supera i problemi evidenziati dall'approccio classico.

Qual è il problema principale della modellazione ideale con M obiettivi contrastanti?

Ciascun obiettivo, preso singolarmente, ha sicuramente un suo punto di ottimo, ma se consideriamo tutti gli obiettivi è chiaro che il concetto di ottimo deve essere riconsiderato.

Diremo quindi che una funzione $f(x)$ **domina** una funzione $f(y)$ poiché:

- **non è peggiore** rispetto a TUTTE le funzioni obiettivo;
- **è migliore** in ALMENO UNA funzione obiettivo.

Questa definizione definisce un **ordine parzialmente stretto** perché la relazione è:

- **Non riflessiva**;
- **Asimmetrica**;
- **Transitiva**.

Le soluzioni che non compaiono in alcun insieme dominato formano l'**insieme delle soluzioni non dominate** (non-dominated set). Questo insieme, a livello globale, è anche noto come **Fronte di Pareto**, che contiene tutte le soluzioni ottimali in *trade-off*. Le soluzioni nel Fronte sono dette **ottime secondo Pareto** (Pareto-optimal solutions).

A livello teorico lo scopo finale di un *ottimizzatore multi-obiettivo* sarebbe quello di trovare il **Fronte di Pareto** e restituire tutte le soluzioni in esso contenuto. Per realizzare un'ottimizzazione del genere bisogna:

1. valutare tutte le soluzioni rispetto a tutte le M funzioni obiettivo;
2. confrontare ogni soluzione con tutte le altre (stabilendo i link di dominanza);
3. le soluzioni mai dominate formano il *Fronte di Pareto*.

Questa soluzione presenta due problemi:

- Il primo riguarda il **carico computazionale**: è impensabile valutare tutte le soluzioni nello spazio di ricerca (è una ricerca esaustiva).
- Il secondo è il seguente: “abbiamo davvero bisogno di **TUTTO** il *Fronte di Pareto*? ”

Ci occorre, quindi, una strategia in grado di:

- ottenere soluzioni diversificate, pur non necessariamente *Pareto-optimal*;
- evitare la ricerca esaustiva per avere soluzioni in tempi accettabili.

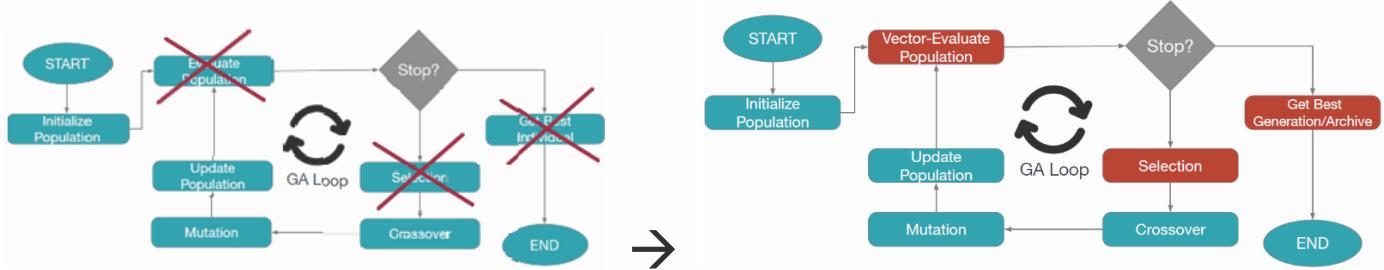
Sarebbe necessario, quindi, adottare una variante di *Hill Climbing* (in grado di restituire una singola soluzione sub-ottimale, non un insieme), eseguendolo più volte da diversi punti dello spazio di ricerca. I risultati di tutte le esecuzioni formerebbero un insieme di soluzioni vicino al *Fronte di Pareto*.

Questa è un'idea molto semplice, ma non è molto efficiente (per via delle molteplici esecuzioni) e comunque non ci garantisce molta diversità tra le soluzioni identificate.

Un'altra soluzione è di utilizzare un **algoritmo genetico** poiché nella sua ultima iterazione dispone già di un insieme di soluzioni sub-ottimali e ben diversificate.

GA per MOO

Per avere un GA modellato sui problemi multi-objettivo dobbiamo modificare alcune componenti di un classico SGA:



~~Get Best Individual~~ → Get Best Generation/Archive

Non siamo più interessati ad una singola soluzione, ma ad un insieme. Quindi possiamo scegliere tra 2 soluzioni:

1. **Best Generation:** restituiamo direttamente l'ultima generazione ottenuta dal GA loop (o la migliore ultima).
2. **Archive:** costruiamo una popolazione “non-evolvente” (cioè fissa, costante) fuori dal loop che contiene i migliori individui di ciascuna generazione. Ad ogni iterazione prendo il miglior individuo e lo aggiungo all'archivio, però, se il miglior individuo ottenuto ad una certa iterazione domina un altro già presente nell'archivio, allora prende il suo posto. Al termine del GA loop, si restituisce l'archivio.

~~Evaluate Population~~ → Vector-Evaluate Population

Non avremmo solo **M funzioni obiettivo**, ma anche **M funzioni di valutazione**. Ogni individuo riceverà un vettore di valutazione. Si avrà un forte impatto sulle prestazioni.

~~Selection~~ → Selection

La **selezione** risulta il nodo più difficile da sciogliere. Infatti, occorre un metodo di ordinamento degli individui all'interno della popolazione; per fare ciò esistono diverse soluzioni:

- ❖ **Soluzione di VEGA:** Dato un GA con M funzioni di valutazione, si divide la popolazione casualmente in M sotto-popolazioni. Ciascuna utilizzerà una delle M funzioni di valutazione per ottenere la funzione di fitness e compiere la selezione internamente alla sotto-popolazione.
E' una soluzione molto semplice, ma tende a preferire troppo le soluzioni ottimali rispetto alle singole funzioni obiettivo, allontanandosi troppo dal fronte di Pareto. In altre parole, è vero che si ottiene molta diversità, ma si perde il *trade-off*.
- ❖ **Soluzione di MOGA:** ad ogni individuo x si assegna un rango pari al “numero di individui che dominano $x” + 1. La funzione di fitness va a mappare questi ranghi in una probabilità di sopravvivenza. Questo implica che: le soluzioni nel fronte di Pareto avranno tutte rango 1. Non è detto che saranno assegnati tutti i ranghi tra 1 e $|P|$.$
- ❖ **Soluzione di NSGA:** si basa sul concetto di **non-dominated set**, e consiste nel creare un ordinamento di tutti i **non-dominated set** in diversi livelli di non-dominanza tramite una procedura iterativa, in una maniera simile al ranking applicato in MOGA. L'algoritmo di ordinamento compie i seguenti passi:
 1. si valuta ogni individuo nella popolazione (ciascuno avrà un vettore di valutazione);
 2. si inizializza il contatore di livello: $j = 1$;

3. si confrontano tra loro tutti gli individui, allo scopo di trovare gli individui non dominati, aggiunti all'insieme P_j ;
4. si memorizza all'interno degli individui il valore j ;
5. si rimuove dalla popolazione P tutti gli individui presenti in P_j ed incrementa j di 1;
6. si va al punto 3 se la popolazione P non è vuota;
7. si restituiscono tutti gli insiemi P_j .

Tra due individui si favorisce quello di **livello minore** poiché è più vicino al vero Fronte di Pareto.

Se invece siamo di fronte a due individui appartenenti allo stesso livello allora abbiamo due soluzioni:

- 1. Soluzione banale:** si seleziona casualmente un fronte con probabilità analoga alla **Rank Selection** ($\text{prob}(l) = (\text{numeroLivelli} - l + 1) / \sum l$), per poi selezionare in maniera uniforme un individuo all'interno del fronte selezionato. In altre parole, nessun individuo all'interno dello stesso fronte è migliore di un altro.
Inoltre, la probabilità di selezionare due o più individui con trade-off simile è molto più alta rispetto alla probabilità di selezionare individui con trade-off diversi.
- 2. Soluzione intelligente:** si favoriscono gli individui più “isolati dalla folla”, tramite una misura di distanza chiamata **crowding distance**. Si opera nel seguente modo; per ogni fronte:
 1. si valuta ogni individuo rispetto a tutte le funzioni di valutazione;
 2. per ogni funzione di valutazione f_i :
 - a. si ordinano gli individui rispetto al valore f_i ;
 - b. si assegna al primo e all'ultimo individuo una **distanza infinita**;
 - c. si itera sugli altri individui, e si assegna a ciascuno una distanza pari alla differenza delle valutazioni f_i delle due soluzioni adiacenti (nell'ordinamento), normalizzata sulle f_i estreme (del primo e dell'ultimo individuo).
 3. per ogni individuo, si sommano tutte le distanze ricevute, ottenendo così la **crowding distance**.
- 3. Soluzione parziale:** si introduce un criterio di preferenza (obiettivo secondario problem-specific) per evitare di avere soluzioni ambigue equivalenti, ad esempio nel trade-off tra comfort e costo delle auto potremmo considerare l'anno di costruzione dell'auto.

5.5.5 Varianti dei GA

Ci sono alcune varianti che portano gli algoritmi genetici ad essere potenziati o a risolvere nuove classi di problemi:

Standy-State GA

Anche noto come **Genitor-style GA**. Invece di creare nuove generazioni modifica continuamente la stessa con miglioramenti graduali. Infatti:

- vengono scelti i migliori due individui (selezione basata su ranghi) come genitori;
- con il crossover si generano un solo figlio, soggetto poi a mutazione;
- il figlio è valutato e andrà, eventualmente, a rimpiazzare il peggior individuo.

Evolve “lentamente” ma garantisce un miglioramento monotono.

Parallel GA

I GA sono efficienti implementazioni parallele. Vediamo le 3 principali:

- **Global Parallelism**: con una popolazione di taglia N (pari), $N/2$ processori, ed una selezione di tipo *Tournament Selection*, applichiamo i seguenti passi:
 1. si distribuiscono le $N/2$ coppie di individui a ciascun processore;
 2. ciascun processore valuta i due individui ed applica la selezione;
 3. si ripetono i passi 1 e 2 per una seconda volta;
 4. ciascun processore applica il crossover e la mutazione dei due figli;
 5. si raccolgono i figli da ciascun processore, formando la nuova generazione.
- **Island Model GA**: con una popolazione di taglia N (molto grande) ed M processori possiamo assegnare a ciascun processore una sotto-popolazione di N/M individui: ciascun processore esegue un qualsiasi GA (SGA, Steady-State GA, ecc.) in maniera indipendente dalle altre (rappresenta quindi "un'isola"). Per evitare **derive genetiche** (quando delle popolazioni piccole tendono ad evolvere verso una direzione specifica, non necessariamente ottimale), ogni K generazioni le sotto-popolazioni si scambiano alcuni individui: si applica la cosiddetta **migrazione**.
- **Cellular GA**: con una popolazione di taglia N (molto grande), si crea una matrice $\sqrt{N} \times \sqrt{N}$ di individui. Ogni elemento (**cella**) viene assegnato ad un singolo processore. Nella variante standard, ogni cella può comunicare soltanto con le celle adiacenti (nord, sud, est, ed ovest, in modulo). Ad ogni iterazione, ogni cella valuta il proprio individuo, si accoppia con l'individuo vicino più forte si genera un singolo figlio, lo si muta, e si rimpiazza il genitore se risulta migliore. Gruppi molto distanti di celle formano delle sotto-popolazioni (isolamento per distanza).

Algoritmi Memetici

Si introduce della ricerca locale all'interno del *GA loop* (ibridizzazione) allo scopo di ridurre il rischio di convergenza prematura. In particolare, dopo aver applicato la mutazione alla nuova generazione di individui si seleziona un subset casuale degli individui che sarà sottoposto ad una fase di ricerca locale, modificandosi in meglio. Questa fase di raffinamento locale rappresenta un'**evoluzione culturale**.

Algoritmi di questo tipo sono anche noti come **Algoritmi Evolutivi Lamarckiani**, che ricalcano la teoria evoluzionistica di Lamarck, che afferma che le caratteristiche utilizzate da un individuo si trasmettono ai figli, cosa non vera secondo Darwin.

Gli algoritmi **memetici co-evolutivi** considerano i meme insieme alla componente genetica di un individuo. La **componente memetica** codifica il miglioramento ottenuto dall'ottimizzazione locale, e partecipa anche essa all'evoluzione. Infatti, contribuisce alla funzione di valutazione ed è viene trasmessa alla prole, in maniera analoga al crossover genetico.

Capitolo 6 – Algoritmi di Ricerca con Avversari

6.1 Ambienti Multi-Agente

In **ambienti multi-agente** ogni agente deve considerare le azioni degli altri ma anche gli effetti di tali azioni, per poter quantificare il proprio benessere.

Invece, in **ambienti competitivi** gli obiettivi degli agenti sono in conflitto. Questo origina i problemi di **ricerca con avversari**, i quali vengono tipicamente indicati come *giochi*.

La **teoria dei giochi** è una branca dell'economia che considera ogni ambiente *multi-agente* come un gioco, indipendentemente dal fatto che l'interazione sia cooperativa o competitiva, a patto che l'influenza di ogni agente sugli altri sia significativa.

Gli ambienti con quantità molto grande di agenti sono spesso considerate **economie** piuttosto che *giochi*. Esistono diverse tipologie di giochi, classificati principalmente in 2 modi:

Classificazione 1: Condizioni di Scelta

Giochi con informazione “perfetta”. Giochi in cui gli stati del gioco sono completamente espliciti agli agenti.

Giochi con informazione “imperfetta”. Giochi in cui gli stati del gioco sono solo parzialmente esplicitati.

Classificazione 2: Effetti della Scelta

Giochi deterministici. Giochi in cui gli stati del gioco sono determinati unicamente dalle azioni degli agenti.

Giochi stocastici. Giochi in cui gli stati del gioco sono determinati anche da fattori esterni (ad esempio, tutti i giochi in cui c'è un dado).

Altre classificazioni tengono conto della composizione degli ambienti, ad esempio se questi sono discreti, statici, episodici, ed altro.

6.2 Teoria dei Giochi: Ambiente

I giochi più comuni sono quelli che vengono definiti a **somma zero** e con **informazione perfetta**. Questi sono i giochi in cui due giocatori, a turno, effettuano azioni che influenzano l'ambiente così come il comportamento dell'altro giocatore.

Nella nostra terminologia, parliamo di **ambienti multi-agente**, **deterministici** e **completamente osservabili**, in cui due agenti agiscono alternandosi e in cui i valori di utilità, alla fine della partita, sono sempre uguali ma di segno opposto.

Ad esempio, consideriamo il caso degli scacchi. Se un giocatore vince, l'altro deve necessariamente perdere. Questa opposizione ci fa parlare di “avversari”.

I giochi richiedono l'abilità di prendere una decisione quando il calcolo di una soluzione ottima non è realizzabile o comunque ne penalizzerebbe l'efficienza.

6.2.1 Teoria dei Giochi: Definizione

Più formalmente, possiamo definire un gioco come un **problema di ricerca** con i seguenti componenti:

- **s_0** : lo **stato iniziale**, che specifica come è configurato il gioco in partenza;
- **GIOCATORE (s)**: definisce il giocatore a cui tocca fare una mossa nello stato s ;
- **AZIONI (s)**: restituisce l'insieme delle mosse lecite in uno stato s ;
- **RISULTATO (s, a)**: il modello di transizione, che definisce il risultato di una mossa;

- **TEST-TERMINAZIONE (s)**: un test di terminazione, che restituisce ‘**vero**’ se la partita è finita e ‘**falso**’ altrimenti. Gli stati che fanno terminare una partita sono detti “**stati terminali**”.
- **UTILITÀ (s, p)**: una funzione utilità, chiamata anche funzione obiettivo o **funzione di payoff**, che definisce il valore numerico finale per un gioco che termina nello stato terminale s per un giocatore p. In altri termini, la funzione dà il punteggio finale da assegnare ai giocatori.

Lo *stato iniziale*, la funzione *AZIONI* e la funzione *RISULTATO* definiscono l'**albero di gioco**, un albero in cui i nodi sono stati del gioco e gli archi le mosse.

6.2.2 Teoria dei Giochi: Decisione Ottime

In un **normale problema di ricerca**, la soluzione ottima è costituita da una sequenza di mosse che portano ad uno stato obiettivo.

In una **ricerca con avversari**, ogni giocatore (MIN e MAX) può “*dire la sua*”. MAX dovrà quindi trovare una strategia che specifichi la sua mossa nello stato iniziale, così come le altre mosse in tutti gli stati possibili risultanti dalla prima mossa di MIN, e così via.

Una **strategia ottima** porta ad un risultato che è almeno pari a quello di qualsiasi altra strategia, assumendo che si stia giocando contro un giocatore infallibile.

Dilemma del Prigioniero

Dilemma del Prigioniero. Due criminali vengono accusati di aver commesso di avere un porto d’armi abusivo e per questo, gli investigatori li arrestano e li chiudono in due celle diverse, impedendo loro di comunicare. Ad ognuno di loro vengono date due scelte: *collaborare*, oppure *non collaborare*.

Ai due criminali viene spiegato che:

1. Se solo uno dei due collabora accusando l’altro, chi ha collaborato evita la pena; l’altro viene però condannato a **7 anni** di carcere;
2. Se entrambi accusano l’altro, vengono entrambi condannati a **6 anni**.
3. Se nessuno dei due collabora, entrambi vengono condannati a **1 anno**, perché comunque già colpevoli di porto abusivo di armi.

Il problema può essere descritto tramite una matrice 2x2:

	Collabora	Non Collabora
Collabora	(6, 6)	(0, 7)
Non Collabora	(7, 0)	(1, 1)

La miglior strategia di questo gioco **non cooperativo** è (**collabora, collabora**) perché un prigioniero non sa cosa sceglierà di fare l’altro. La scelta di collaborare è, quindi, la migliore possibile per uno dei due, in assenza di informazioni su cosa farà l’altro.

Questo perché per ognuno dei due lo scopo è quello di **minimizzare** la propria condanna. Quindi, ogni prigioniero:

Collaborando	Rischia da 0 a 6 anni;
Non Collaborando	Rischia da 1 a 7 anni;

Supponiamo che i due si siano promessi di non collaborare in caso di arresto. Sono ora rinchiusi in due celle diverse e si chiederanno se la promessa sarà mantenuta dall'altro; se un prigioniero non rispetta la promessa e l'altro sì, il primo è allora liberato. C'è dunque un **dilemma**: *collaborare o non collaborare*. La teoria dei giochi ci dice che c'è un solo equilibrio (**collabora, collabora**).

Equilibrio di Nash. Questo equilibrio, ovvero quello (*collabora, collabora*) è chiamato equilibrio di Nash. La definizione di equilibrio di Nash ha avuto negli anni implicazioni non solo matematiche/economiche, ma persino filosofiche.

La scelta migliore per la ‘società’ composta dai due prigionieri sarebbe di non parlare, ma questa scelta **non è un equilibrio di Nash**: infatti uno qualsiasi dei due, se fosse il solo a parlare, potrebbe migliorare la sua situazione. **Paradossalmente, l'unico equilibrio di Nash di questo esempio è la situazione peggiore per la ‘società’, quella in cui entrambi parlano prendendo 6 anni a testa.**

Dal punto di vista del singolo prigioniero, infatti:

- Se il suo compagno non ha parlato, parlando è libero invece di essere condannato ad un anno;
- Se il suo compagno ha parlato, parlando è condannato a 6 anni invece che a 7;

A ciascuno dei due prigionieri **conviene parlare**; dal punto di vista dell’equilibrio di Nash è irrilevante il fatto che se un prigioniero parla peggiora la situazione dell’altro.

Secondo la teoria economica, detta ‘*classica*’, di Adam Smith, se ogni componente di un gruppo segue il proprio interesse personale, ogni azione individuale aumenta la ricchezza complessiva del gruppo.

La teoria dei giochi dimostra che non è così. Se ogni componente del gruppo fa ciò che è meglio per sé, il risultato finale è, in generale, un **equilibrio di Nash** che però non rappresenta la soluzione migliore per la ‘società’ e può inoltre rappresentare un’allocazione inefficiente delle risorse.

Ottimo Paretiano. Wilfredo Pareto fu un famoso economista, noto per alcuni fondamentali concetti come quello di **ottimo paretiano** (vedi algoritmi genetici). Si ha un ottimo paretiano quando si è raggiunta una situazione in cui non è possibile migliorare la condizione di una persona senza peggiorare la condizione di un'altra. L’ottimo paretiano si raggiunge se si collabora, a differenza dell’equilibrio di Nash, anche se nessuno accetta di peggiorare anche di poco la propria situazione.

Nel caso del *dilemma del prigioniero* si hanno **tre ottimi paretiani**: il caso in cui nessuno dei due prigionieri accusa l’altro (1,1) ed i due casi in cui solo A o solo B accusa l’altro (0,7 e 7,0). Se ci si trova nella situazione in cui entrambi accusano l’altro, infatti, è possibile ottenere un vantaggio per entrambi se tutti e due ritirano l’accusa.

Nelle due situazioni intermedie, invece, se l’unico prigioniero che ha accusato l’altro ritirasse l’accusa avrebbe un danno, passando dalla libertà ad un anno di carcere.

Questi sono ‘*ottimi di Pareto*’ perché non è possibile migliorare la situazione dell’uno senza peggiorare quella dell’altro e rappresentano una allocazione ottimale delle risorse. Un ‘*ottimo di Pareto*’ non è però necessariamente la situazione migliore per la ‘società’.

Le strategie egoistiche non portano ad una soluzione ottimale per la società né se non ci si cura di quanto succede agli altri (come nell’equilibrio di Nash) ma neppure se si è disposti a collaborare a patto di non avere svantaggi (come nella ricerca dell’ottimo di Pareto).

È possibile che per ottenere la situazione migliore si debba imporre a qualcuno di rinunciare a qualche cosa: in una società evoluta questo è il ruolo della legge.

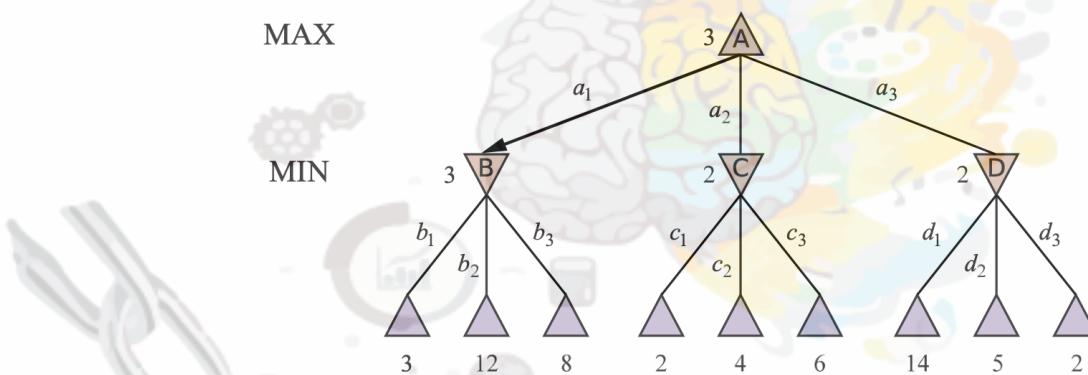
Dinamiche Dominanti. Parliamo di dinamiche dominanti quando un giocatore fa il meglio che può indipendentemente da ciò che farà l'altro.

6.3 Algoritmo Minimax

L'**algoritmo minimax** è un algoritmo ricorsivo, il quale analizza l'albero dal basso verso l'alto. È finalizzato nel cercare la mossa migliore in un gioco a somma zero che si svolge tra due giocatori:

- **Mini**: è il giocatore che deve minimizzare il valore di gioco.
- **Max**: è il giocatore che deve massimizzare il valore di gioco.

Le mosse possibili per **Max** nel nodo radice sono etichettate con a_1, a_2 e a_3 , mentre le possibili risposte di **Mini** con b_1, b_2 e b_3 e così via. Questo gioco termina dopo una sola mossa di **Max** ed una di **Mini**. I valori di utilità degli stati terminali vanno da 2 a 14.



Dato un albero di gioco, la strategia ottima può essere determinata analizzando il valore minimax di ogni nodo, che scriveremo **MINIMAX(n)**.

Valore Minimax. Il valore minimax di un nodo corrisponde all'utilità di trovarsi in un determinato stato, assumendo che entrambi gli agenti giochino in modo ottimo da lì alla fine della partita.

MINIMAX(n)

$$= \begin{cases} UTILITA'(s, MAX) & se TEST - TERMINALE(s); \\ max_{a \in AZIONI(s)} VALORE - MINIMAX - RISULTATO(s, a) & se GIOCATORE(s) = MAX; \\ min_{a \in AZIONI(s)} VALORE - MINIMAX - RISULTATO(s, a) & se GIOCATORE(s) = MIN; \end{cases}$$

```
function DECISION-MINIMAX(s) returns un'azione
    return argmaxa ∈ AZIONI(s) VALORE-MIN(RISULTATO(s, a))
```

```
function VALORE-MAX(s) returns un valore di utilità
    if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
    v ← -∞

    for each a in AZIONI(s) do
        v ← MAX(v, VALORE-MIN(RISULTATO(s, a)))

    return v
```

```
function VALORE-MIN(s) returns un valore di utilità
    if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
    v ← +∞

    for each a in AZIONI(s) do
        v ← MIN(v, VALORE-MAX(RISULTATO(s, a)))

    return v
```

Le funzioni **VALORE-MIN** e **VALORE-MAX** non fanno altro che attraversare l'intero albero di gioco fino alle foglie per determinare il valore da "far risalire".

Se troviamo un nodo foglia, restituiamo il valore utilità della foglia. Altrimenti, iteriamo sulle azioni e, per ognuna di esse, cerchiamo il valore minimax.

Per **MAX**, questo sarà dato dal valore massimo contenuto nel sotto-albero di livello inferiore, il quale è costruito tramite la funzione **VALORE-MIN**.

Per **MIN**, il discorso è simile. Cercheremo il valore minimo del sotto-albero costruito con la funzione **VALORE-MAX**.

La notazione **argmax** calcola il valore massimo del sotto-albero di livello inferiore.

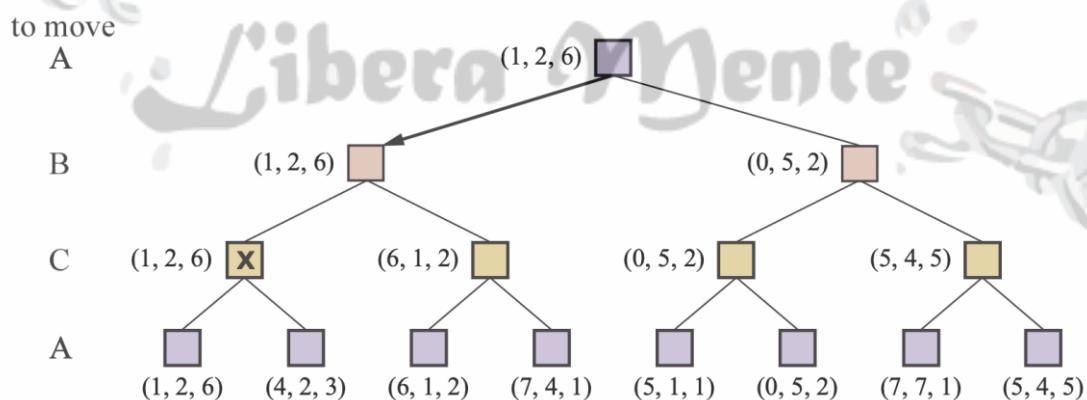
Proprietà

Verifichiamo le proprietà di questo tipo di strategia:

- **Completezza**: l'algoritmo è completo se l'albero è finito.
- **Complessità temporale**: la complessità è governata dalla dimensione dell'albero di gioco. Sia **m** la profondità massima, nel caso pessimo la ricerca genererà $O(b^m)$ nodi.
- **Ottimalità**: l'algoritmo è ottimo nel caso in cui l'avversario è ottimo.
- **Complessità spaziale**: la ricerca deve memorizzare un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi per ciascun nodo sul cammino. Una volta che un nodo è stato espanso, può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente. Per un albero di gioco con fattore di ramificazione **b** e profondità **m**, la complessità sarà di $O(bm)$.

Algoritmo Minimax in giochi multi-player

L'algoritmo minimax può essere adattato e utilizzato anche in casi in cui al gioco partecipano più giocatori. Altre complicazioni, a livello di design, consistono nell'esistenza di **alleanze** tra giocatori. Infatti, tali alleanze fanno parte della strategia ottima e dovrebbero quindi essere modellate all'interno dell'albero di gioco. Prendiamo, ad esempio, un gioco con tre giocatori: A, B e C.



- Prima di tutto, ad ogni nodo dovremo associare un vettore e non più un singolo valore minimax.
- Per gli **stati terminali**, il vettore fornirà l'utilità dello stato dal punto di vista di ogni giocatore.
- Per gli **stati non terminali**, il discorso cambia. Prendiamo il nodo X: in questo stato il giocatore C deve decidere cosa fare. Le due possibili scelte portano ad un guadagno di 6 e 3 e, pertanto C dovrebbe scegliere la prima mossa.

Questo porta ad un vettore minimax che “danneggia” gli altri giocatori. Tuttavia, è questo il modo di ragionare corretto: il valore passato verso l'alto è il vettore di utilità dello stato successore più favorevole a chi sta scegliendo la mossa.

Complessità temporale: Potatura

Analizziamo la **complessità temporale** dell'*algoritmo Minimax*: $O(b^m)$. Sfortunatamente non possiamo eliminare l'esponente, ma possiamo almeno dimezzarlo: infatti è possibile calcolare la decisione minimax corretta senza guardare tutti i nodi dell'albero di gioco.

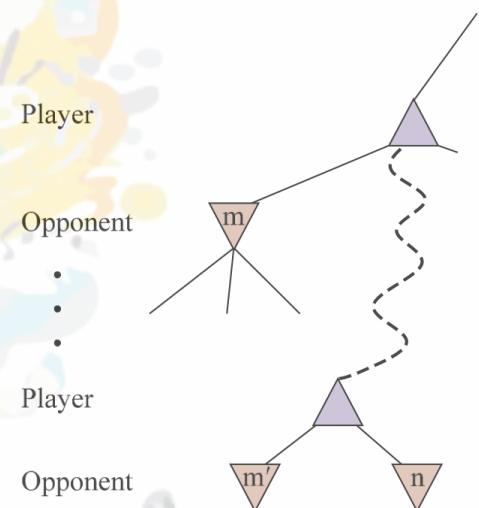
In questo caso parleremo di **potatura**, ovvero la tecnica che consente di evitare di prendere in considerazione grandi porzioni dell'albero. La tecnica che analizzeremo è chiamata **potatura alfa-beta**, che applicata ad un albero minimax restituisce lo stesso risultato della tecnica minimax standard, ma elimina (pota) i rami che non servono per la decisione finale.

6.3.1 Algoritmo Minimax con Potatura Alfa-Beta

La **potatura alfa-beta** può essere applicata ad alberi di qualunque profondità e spesso è possibile potare interi sotto-alberi, rendendo la potatura ancora più efficace.

Il principio generale è questo:

- consideriamo un nodo n da qualche parte dell'albero tale che il giocatore abbia la possibilità di spostarsi in quel nodo.
- Se esiste una scelta migliore m a livello del nodo padre o di qualunque nodo precedente, allora n non sarà mai raggiunto in tutta la partita.
- Di conseguenza, possiamo potare n non appena abbiamo raccolto abbastanza informazioni per poter giungere a questa conclusione.



Alfa. Il valore della scelta migliore (quella con il valore più alto) per **MAX** che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.

Beta. Il valore della scelta migliore (quella con il valore più basso) per **MIN** che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.

Pseudocodice

```
function RICERCA-ALFA-BETA(s) returns un'azione
    v <- VALORE-MAX(s, -∞, +∞)
    return l'azione in AZIONI(s) con valore v

function VALORE-MAX(s) returns un valore di utilità
    if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
    v <- -∞

    for each a in AZIONI(s) do
        v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
        if v ≥ beta then return v
        alfa <- MAX(alfa, v)

    return v

function VALORE-MIN(s) returns un valore di utilità
    if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
    v <- +∞

    for each a in AZIONI(s) do
        v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
        if v ≤ alfa then return v
        beta <- MIN(beta, v)

    return v
```

- La procedura di base è la stessa dell'algoritmo standard. Aggiungiamo però il codice necessario ad aggiornare alfa e beta e il codice necessario per passare alfa e beta da una procedura ad un'altra.
- Lo stesso vale per la funzione VALORE-MIN.
- Inoltre, l'algoritmo, nel caso di MAX, stoppa l'esecuzione quando il valore dello stato corrente è maggiore o uguale all'attuale beta (abbiamo trovato uno stato migliore, pertanto, non è conveniente proseguire l'esplorazione).

Ordinamento delle mosse

L'efficacia della **potatura alfa-beta** dipende molto dall'ordinamento in cui gli stati sono esaminati. Quindi, potrebbe essere una buona idea quella di esaminare per prima i successori più promettenti usando una strategia best-first. In questo modo, la *ricerca alfa-beta* dovrà esaminare "solo" $O(b^{m/2})$ nodi, piuttosto che $O(b^m)$.

Ordinamento dinamico delle mosse

Un modo per ottenere informazioni dalla mossa corrente è quello di utilizzare una **ricerca ad approfondimento iterativo**: con questa strategia, il limite viene incrementato progressivamente, finché un nodo obiettivo non è identificato.

Con questa strategia, si cerca innanzitutto uno strato in profondità e si registra il miglior cammino di mosse. Poi si cerca uno strato ancora in profondità, ma si utilizza il cammino registrato allo scopo di fornire informazioni per l'ordinamento delle mosse.

Le mosse migliori sono spesso chiamate **mosse killer** e si parla di **euristica della mossa killer** quando queste vengono provate per prime. Inoltre, la ricerca ad approfondimento iterativo per l'ordinamento delle mosse è analoga a quella in ampiezza, poiché ad ogni iterazione l'algoritmo esplora completamente un livello di nodi prima di prendere in considerazione il successivo.

Quindi, possiamo esplorare i nodi in ampiezza e decidere come variare l'ordinamento in base ai valori **minimax**.

Un altro modo è quello di considerare le **trasposizioni**, che portano l'albero di gioco a contenere **cammini ridondanti**: stati ripetuti più volte nello spazio degli stati che possono determinare un aumento esponenziale del costo di ricerca.

Quindi, è consigliato memorizzare in una **tabella hash** una configurazione che viene incontrata per la prima volta. La tabella viene chiamata **tabella delle trasposizioni** ed è identica alla *lista esplorati* degli algoritmi di ricerca non informata.

Proprio perché identica alla lista esplorati, possiamo decidere di ordinare le mosse successive sulla base della conoscenza già acquisita esplorando una trasposizione.

Dall'altra parte, se l'albero di gioco avesse milioni di stati, si arriverà a dover mantenere in tabella solo alcune delle trasposizioni. Esistono diverse **strategie** per selezionare le trasposizioni da mantenere in memoria. Una strategia potrebbe essere derivata tramite la definizione di una funzione euristica.

Decisione imperfette in tempo reale

L'**algoritmo minimax** genera l'intero spazio di ricerca, mentre quello **alfa-beta** ci permette di poterne buona parte.

Tuttavia, anche **alfa-beta** deve condurre la ricerca fino agli stati terminali però questa profondità risulta non gestibile, perché le mosse devono essere calcolate in un tempo ragionevole.

Per questo, Claude Shannon propose che i programmi "tagliassero" la ricerca prima di raggiungere le foglie applicando una funzione di valutazione euristica agli stati. In questo modo, i nodi non terminali diventano foglie.

Il *minimax* viene modificato in due punti:

1. La **funzione di utilità** viene sostituita da **EVAL**, che fornisce una stima dell'utilità della posizione raggiunta.
2. Il **test di terminazione** viene sostituito con un test di taglio, **cutoff test**, che decide quando applicare **EVAL**.

Quindi, la nuova formulazione porta il seguente problema:

$$MINIMAX(n) = \begin{cases} EVAL(s) & \text{se } TEST - TAGLIO(s); \\ \max_{a \in AZIONI(s)} H - MINIMAX(RISULTATO(s, a), d + 1) & \text{se } GIOCATORE(s) = MAX; \\ \min_{a \in AZIONI(s)} H - MINIMAX(RISULTATO(s, a), d + 1) & \text{se } GIOCATORE(s) = MIN; \end{cases}$$

Una *funzione di valutazione* restituisce una stima del guadagno atteso in una determinata posizione, esattamente come le *funzioni euristiche*.

1. Funzione di valutazione

La **funzione di valutazione** dovrebbe:

1. ordinare gli stati terminali nello stesso modo della funzione di utilità;
2. essere veloce da calcolare (altrimenti, non avrebbe senso sostituire la funzione di utilità);
3. per gli stati non terminali dovrebbe avere una forte correlazione con la probabilità reale di vincere la partita.

La maggior parte delle *funzioni di valutazione* si basano sulle **caratteristiche di uno stato**: ad esempio, negli scacchi avremo come caratteristiche il numero di pedoni bianchi, il numero di pedoni neri, di regine bianche, di regine nere, e così via.

Le **caratteristiche** definiscono le **classi di equivalenza**, cioè un insieme di stati aventi lo stesso valore per tutte le caratteristiche.

2. Test di terminazione

Nella definizione del **test di taglio**, dobbiamo fare attenzione ai cosiddetti **stati non quiescenti**: stati che portano ad una variazione del valore delle mosse.

Supponiamo che, in una partita a scacchi, tocca al bianco muovere mentre il nero è in vantaggio, poiché ha un cavallo e due pedoni in più rispetto al bianco.

Il programma assocerebbe questo stato ad un valore euristico molto positivo, ritenendo che la situazione porterà ad una vittoria. Ma la mossa successiva del bianco cattura la regina nera, portandolo in una situazione di vittoria quasi certa. Questo è uno stato non quiescente, che porta ad un cambio netto dello stato di gioco.

La *funzione di valutazione* dovrebbe perciò essere applicata solo a **posizioni quiescenti**, ovvero quelle per cui sia improbabile il verificarsi di grandi variazioni al valore nelle mosse successive.

Talvolta, potrebbe essere inclusa nella definizione dell'algoritmo una ricerca di quiescenza, in cui stati non quiescenti vengono espansi fino a raggiungere posizioni quiescenti.

```

function VALORE-MAX(s) returns un valore di utilità
  if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
  if TEST-TAGLIO(s) then return EVAL(s)
  v <-  $-\infty$ 

  for each a in AZIONI(s) do
    v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
    if v  $\geq$  beta then return v
    alfa <- MAX(alfa, v)

  return v

```

L'eventuale valutazione della quiescenza di *s* sarà a carico della funzione TEST-TAGLIO.

6.3.2 Algoritmo Minimax con Potatura Alfa-Beta: miglioramenti

Potatura in avanti. Si esplorano solo alcune mosse ritenute promettenti, tagliando le altre.

Un approccio è la **beam search**: di volta in volta, si considerano solo le prime *k* migliori mosse, definite sulla base della **funzione di valutazione**. Tuttavia, non garantisce la non potatura delle mosse migliori.

Un secondo approccio sono i **tagli probabilistici** basati sull'esperienza: questo utilizza statistiche ricavate dalle precedenti esperienze per ridurre la possibilità che la mossa migliore venga potata.

Database di mosse di apertura e chiusura. L'idea è che all'inizio di una partita ci sono poche mosse sensate e ben studiate, per cui è inutile esplorarle tutte. In maniera simile, per le fasi finali potrebbe essere possibile predisporre delle chiusure così da non avere alcuna necessità di esplorare lo spazio.

Esempio:

I giochi di scacchi utilizzano tipicamente un database di questo tipo. All'inizio della partita, utilizzeranno mosse che sono *"storicamente buone"*, ovvero definite sulla base dell'osservazione dei più grandi scacchisti del mondo. Alla fine, avranno una maggior capacità di computazione rispetto all'uomo, essendo per questo capaci di definire delle strategie più efficaci/efficienti.

6.3.3 Dai giochi deterministici ai giochi stocastici

Possiamo complementare gli algoritmi visti finora introducendo dei **nodi possibilità** accanto a quelli di **scelta**. In questo modo, per calcolare i valori di MIN e MAX dovremo tener conto delle probabilità dell'esperimento casuale.

I **nodi terminali**, quelli per cui è già noto il risultato dell'evento stocastico, saranno invece calcolati nello stesso modo visto precedentemente.

Un **albero di gioco stocastico** è rappresentato in questo modo:

- i **nodi possibilità** sono visualizzati come cerchi;
- i **nodi di scelta** come triangoli.
- i **nodi terminali** sono calcolati secondo la regola standard dell'algoritmo minimax.

Esempio:

Sappiamo che MIN con probabilità **0.9** farà **2** e con probabilità **0.1** farà **3**.

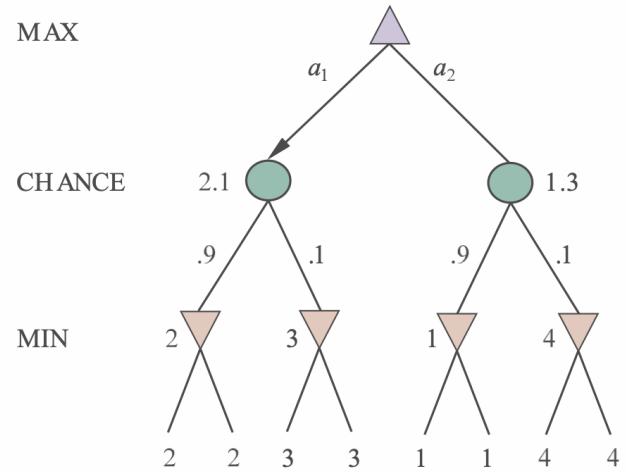
Dall'altro lato, con probabilità **0.9** farà **1** e con probabilità **0.1** farà **4**.

Quindi, deriviamo i valori attesi:

$$0.9 \times 2 + 0.1 \times 3 = 2.1$$

$$0.9 \times 1 + 0.1 \times 4 = 1.3$$

Di conseguenza, la mossa migliore per MAX è a_1 .



6.3.4 Dai giochi deterministici ai giochi parzialmente osservabili

In questa categoria di giochi, l'incertezza sullo stato di gioco nasce dall'impossibilità di accedere alle scelte fatte dall'avversario: ad esempio, i giochi di carte in cui le carte iniziali dell'avversario non sono note.

Potremmo pensare a questi giochi come un caso particolare dei *giochi stocastici*. Sebbene l'analogia non sia corretta, può servire per definire un algoritmo efficace.

Questo equivale a calcolare:

$$\operatorname{argmax}_s P(s) \operatorname{MINIMAX}(\operatorname{RESULT}(s, a))$$

Purtroppo, però, in molti casi, il numero di possibili distribuzioni è troppo grande. Una soluzione consiste nell'applicazione di **un'approssimazione Monte Carlo**: in cui prendiamo un campione casuale di N distribuzioni e la probabilità di s di apparire nel campione è proporzionale alla probabilità P(s).

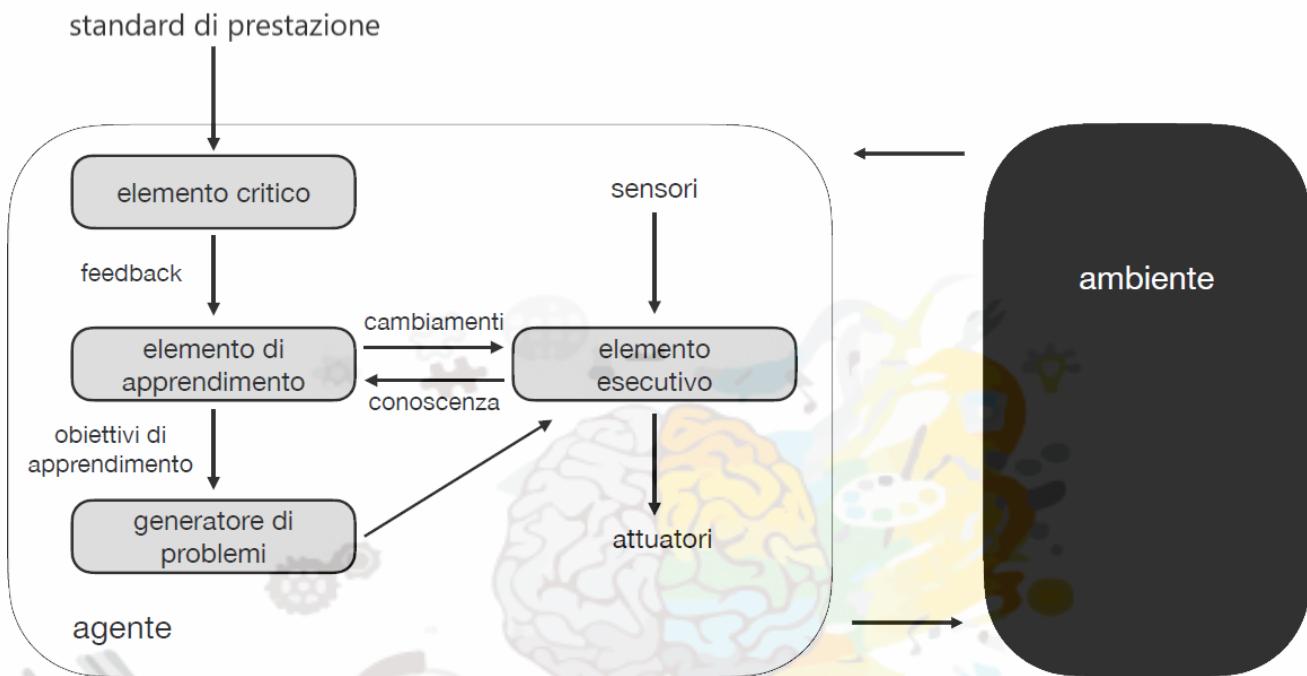


ALGORITMI DI APPRENDIMENTO

Capitolo 7 – Teoria dell’Apprendimento

7.1 Agenti capaci di apprendere

L'apprendimento presenta il vantaggio di permettere agli agenti di operare in ambienti inizialmente sconosciuti diventando col tempo più competenti.



Un agente di questo tipo ha 4 componenti principali:

- **Elemento di apprendimento.** L'elemento responsabile del miglioramento interno.
- **Elemento esecutivo.** L'elemento responsabile della selezione delle azioni esterne. Questo è l'elemento che abbiamo considerato finora come agente.
- **Elemento critico.** L'elemento responsabile di fornire **feedback** sulle prestazioni correnti dell'agente, così che l'elemento di apprendimento possa determinare se e come modificare l'elemento esecutivo affinché si comporti meglio in futuro.
- **Generatore di problemi.** L'elemento responsabile di suggerire azioni che portino ad esperienze nuove e che portino l'agente ad apprendere nuove conoscenze da sfruttare per migliorare le sue azioni.

Cosa significa apprendere?

L'apprendimento è un processo tramite il quale un sistema migliora le sue prestazioni sulla base dell'esperienza. Nel nostro caso, l'apprendimento ha a che fare con degli agenti intelligenti che migliorano automaticamente operando in un ambiente.

Cosa significa apprendere, più tecnicamente?

Apprendimento = Migliorare con l'esperienza nell'esecuzione di un **task**;

- Migliorare nell'esecuzione del task T;
- Rispetto alla misura di prestazione P;
- Sulla base dell'esperienza.

7.2 Machine Learning

Il **machine learning** riguarda lo studio e la costruzione di algoritmi che possano imparare dai dati e sulla base di questi fare previsioni.

Sono due le caratteristiche che hanno portato alla popolarità del *machine learning*: la disponibilità dei dati e la disponibilità di strumenti computazionali adeguati.

Gli algoritmi che andremo ad analizzare consentono di andare oltre la programmazione classica, nel senso che non si limiteranno ad eseguire dei comandi esplicativi, ma saranno in grado di prendere **decisioni data-driven**.

Apprendimento supervisionato

L'**apprendimento supervisionato** è quello in cui un agente apprende usando dei dati **etichettati**.

Le **etichette** determinano la **variabile dipendente**, ovvero quello che l'agente dovrà apprendere.

Quindi, per ogni osservazione, oltre ai dati di input è noto anche il valore della **variabile dipendente**.

L'apprendimento si dice supervisionato proprio perché il progettista deve fornire all'agente delle *etichette* che abiliteranno l'apprendimento.

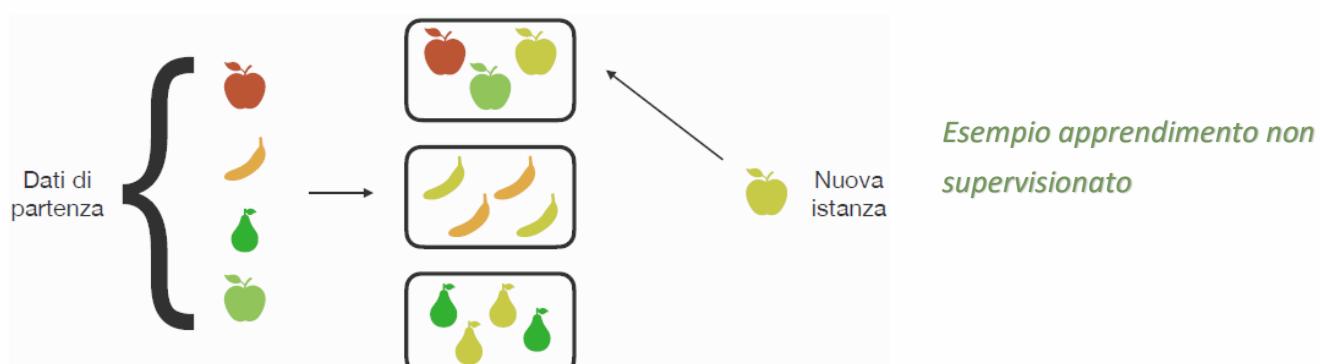
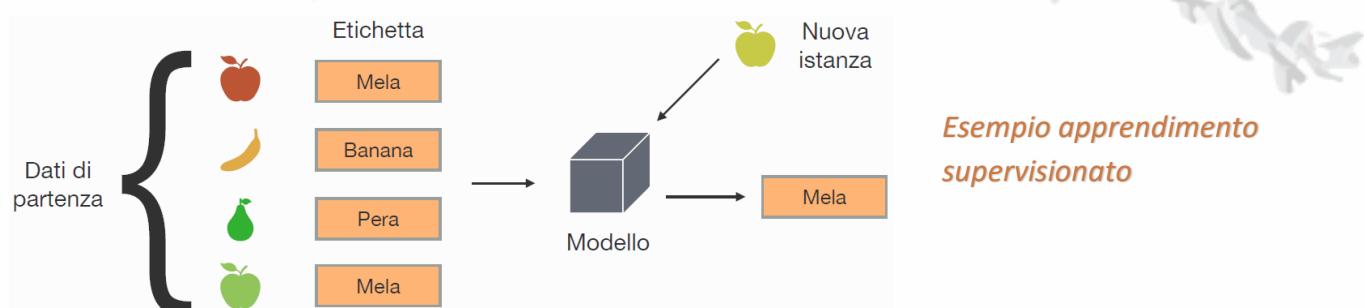
Apprendimento non supervisionato

L'**apprendimento non supervisionato** è quello in cui un agente apprende usando dati *non etichettati*.

L'agente sarà quindi in grado di imparare senza conoscere il valore reale della **variabile dipendente**.

Generalmente, agenti non supervisionati vengono usati per problemi più complessi di quelli supervisionati.

L'apprendimento si dice non supervisionato perché il progettista lascerà all'agente il compito di apprendere sulla base dei dati a disposizione.



Oltre alla classica distinzione tra apprendimento supervisionato e non supervisionato, esistono altre tipologie di machine learning, come l'apprendimento **semi-supervisionato** e per **rinforzo**.

- Nell'**apprendimento semi-supervisionato** alcuni dei dati sono etichettati, altri no. L'agente intelligente sarà tenuto ad apprendere quali sono le etichette mancanti.
- Nell'**apprendimento per rinforzo**, l'agente compie azioni in maniera sequenziale e, al termine di ogni sequenza, gli verrà assegnata una “*ricompensa*” che ha lo scopo di incoraggiare comportamenti corretti.

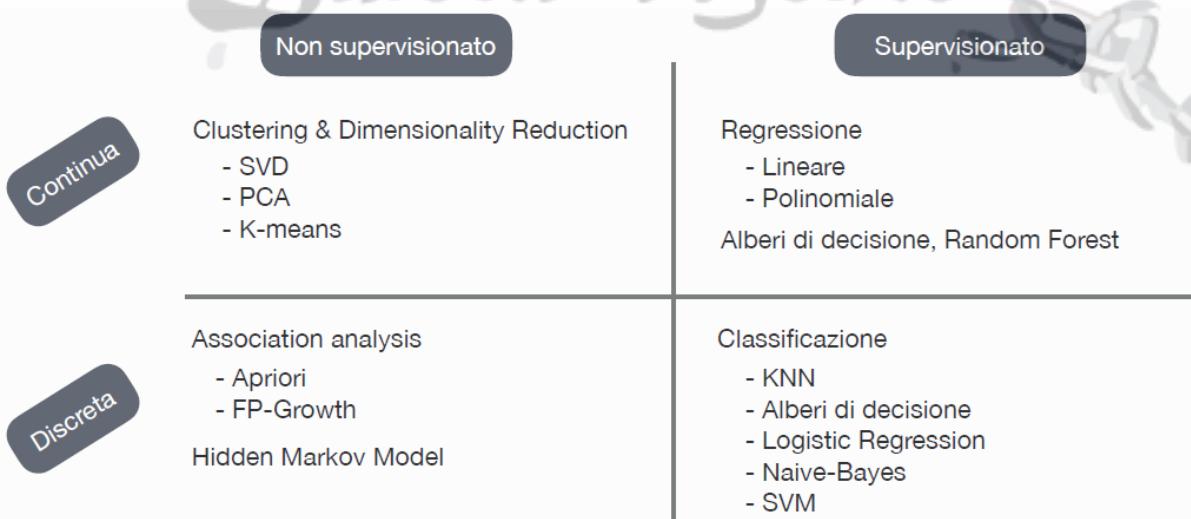
Classificazione degli algoritmi

Oltre alla classificazione degli algoritmi, un altro modo di suddividere i problemi di *machine learning* si basa sull'**output** che si intende ottenere. Per *output* si intende il range di valori che la variabile dipendente potrà assumere.

- Se questo è **continuo**, allora si parla di **regressione**.
Esempio: stimare lo stipendio di una persona in base al titolo di studio;
- Se questo è **discreto**, allora si parla di **classificazione**.
Esempio: valutare se una e-mail è spam in base al suo oggetto;
- Se questo è la **suddivisione dei dati in gruppi**, allora si parla di **clustering**.
Esempio: identificare se esistono gruppi di utenti sul web con comportamento simile.

Esistono altre centinaia di problemi. Ad esempio:

- la **riduzione della dimensionalità** viene usata per ridurre le caratteristiche significative ed eliminare le caratteristiche ridondanti in un insieme di dati.
- Il **mining delle associazioni** serve ad identificare dei pattern comuni in un insieme di transazioni.



Come scegliere il giusto algoritmo?

Ci si affida al **metodo empirico**. Se la teoria ci aiuta a fare una prima selezione del problema e degli algoritmi che possono essere utilizzati, la pratica ci fa scegliere la soluzione migliore.

Procediamo dunque alla **misura dell'errore**: si costruiscono diversi modelli, si calcola l'errore per ciascuno di essi e si sceglie poi quello con i migliori risultati. In altri termini, la *misura dell'errore* ci fornisce una base di confronto tra più modelli.

Errore, Bias e Varianza

Errore. L'errore, o residuo, di un modello di *machine learning* è la differenza tra il valore atteso della variabile da predire e il suo valore attuale.

L'errore irriducibile è quell'errore che dipende esclusivamente dai dati e che avremo sempre e comunque. Oltre questo tipo di errore, dobbiamo considerare anche il fatto che *i modelli di machine learning non sono infallibili* e potrebbero produrre predizioni errate.

La selezione del modello si riferisce alla capacità di valutare le prestazioni di diversi modelli di machine learning al fine di scegliere il migliore. Il modello di machine learning di base è il seguente: $\mathbf{Y} = \mathbf{f}(\mathbf{X}) + \epsilon$ dove $\epsilon = \text{bias} + \text{varianza}$.

Tutti gli sforzi relativi alla ricerca del miglior *modello di machine learning* puntano a minimizzare l'**errore riducibile**, il quale dipende da due fattori principali:

1. **Bias:** indica l'insieme di decisioni usate dal modello per predire un valore di output dati degli input che non ha ancora incontrato (anche detto **underfitting**). Il modello ha un certo *bias* se, quando viene addestrato su diversi dataset, l'output che restituisce è sistematicamente sbagliato.
2. **Varianza.** Il modello ha una certa *varianza* se, quando viene addestrato su diversi dataset, l'output che restituisce è sistematicamente diverso. Un'alta *varianza* è anche detta **overfitting**.

Quindi, l'obiettivo è di rendere nulli **bias** e **varianza**. Sfortunatamente, *bias* e *varianza* sono inversamente correlati: tanto diminuisce il *bias* tanto aumenta la *varianza* e viceversa.

Quindi, è necessario trovare un **compromesso bias-varianza**, ovvero un compromesso tra la "flessibilità" del modello ed il comportamento su dati che non ha mai visto.

Underfitting

Un modello con **bias elevato** è più semplice di quanto dovrebbe essere e quindi tende a sotto-dimensionare i dati. In altre parole, il modello non riesce ad apprendere ed acquisire gli schemi del dataset.

Chiaramente, un modello di questo tipo non si adatta correttamente all'insieme di dati di input e quindi avrà una **bassa precisione** quando dovrà predire nuovi dati, ovvero sbaglierà frequentemente.

Inoltre, un modello di questo tipo **non potrà risolvere problemi complessi**, ovvero problemi per i quali l'insieme di dati di input è particolarmente intricato.

Overfitting

Un modello con **varianza elevata** è più complesso di quanto dovrebbe essere e quindi tende a sovra-dimensionare i dati. In altre parole, è uno studente che studia a memoria e/o tende a complicare troppo le cose.

Un modello di questo tipo ha due possibili conseguenze:

1. non riesce ad apprendere dati anche solo leggermente diversi da quelli che già conosce;
2. oppure si comporta in maniera troppo complessa, peggiorando le prestazioni.

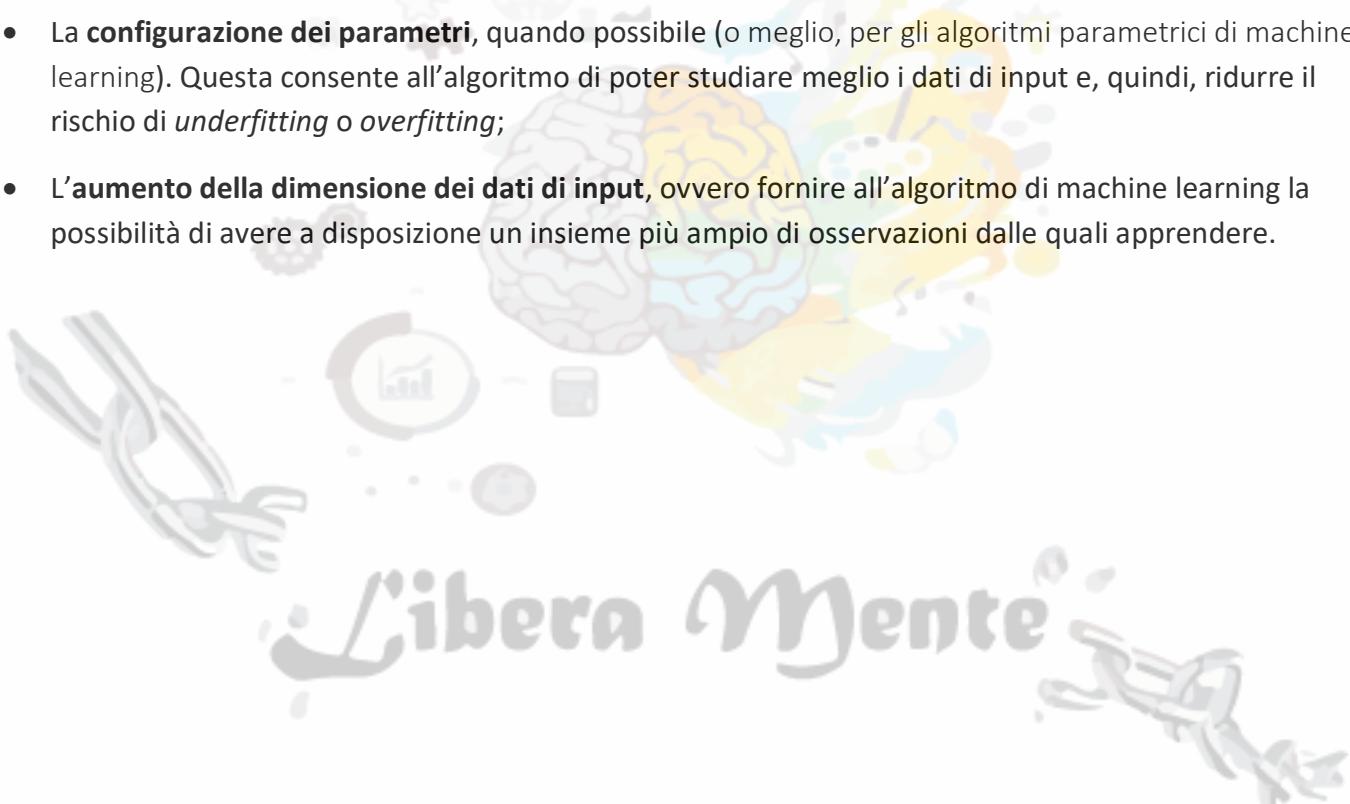
Quindi, un modello di questo tipo tenderà a risolvere **problemi semplici utilizzando soluzioni complesse**, non essendo capace di generalizzare le competenze acquisite dal dataset di input.

Per diagnosticare problemi di **underfitting** e **overfitting**, è necessario valutare il modello generato su un insieme di dati quanto più ampio possibile: più dati abbiamo, più è facile per un algoritmo di machine learning apprendere correttamente.

In alcuni casi, è possibile risolvere o almeno diminuire i rischi di *underfitting* e *overfitting* lavorando sulla **configurazione** degli algoritmi di machine learning.

Alcune operazioni tipicamente utilizzate riguardano:

- La **selezione delle caratteristiche rilevanti**, tramite la quale un algoritmo di machine learning riesce ad apprendere “meglio”, focalizzando l’attenzione solo sui dati che rappresentano la *variabile dipendente*;
- La **convalida incrociata**, tramite la quale vengono generati dei dataset di test, così da consentire all’algoritmo di machine learning di perfezionare l’apprendimento sui vari insiemi di test;
- La **configurazione dei parametri**, quando possibile (o meglio, per gli algoritmi parametrici di machine learning). Questa consente all’algoritmo di poter studiare meglio i dati di input e, quindi, ridurre il rischio di *underfitting* o *overfitting*;
- L’**aumento della dimensione dei dati di input**, ovvero fornire all’algoritmo di machine learning la possibilità di avere a disposizione un insieme più ampio di osservazioni dalle quali apprendere.



Capitolo 8 – Ingegneria del Machine Learning

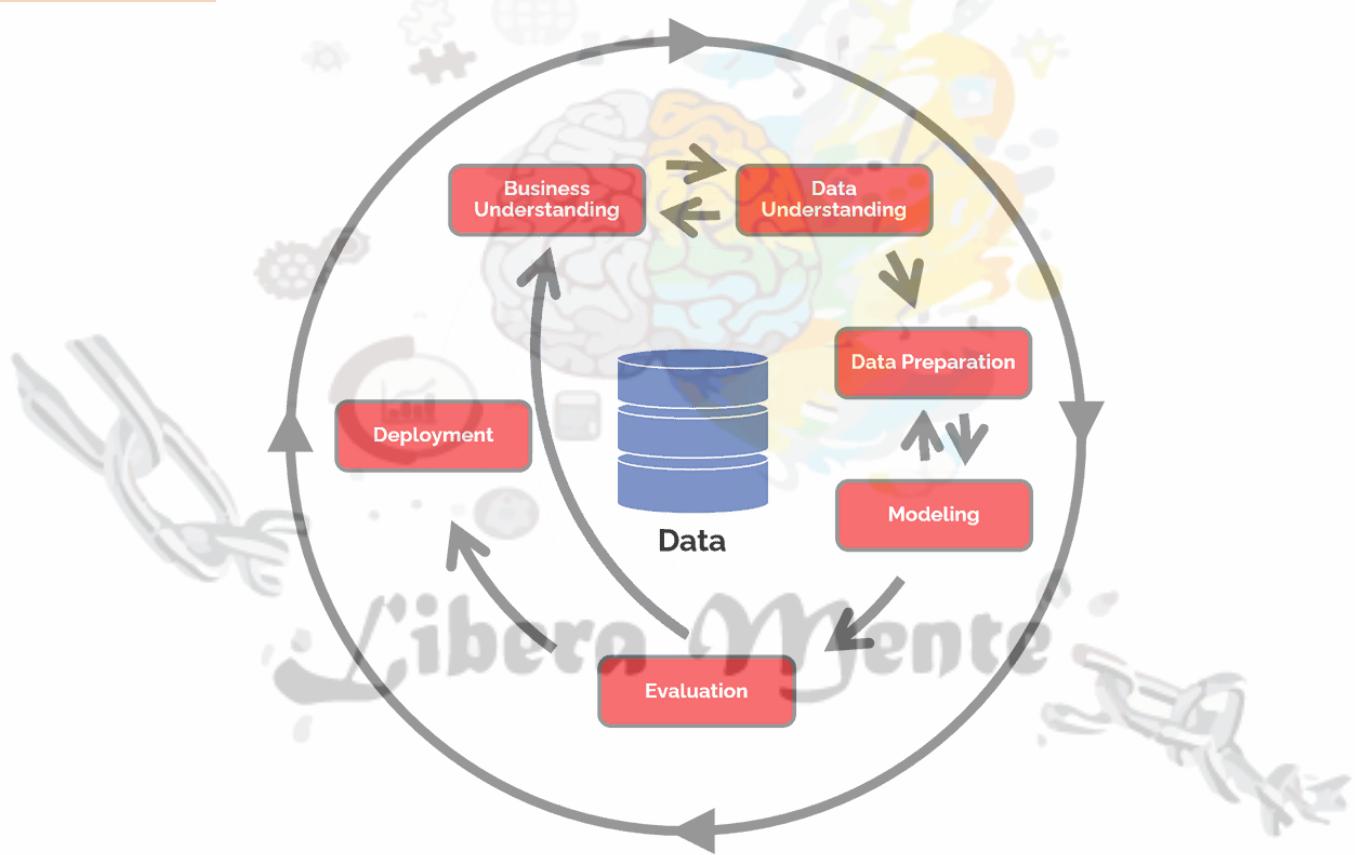
8.1 Modello CRISP-DM

Quando progettiamo una soluzione basata su machine learning dobbiamo pensare a due cose principali: **data e software engineering**.

Tutto ciò può essere riassunto dal cosiddetto **modello CRISP-DM**, che rappresenta il **ciclo di vita** di progetti basati su *intelligenza artificiale* e *data science*.

Possiamo paragonare il modello *CRISP-DM* ad un **modello a cascata con feedback** utilizzato per lo sviluppo di sistemi software tradizionali.

Il *CRISP-DM* è un **modello non sequenziale** in cui le diverse fasi possono essere eseguite un numero illimitato di volte.



➤ Business Understanding

La prima fase è chiaramente quella di **raccolta dei requisiti** e di **definizione degli obiettivi di business** che si intende raggiungere (ovvero, che cosa deve fare il machine learner che stiamo progettando).

La fase di **business understanding** prevede la definizione dei cosiddetti **business success criteria**, ovvero i criteri secondo i quali potremo accettare che il sistema costruito è in linea con gli obiettivi di business.

In questa fase, bisogna inoltre determinare la disponibilità delle risorse, stimare i rischi, definire i relativi piani di contingenza e condurre una analisi costi-benefici.

Oltre che definire i criteri di successo da un punto di vista di business, è inoltre necessario definire gli obiettivi tecnici che si intendono raggiungere. Infine, verranno selezionate le tecnologie ed i tool necessari agli obiettivi.

Output: Piano di progetto, il documento che spiega l'esecuzione del progetto da un punto di vista di gestione tecnica e socio-tecnica.

➤ Data Understanding

Sulla base degli obiettivi definiti nella fase precedente, il **secondo passo** consiste nell'**identificazione, collezione e analisi dei dataset** che possono portare al raggiungimento degli obiettivi: cioè vengono acquisiti i dati necessari al raggiungimento degli obiettivi di business e tecnici.

1. I dati verranno poi **caricati** in un tool di analisi dei dati.
2. I dati vengono quindi **esaminati** e **documentati** rispetto al loro formato, il numero di record e il significato di ciascun campo.
3. Il **terzo task** consiste nell'**esplorazione dei dati**: cioè vengono visualizzati e vengono identificate eventuali relazioni.
4. Infine, il processo di **qualità dei dati**: vengono identificati e documentati possibili problemi di qualità dei dati (ad esempio, dati mancanti).

Output: **Documento di analisi dei dati**, che riporta i metodi di estrazione ed analisi, oltre che le relazioni tra i dati ed eventuali problemi di qualità.

➤ Data Preparation

L'obiettivo di questa fase è quello di **preparare i dati** in maniera tale che possano essere utilizzati nei passi successivi del processo. Questo include un processo fondamentale, noto come **feature engineering**, ovvero la selezione delle caratteristiche del problema che hanno maggiore potenza predittiva.

Inoltre, questa fase include l'implementazione dei processi di **pulizia dei dati** sulla base dei *problem di qualità* riscontrati nella fase precedente.

Infine, i dati vengono formattati in maniera tale che possano essere prese in input da un modello di machine learning (questo potrebbe dipendere dai tool selezionati in fase di *business understanding*).

Output: **Insieme di dati di input**, ovvero l'insieme di dati che verranno considerati in fase di modellazione dell'algoritmo di machine learning.

➤ Data Modeling

Una volta sistemati i dati, si passa alla fase di **modellazione**: va selezionata la tecnica o l'algoritmo da utilizzare. Ad esempio

- Conviene modellare il problema come un problema di classificazione o regressione?
- Quale soluzione sarà più adatta e utile al raggiungimento degli obiettivi?

Dopodiché, si passa alla fase di **addestramento**. In questo caso, si configurano i parametri del modello selezionato, si addestra il modello e si descrivono i risultati ottenuti in fase di addestramento.

Molto spesso, il progettista sarà costretto a tornare nella fase di **data preparation** per effettuare ulteriori operazioni sui dati.

Output: **Modello di machine learning**, ovvero l'algoritmo addestrato e configurato sui dati a disposizione.

➤ Evaluation

La fase di **validazione** ha l'obiettivo di valutare se i risultati sono chiari, se sono in linea con gli obiettivi di business e se rivelano delle prospettive aggiuntive alle quali il progettista non aveva pensato.

In questa fase, è inoltre importante verificare la consistenza e la solidità dell'intero processo. Ad esempio:

- Ci sono degli aspetti che non sono convincenti?
- Ci sono delle alternative metodologiche che potrebbero portare a risultati diversi?
- E come queste alternative impattano i risultati ottenuti?

Una volta avute le risposte, si può procedere alla definizione dei prossimi passi da effettuare:

- Possiamo considerare la definizione dell'approccio completa?
- Oppure è necessario fare un passo indietro e valutare opzioni diverse?

Output: Risultati della validazione del modello, che rivelano il grado di attendibilità e conformità dell'algoritmo rispetto agli obiettivi di business.

➤ Deployment

La fase di **deployment** ha l'obiettivo di mettere in funzione l'approccio definito e, quindi, renderlo usabile.

- Il modello generato, precisamente, come dovrà essere reso disponibile?
- Quale sarà il grado di interazione con gli utenti?
- Ed in che modo gli utenti utilizzeranno il modello?

In altri termini, questa fase vede il passaggio dall'**ingegneria del machine learning** all'**ingegneria del software** e all'**ingegneria dell'usabilità**.

Questo è ancora più evidente se consideriamo che questo modello non resterà nel suo stato in eterno bensì avrà bisogno di essere costantemente **monitorato** e **manutenuto**.

Infatti, secondo le prime due leggi di *Lehman* sull'evoluzione di sistemi software:

- 1) un sistema che non cambia è un sistema che diventerà presto inutile;
- 2) la complessità del sistema crescerà inesorabilmente nel tempo.

Output: Report finale di progetto, che descrive tutte le fasi condotte, oltre che il piano di manutenzione e monitoraggio.

Note sul modello **CRISP-DM**

È bene notare che il modello **CRISP-DM** non è per definizione, rigido. Infatti, sono molte le fasi in cui un progettista potrebbe trovarsi a tornare sui suoi passi e/o effettuare azioni più e più volte, finendo quindi in una situazione agile.

Quindi, il modello è di fatto **flessibile** e può essere considerato sia tradizionale che agile, dipendentemente dal livello di flessibilità che si intende mettere in atto.

Dall'altra parte però, poiché i progettisti sono, per natura, *socio-tecnici*, il modello **CRISP-DM** non chiarisce chi è responsabile di fare cosa. Un modo migliore di sviluppare modelli di machine learning è di utilizzare il **modello SCRUM**.

8.2 Modello SCRUM e TDSP

SCRUM è un modello di ciclo di vita che prevede la divisione dei blocchi di lavoro in **sprint**, ovvero dei brevi periodi di tempo in cui vengono definiti determinati requisiti, analizzati, progettati e sviluppati.

A differenza dei cicli di vita tradizionali, SCRUM prevede lo **sviluppo incrementale** di porzioni del sistema. L'**incrementalità** serve da un lato ad avere una migliore interazione con gli **stakeholder** (dovuta al rilascio di piccole parti del sistema) e dall'altro a minimizzare i rischi (che nel processo tradizionale sono dovuti al possibile *misunderstanding* dei requisiti).

SCRUM prevede, tra gli altri, 3 ruoli fondamentali:

1. Il **product owner**, colui il quale rappresenta gli **stakeholder**, definisce e prioritizza i requisiti, per aggiungerli al *backlog*;
2. Lo **SCRUM Master**, colui che agevola il lavoro e coordina le attività di sviluppo. Sebbene sia un ruolo manageriale, non è da confondere con un *project manager*, poiché non ha alcuna responsabilità di gestione del personale.
3. Il **team di sviluppo**, che è responsabile della consegna del prodotto. Il team dovrebbe essere composto da 3 - 9 persone con competenze trasversali, le quali si occupano della documentazione e dello sviluppo del prodotto in ogni sprint.

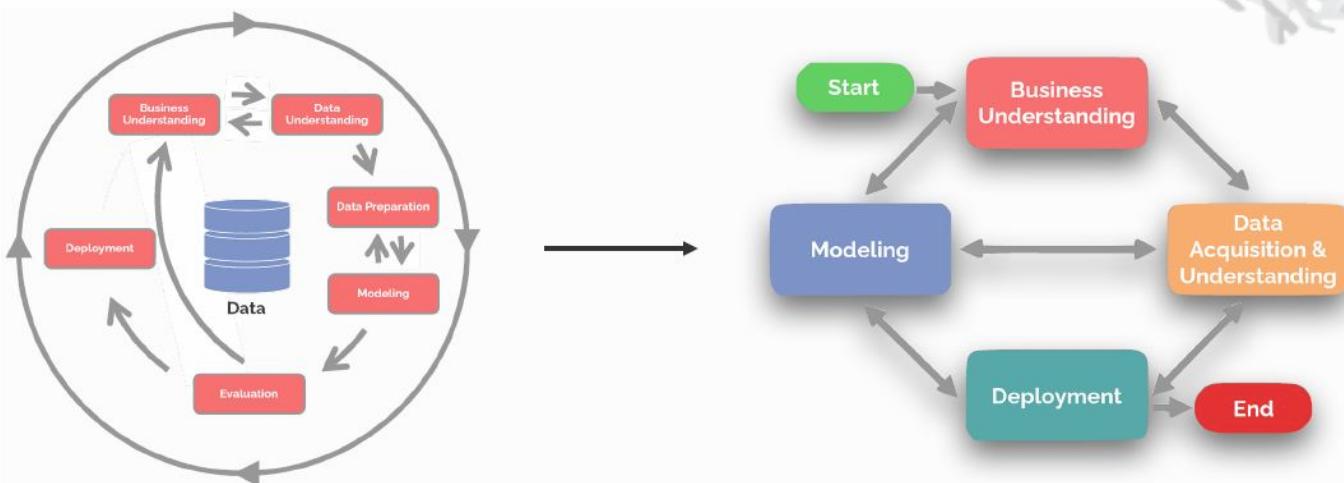
In SCRUM, ogni sprint inizia con uno **sprint planning**, in cui vengono definite le attività da compiere nel prossimo periodo temporale.

Il lavoro è monitorato tramite i **daily scrum**, una riunione quotidiana in cui i progressi vengono riportati.

Alla fine dello sprint, viene effettuata una **sprint review**, dove il team mostra i progressi effettuati.

Infine, lo sprint viene chiuso con uno **sprint retrospective**, dove il team riflette sui problemi riscontrati e definisce i miglioramenti da fare nel prossimo sprint.

Combinando SCRUM e CRISP-DM, si ottiene il **TDSP** (Team Data Science Process), il quale incorpora aspetti socio-tecnici nell'esecuzione dei progetti.



La grande differenza sta nel fatto che, ad intervalli regolari, viene effettuata una fase di **customer acceptance**, ovvero una validazione di come il sistema implementa i requisiti di business.

Inoltre, TDSP definisce 6 ruoli esplicativi:

1. **Project manager**: il responsabile dell'intero progetto;
2. **Project lead**: simile allo *SCRUM master*, è il team leader.
3. **Data engineer**: il responsabile della *data acquisition*;
4. **Data scientist**: il responsabile della *data understanding*;
5. **Application developer**: il responsabile dell'implementazione dell'applicazione;
6. **Solution architect**: progetta e manutiene le architetture delle applicazioni;

Vantaggi TDSP

TDSP enfatizza la necessità di rilasci incrementali, il ché minimizza i rischi connessi a potenziali *mistradimento* dei requisiti.

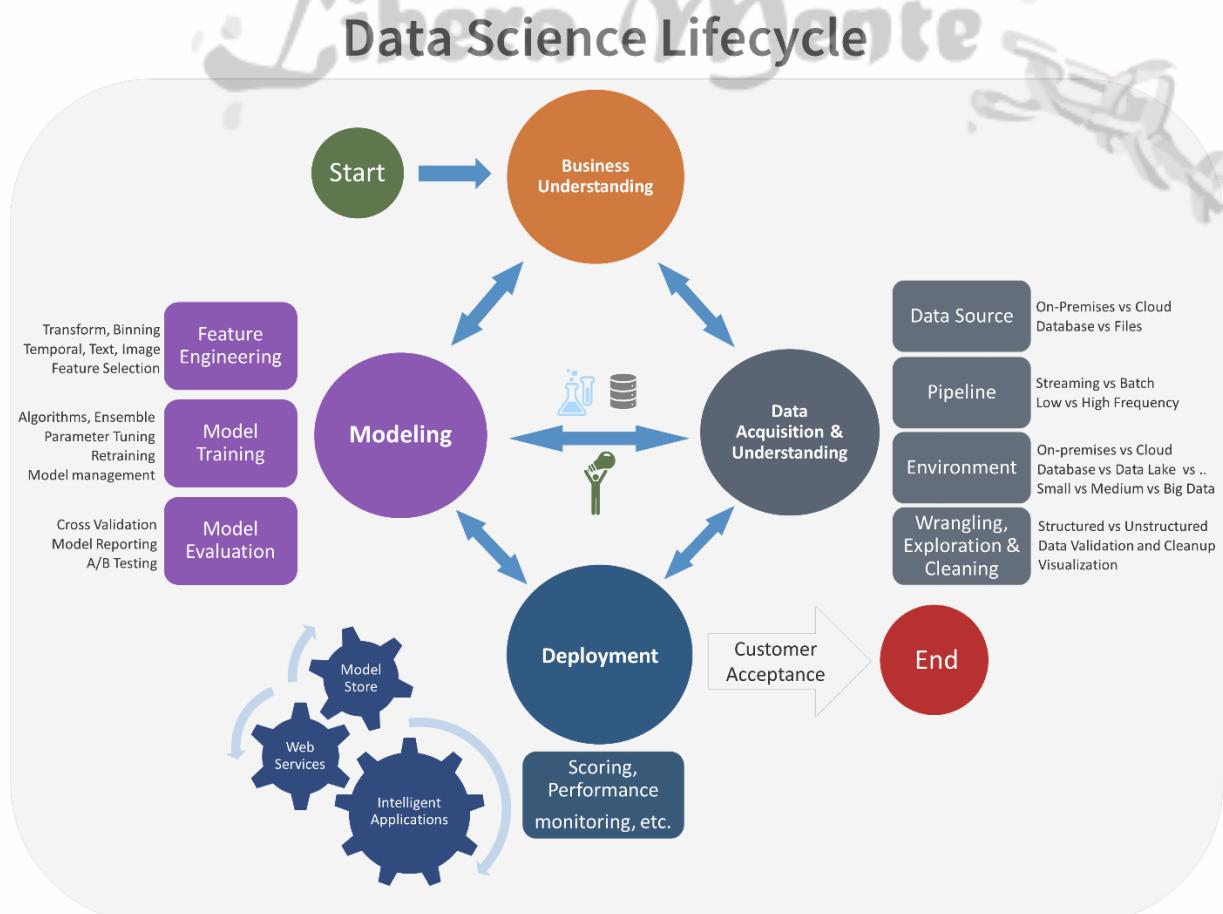
Più importante, TDSP riconosce la complementarietà tra ingegneria del machine learning e ingegneria del software. I ruoli previsti da TDSP includono:

- esperti di project management e lead;
- esperti di ingegneria del software;
- esperti di data engineering e science.

Altro aspetto da non sottovalutare è che il TDSP utilizza una terminologia e dei tool simili a quelli di SCRUM, il che semplifica la comprensione delle responsabilità.

Svantaggi TDSP

Di contro, TDSP ha la necessità di definire degli sprint che potrebbe essere controproducente: a differenza dei progetti software tradizionali, la definizione di sistemi di machine learning potrebbe incappare in problemi tecnici non di poco conto, come ad esempio la mancanza o la bassa qualità dei dati.



Capitolo 9 – Data Quality e Feature Engineering

9.1 Dati

L'intelligenza artificiale è una scienza che richiede un ampio utilizzo di **dati**, in particolare quando parliamo di algoritmi di apprendimento in quanto l'apprendimento avviene esclusivamente con l'acquisizione di dati.

Ingegneria dei dati è l'insieme delle tecniche e degli algoritmi che consentono l'**estrazione**, l'**analisi** e la **preparazione** di dati che siano fruibili da altre tecniche o da algoritmi di *data analytics*.

Qualità dei dati descrive l'accuratezza, la completezza e la consistenza dei dati.

Avere dati di qualità è l'unico modo di costruire strumenti di intelligenza artificiale capaci di assistere gli utenti nel processo di *decision making*.

Data governance gestisce la disponibilità, usabilità, integrità e sicurezza dei dati. E' basata su standard interni ad un'organizzazione o politiche che ne regolano l'utilizzo.

Data leakage è il problema che si presenta quando un modello è capace di lavorare accuratamente in fase di addestramento, ma non in fase di rilascio.

Tra i *data leakage* abbiamo i **leaky predictor**, ovvero quelle caratteristiche che ci aiutano a caratterizzare il problema, ma che nella pratica non saranno disponibili il più delle volte, potenzialmente causando quindi il fallimento nostro modello (non possiamo contare su una caratteristica se questa non sarà disponibile).

9.2 Tipologie di dati

In termini di *machine learning*, un **dato** è un qualsiasi elemento di cui si dispone per formulare un giudizio o risolvere un problema. Esistono diversi tipi di dati:

- I **dati strutturati** sono dati organizzati in righe e colonne. Il formato dei *dati strutturati* è molto stretto, nel senso che per ogni colonna sappiamo con esattezza il significato di un dato e quali sono i tipi di informazione per quel dato. Spesso questi dati sono memorizzati in database che rappresentano anche le relazioni tra i dati. In questo caso, i dati possono essere recuperati tramite **query**.



- I **dati non strutturati** possono essere rappresentati da qualsiasi tipo di file che non ricade nella categoria dei *dati strutturati*. Sono sicuramente i più difficili da estrarre poiché richiedono degli strumenti ad-hoc, come parser e/o tool di formattazione creati sulla base dello specifico dato. In questa categoria rientrano i **dati testuali**, per i quali è stata definita una branca specifica dell'intelligenza artificiale nota come **Natural Language Processing**.



- I **dati semi-strutturati** in cui il formato è a metà tra lo strutturato e lo non strutturato. Mentre il formato è fissato, la struttura non ha una definizione stretta. I *dati semi-strutturali* sono generalmente memorizzati come file. Alcuni però potrebbero anche essere presentati all'interno di **database document-oriented**.



9.3 Ingegneria dei dati

Data Cleaning

La **data cleaning** è fase di pulizia dei dati che ci aiuta in casi in cui abbiamo a che fare con dei dati mancanti o rumorosi.

Data imputation: insieme di tecniche che possono stimare il valore di dati mancanti sulla base dei dati disponibili oppure semplificare il problema dei dati mancanti.

Per risolvere il problema dei dati mancanti abbiamo diverse soluzioni:

1. **Scartare le righe** del dataset che presentano dati mancanti: una soluzione facile, ma non sempre applicabile. Se per il problema in esame non abbiamo tante osservazioni, scartare le righe diventa un problema.
2. **Scartare le colonne** del dataset che presentano dati mancanti. Se la colonna che presenta dati mancanti rappresenta una caratteristica rilevante per il problema in esame, non possiamo scartarla.
3. Possiamo usare l'**imputazione statistica**, la quale si basa sull'applicazione di semplici tecniche statistiche per stimare il valore dei dati mancanti. Questa tecnica è facile da applicare però presenta 2 problemi:
 - non può essere applicata su dati non-numerici;
 - non considera l'incertezza quando va ad imputare i dati.
4. Con l'**imputazione tramite most frequent imputation** i dati mancanti sono sostituiti dal valore più frequente contenuto in una colonna (però quando ci sono tanti dati mancanti, si rischia di influenzare eccessivamente la distribuzione della variabile).
5. Con l'**imputazione deduttiva** il progettista definisce una regola di imputazione sulla base di una deduzione logica (però non è sempre facile trovare una regola valida e sicuramente non è un approccio scalabile).

Feature Scaling

Se l'insieme dei valori per una determinata caratteristica è molto diverso rispetto ad un altro, c'è il rischio di "far confondere" l'algoritmo di apprendimento, infatti, l'algoritmo potrebbe sottostimare/sovraffidare l'importanza di una caratteristica poiché questa ha una scala di valori molto inferiore/superiore rispetto alle altre.

Il **feature scaling** è l'insieme di tecniche che consentono di normalizzare o scalare l'insieme di valori di una caratteristica. Il metodo più comune per normalizzare è chiamato **min-max normalization**:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

Dove **a** e **b** rappresentano i valori minimo e massimo che vogliamo ottenere dalla normalizzazione (ad esempio, 0 e 1 se vogliamo normalizzare nell'intervallo [0, 1]).

Una alternativa è quella della **z-score normalization**, cioè la **normalizzazione** di default implementata da molti dei tool di machine learning:

- x è il *valore originale*;
- \bar{x} è la *media della distribuzione*;
- σ è la *deviazione standard*.

$$x' = a + \frac{(x - \bar{x})}{\sigma}$$

Feature selection

Il **feature engineering** è il processo in cui il progettista utilizza la propria conoscenza del dominio per determinare le caratteristiche (feature) dai dati grezzi estraibili tramite tecniche di **data mining**.

In altri termini, il *feature engineering* ci consente di “dare un senso” ai **dati strutturati, non strutturati o semi-strutturati** che possiamo estrarre dalle diverse sorgenti a disposizione.

L’obiettivo finale è quello di definire delle caratteristiche, anche chiamate **feature, metriche, o variabili indipendenti** che possano caratterizzare gli aspetti principali del problema in esame e, quindi, avere una buona potenza predittiva.

Questo processo è quello più creativo e complesso dell’intera modellazione, poiché dipende dal problema specifico e dall’abilità del progettista di individuare quali sono le caratteristiche che possono influenzare la predizione di un fenomeno.

Quindi, se nelle fasi precedenti abbiamo potuto estrarre e manipolare una grande quantità di dati, arrivati a questo punto ci interessa selezionare le sole variabili o i soli dati che hanno un impatto importante sulla variabile che intendiamo predire.

Identificare buone *feature* porta diversi vantaggi da un punto di vista pratico:

- la flessibilità del modello di machine learning aumenta: anche scegliendo un algoritmo di apprendimento sub-ottimo, le prestazioni resteranno alte;
- la semplicità del modello aumenta: anche scegliendo una configurazione sub-ottima dell’algoritmo di apprendimento, le prestazioni resteranno alte;
- il livello di explainability del modello aumenta: avendo delle buone *feature*, riusciremo facilmente a capire il perché del comportamento del modello;
- le prestazioni del modello aumentano: avendo delle buone *feature*, è più semplice per il machine learner apprendere e capire come comportarsi su dati ignoti.

E quindi, come posso identificare queste feature?

Il **feature engineering** ha diversi sottocampi, tra cui la:

- **feature extraction**: punta a generare automaticamente delle feature dai dati e riduce della dimensionalità poiché parte da un insieme più grande di dati per identificare quelli più significativi. Inoltre, la *feature extraction* parte dai dati grezzi e li converte in nuovi tipi di dati.
- **feature construction**: punta alla creazione di feature da parte del progettista.
- **feature selection**: seleziona le variabili più significative partendo da quelle a disposizione.

Il modo più semplice per approcciare il problema della *feature selection* è quello di usare **metodi non supervisionati**:

- **Eliminazione di feature con bassa varianza:** questo metodo prevede l'eliminazione delle caratteristiche che hanno valori simili nel dataset. Di default, il metodo prevede l'eliminazione di variabili con varianza zero.
- **Eliminazione univariata di feature:** questo metodo prevede la selezione delle variabili sulla base di test statistici. Ogni **variabile indipendente** viene correlata con la **variabile dipendente**, ottenendo quindi una classifica delle variabili basata sulla correlazione. Si sceglieranno solo le k migliori variabili.

Data balancing

La maggior parte dei modelli di machine learning funzionano bene solo quando il numero di esempi di una certa classe (ad esempio, il fatto che una mail sia classificata come 'spam') è simile al numero di esempi di un'altra classe (la classe 'no-spam').

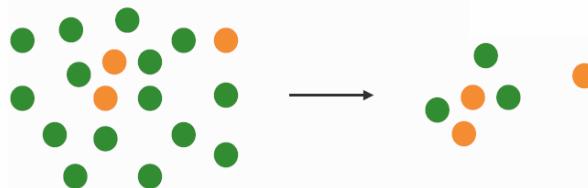
Problema: Molti dei problemi reali sono sbilanciati.

Se non considerassimo il problema dello **sbilanciamento dei dati**, definiremmo un modello di machine learning capace di caratterizzare bene solo gli esempi della classe più popolosa, che nella maggior parte dei casi è quella meno interessante.

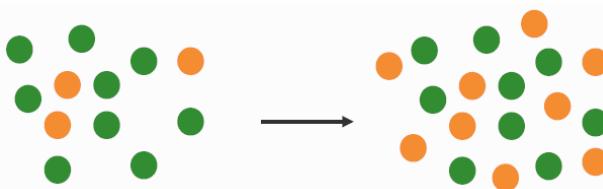
Il data balancing è l'insieme di tecniche per convertire un dataset sbilanciato in un dataset bilanciato.

Sono 2 le tecniche applicabili:

1. **Undersampling:** metodo tramite il quale vengono casualmente eliminate un numero di istanze (righe) del dataset della classe di maggioranza.
 - Il primo problema è che se ho un numero molto basso di istanze della *classe di minoranza*, i dati non saranno sufficienti per apprendere né la classe di maggioranza originaria né tantomeno quella di minoranza.
 - Il secondo problema sta nell'approccio: un *undersampling* casuale potrebbe portare alla rimozione di istanze particolarmente rilevanti per l'apprendimento del modello. Possiamo risolvere il secondo problema tramite una tecnica di *undersampling* basata su **clustering**.



2. **Oversampling:** metodo tramite il quale vengono casualmente aggiunte un numero di istanze (righe) del dataset della classe di minoranza. La duplicazione di istanze potrebbe creare overfitting. Il modello potrebbe saper imparare "a memoria" quali sono le istanze della classe di minoranza solo perché queste rappresentano delle copie che si ripetono più volte.



Capitolo 10 – Classificazione e classificatori

10.1 Problemi di classificazione

La **classificazione** è un task in cui l'obiettivo è predire il valore di una variabile categorica, chiamata variabile dipendente, target, o classe, tramite l'utilizzo di un **training set**, ovvero un insieme di osservazioni per cui la variabile target è nota.

I problemi di classificazione sono istanze di problemi di **apprendimento supervisionato**.

Un **problema di classificazione** porta alla costruzione di un modello, ovvero di uno strumento che fa uso di un algoritmo di apprendimento, anche detto **classificatore**, per classificare i nuovi elementi sulla base del *training set*.

Esistono diversi *classificatori*, ognuno dei quali si distingue dall'altro per via delle assunzioni fatte sui dati così come le proprietà che portano alla classificazione. Tra i classificatori che analizzeremo, abbiamo:

Naive Bayes e **Decision Tree**. Inoltre, oltre i classificatori di base, esistono i cosiddetti **ensemble**: metodi che consentono di combinare insieme più classificatori.

Un semplice esempio è il *Majority Voting*, un algoritmo che classifica una nuova istanza sulla base delle predizioni fatte dalla maggioranza dei classificatori di base.

10.2 Classificazione probabilistica: **Naive Bayes**

Naive Bayes è un algoritmo che considera le caratteristiche della nuova istanza da classificare e calcola la probabilità che queste facciano parte di una classe tramite l'applicazione del teorema di Bayes.

L'algoritmo è chiamato **naive** (ingenuo) poiché assume che le caratteristiche non siano correlate l'una all'altra. Quindi, l'algoritmo non andrà a valutare in fase di classificazione l'utilità data dalla combinazione di più caratteristiche.

Ad esempio: una mela è un frutto di colore rosso, rotondo e avente un diametro di circa 8 centimetri. Sebbene queste caratteristiche siano dipendenti l'una dall'altra (ad esempio, la forma rotonda è collegata al fatto che il diametro sia di 8 centimetri), *Naive Bayes* considererà tutte queste proprietà come indipendenti per la probabilità che questo frutto sia una mela.

Tutto questo perché la classificazione viene eseguita sulla base del **teorema di Bayes**:

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

$P(A)$ e $P(B)$ rappresentano le probabilità di osservare A e B indipendentemente dall'altro.

$P(B|A)$ rappresenta la probabilità di osservare B dato che A si è già verificato.

$P(A|B)$ rappresenta la probabilità di osservare A dato che B si è già verificato.

Classificazione con Naive Bayes

La **classificazione** avviene secondo 3 step:

1. **Calcolo della probabilità della classe**: la probabilità di una classe è il rapporto tra il numero di istanze che appartengono a ciascuna classe per il numero totale di istanze.
2. **Calcolo della probabilità condizionata**: si applica il teorema di Bayes, per determinare le probabilità condizionate.
3. **Decisione**: è identificata nella classe che ottiene il valore di probabilità più elevato.

10.3 Classificazione basata su entropia: *Decision Tree*

L'**albero decisionale** è un algoritmo che mira a creare un albero i cui nodi rappresentano un sottoinsieme di caratteristiche del problema e gli archi rappresentano delle decisioni.

L'obiettivo di un *albero decisionale* è di predire il valore di una variabile *target* sulla base di decisioni conferite dai dati di training.

Gli *alberi decisionali* sono particolarmente utili per la loro facilità di lettura: navigando l'albero è possibile comprendere il motivo per cui è stata fatta una determinata predizione. Infatti, ogni percorso dell'albero corrisponde ad una regola.

Inoltre, gli alberi decisionali possono essere utilizzati sia per **problemi di classificazione** che per **problemi di regressione**. Il loro funzionamento è abbastanza semplice e si compone di 3 passi:

Step 1. Si posiziona la miglior caratteristica del *training set* come **radice** dell'albero.

Step 2. Si divide il *training set* in sottoinsiemi. Un **sottoinsieme puro** contiene tutti e soli gli elementi di una classe (ad esempio, tutte le banane, tutte le mele, tutte le arance). Se un **sottoinsieme non è puro** allora abbiamo ancora incertezza nella classificazione e dobbiamo procedere a dividere ulteriormente il dataset.

Step 3. Si ripetono gli step (1) e (2) su ogni sottoinsieme fin quando non viene raggiunto un nodo foglia in ogni sottoalbero.

Come decidiamo con quale attributo dividere il dataset?

L'**information Gain** è una misura che indica il grado di purezza di un attributo, ovvero quanto un certo attributo sarà in grado di dividere adeguatamente il dataset.

Nella teoria dell'informazione, l'**entropia** indica in che misura un messaggio è ambiguo e difficile da capire: maggiore è l'entropia, minore sarà la quantità di informazione del messaggio.

$$H(D) = - \sum_c p(c) \cdot \log_2 p(c) \quad \text{dove } p(c) \text{ è la proporzione della classe } c \text{ nel dataset } D.$$

Nei decision tree, l'**entropia** è utilizzata come base per l'analisi dell'**information gain**. Partendo dalla radice dell'albero decisionale, usiamo l'entropia per dividere i dati in sottoinsiemi che contengono istanze simili (o omogenee). Per ogni attributo del dataset calcoleremo il suo **gain**:

$$Gain(D, A) = H(D) - \sum_{v \in \text{values}(A)} \frac{|D_v|}{|D|} \cdot H(D_v)$$

dove

- D è l'entropia del dataset;
- D_v è il sottoinsieme di D per cui l'attributo A ha valore v ;
- $|D_v|$ è il numero di elementi di D_v ;
- $|D|$ è il numero di elementi del dataset.

In base ai concetti di entropia ed information gain, possiamo raffinare la procedura di creazione di un albero decisionale:

1. Calcolare l'entropia per ogni attributo del dataset;
2. Dividere il training set in sottoinsiemi utilizzando l'attributo, per cui l'entropia sia minimizzata o, in maniera equivalente, l'information gain è massimizzato;
3. Creare un nodo dell'albero decisionale contenente quell'attributo;
4. Ripetere i passi precedenti fin quando tutti i sottoinsiemi definiti non siano puri.

Come si decide qual è il classificatore da usare per un determinato problema?

Possiamo dividere il dataset di partenza in maniera tale da considerare alcune istanze come non note.

Creeremo quindi due insiemi:

1. il **training set**, che sarà composto delle istanze che l'algoritmo utilizzerà per l'addestramento;
2. il **test set**, che sarà composto delle istanze che l'algoritmo addestrato dovrà predire per la classe di appartenenza.

Esistono diversi modi di dividere *training* e *test set*.

Convalida incrociata (k-fold cross validation)

Un altro metodo è chiamato **convalida incrociata**: metodo statistico che consiste nella ripetuta partizione e valutazione dell'insieme dei dati di partenza.

La *convalida incrociata* prevede i seguenti passi:

1. Mischiare in maniera casuale i dati di partenza;
2. Dividere i dati in k gruppi. È molto comune utilizzare un $k = 10$, infatti, in questo caso si definisce come *10-fold cross validation*;
3. Per ogni gruppo:
 - 3.1 Considerare il gruppo come test set;
 - 3.2 Considerare i rimanenti $k - 1$ gruppi come training set;
 - 3.3 Addestrare il modello con i dati del training set;
 - 3.4 Valutare le prestazioni del modello ed eliminarlo.

È importante notare che ogni istanza sarà assegnata ad un **unico** gruppo durante l'intera procedura di validazione altrimenti, mischieremmo i dati di training con quelli di test, influenzando le capacità predittive del classificatore.

Il **data leakage** è il problema che si presenta quando un modello è capace di lavorare accuratamente in fase di addestramento, ma non in fase di rilascio. Mischiare dati di *training* e *test* porta ad un altro caso di *data leakage*, noto come **leaky validation strategy**.

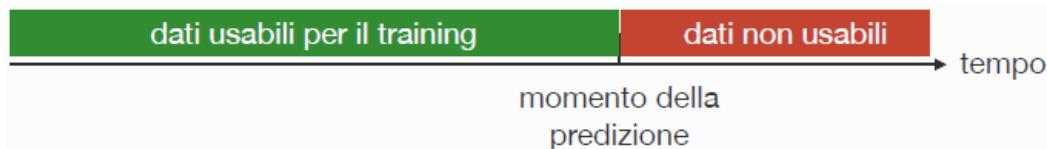
La *convalida incrociata* può avere alcuni problemi:

1. Essendo casuale, il primo passo della validazione potrebbe portare un vantaggio al classificatore.

Soluzione 1. Ripetere la validazione N volte, in modo da limitare l'influenza della casualità del primo passo. In questo caso, parliamo di **N-times k-fold validation**.

Soluzione 2. Modificare il primo passo della validazione in modo da avere un campionamento stratificato, che porta i sottoinsiemi creati ad avere un numero simile di istanze delle diverse classi. Qui parliamo di **stratified k-fold validation**.

2. La convalida incrociata non può essere usata facilmente nel caso in cui i dati seguano un ordine temporale. In questo caso, una convalida incrociata mischierebbe dati temporali, il ché porta ad un caso irrealistico. L'utilizzo della convalida *cross-fold classica* crea un caso di **leaky validation**.



Metriche di valutazione

Per valutare la bontà delle predizioni utilizzeremo la **matrice di confusione**, anche detta *tavella di errata classificazione*, la quale restituisce una rappresentazione dell'accuratezza di un classificatore.

La **matrice di confusione** è una matrice con cui poter indicare se ed in quanti casi il classificatore ha predetto correttamente o meno il valore di un'etichetta del test set.

Sulla base dei valori della matrice di confusione, possiamo poi calcolare diverse metriche, in particolare, **precision** e **recall** sono due delle metriche principali nei processi di classificazione di dati.

Ad esempio, in un problema binario (classificazione true/false):

1. La **precision** indica il numero di predizioni corrette per la classe 'true' rispetto a tutte le predizioni fatte dal classificatore. In altri termini, indica quanti errori ci saranno nella lista delle predizioni fatte dal classificatore. È una metrica che vogliamo massimizzare.
2. La **recall** indica il numero di predizioni corrette per la classe 'true' rispetto a tutte le istanze positive di quella classe. In altri termini, indica quante istanze positive nell'intero dataset il classificatore può determinare. È una metrica che vogliamo massimizzare.

L'**accuracy** indica il numero totale di predizioni corrette per tutte le classi.

IMPORTANTE: Le *metriche* forniscono indicazioni, vanno interpretate correttamente sulla base dei problemi che si analizzano. Mai fidarsi ciecamente delle metriche.

Capitolo 11 – Regressione e regressori

11.1 Problemi di regressione

Regressione: task che ha l'obiettivo di predire il valore di una variabile numerica, chiamata **variabile dipendente** o di **risposta**, tramite l'utilizzo di un *training set*, ovvero un insieme di osservazioni per cui la variabile dipendente è nota.

I **problemi di regressione** sono istanze di *problem di apprendimento supervisionato*. Un **problema di regressione** porta alla costruzione di un **modello**, ovvero di uno strumento che fa uso di un algoritmo di apprendimento, anche detto **regressore**, per predire i nuovi elementi sulla base del training set.

I **regressori** sono delle funzioni matematiche che cercano di descrivere i dati. Diversi *regressori* si distinguono tra di loro per via delle assunzioni fatte sui dati così come delle specifiche proprietà che portano alla **regressione**, ma anche dal numero di variabili indipendenti (**predittori**) di cui disponiamo.

Analizzeremo due *regressori* in particolare: la **regressione singola** e la **regressione multipla**.

Le tecniche di *regressione* sono più complesse di quelle usate in *classificazione*, poiché non possono limitarsi a giudicare le classi generate, ma devono interpretare i valori numerici predetti da più modelli.

11.2 Regressione lineare e multipla

La differenza tra **modelli singoli** e **multipli** dipende dal numero di **predittori** che abbiamo a disposizione. Questo avrà un impatto sulla funzione che verrà generata.

L'obiettivo è quello di trovare una **funzione** che si adatti ai dati di training, cosicché possa essere efficace per predire il valore della variabile dipendente su nuovi dati.

La **funzione di regressione lineare** sarà:

$$\tilde{y} = \beta_0 + \beta_1 \cdot x_1$$

- dove l'intercetta β_0 indica il valore di y_i quando x_i è zero;
- β_1 indica l'**inclinazione** della retta, ovvero la variazione di y_i quando x_i aumenta di un'unità.

$$y = \tilde{y} + \epsilon \quad \rightarrow \quad \epsilon = y - \tilde{y}$$

- ϵ indica il **residuo**, cioè la differenza tra il valore reale ed il valore predetto dal modello di regressione.

Metriche di valutazione

Mean Absolute Error (MAE)

La **metrica MAE** indica la differenza media osservata tra i valori predetti e i valori reali del test set:

$$\frac{\sum_{i=1}^n |\tilde{y} - y|}{n}$$

Mean Squared Error (MSE)

La **metrica MSE** indica l'errore quadratico medio commesso sui dati presenti nel test set:

$$\frac{\sum_{i=1}^n (\tilde{y} - y)^2}{n}$$

Root Mean Squared Error (RMSE)

La metrica RMSE indica la radice quadrata dell'errore quadratico medio commesso sui dati presenti nel test set:

$$\sqrt{\frac{\sum_{i=1}^n (\tilde{y} - y)^2}{n}}$$

I modelli di regressione puntano a trovare la retta che si adatta meglio ai dati. Il criterio più semplice è quello della **minimizzazione del residuo**: più basso è il residuo, più la retta sarà vicina ai dati.

La somma dei residui è detta **funzione di perdita** (loss function). Molti delle tecniche di regressione mirano a minimizzare la *funzione di perdita*.

Calcolare i parametri della funzione (β_0 e β_1)

Il metodo più conosciuto è quello dei **minimi quadrati** (least squared).

1. Per ogni punto noto (x, y) dell'insieme N di punti, calcolare x^2 e xy .
2. Sommare tutti gli x , y , x^2 e xy . Questo ci porta ad avere: $\sum x$, $\sum y$, $\sum x^2$, $\sum xy$.
3. Calcolare l'inclinazione della retta, tramite la formula:

$$\beta_1 = \frac{N \cdot \sum xy - \sum x \cdot \sum y}{N \cdot \sum(x^2) - (\sum x)^2}$$

4. Calcolare l'intercetta della retta, tramite la formula:

$$\beta_0 = \frac{\sum y - \beta_1 \cdot \sum x}{N}$$

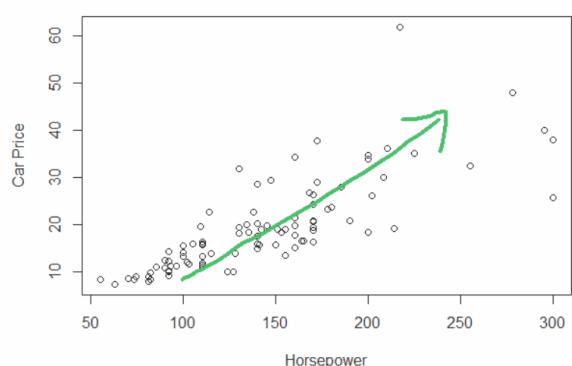
5. Mettere tutto insieme nella versione:

$$\tilde{y} = \beta_0 + \beta_1 \cdot x_1$$

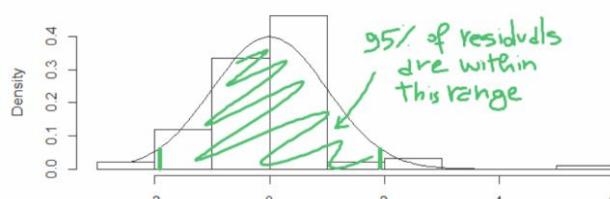
Quando usare la regressione lineare

La regressione lineare può essere utilizzata solo in determinati contesti. In particolare, il suo utilizzo assume:

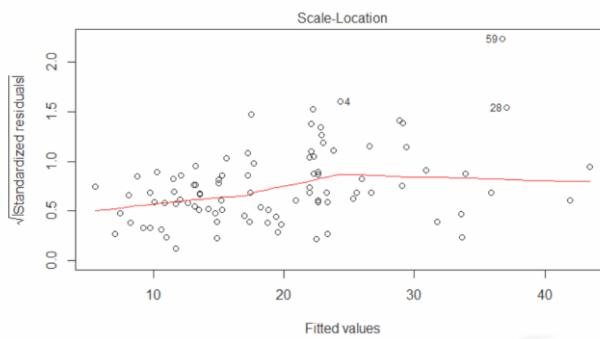
- **Linearità dei dati.** La relazione tra variabile indipendente X e variabile dipendente Y deve essere lineare, ovvero può essere rappresentata tramite una funzione lineare.



- **Normalità dei residui.** Gli errori residui devono essere normalmente distribuiti.



- **Omoschedasticità.** Gli errori residui devono avere una varianza costante. Questa può essere verificata andando a plottare i residui standardizzati vs i valori predetti. Se la proprietà è soddisfatta, vedremo un trend orizzontale piuttosto che punti sparsi nello spazio.



- **Indipendenza degli errori.** Gli errori residui devono essere indipendenti per ogni valore di X. Un test statistico particolarmente utile è noto come Durbin-Watson: quando gli errori sono indipendenti, il valore del test sarà vicino a 2.

Regressione multipla

Se non abbiamo un'unica variabile indipendente, allora parliamo di **regressione lineare multipla**:

$$\tilde{y} = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \dots + \beta_m \cdot x_m$$

$$y = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \dots + \beta_m \cdot x_m + \epsilon$$

I parametri rappresentano l'effetto che la variazione di un'unità delle variabili indipendenti hanno sulla variabile dipendente.

Il metodo dei **minimi quadrati** rappresenta il modello più semplice per la risoluzione di un *problema di regressione multipla*.

La differenza sta nel fatto che verrà utilizzata una forma matriciale per rappresentare le variabili dipendenti ed indipendenti, anche perché non vogliamo più cercare una singola retta di regressione, ma un **piano** che meglio *interpola* i dati di training.

Inoltre, per la regressione lineare multipla valgono gli stessi **vincoli di linearità** e **omoschedasticità** della regressione lineare singola.

Infine, avendo più variabili, potremmo cadere nel problema della **multicollinearità**, per questo è importante eliminare le **variabili ridondanti**.

Capitolo 12 – Clustering

12.1 Problemi di clustering

Clustering: task in cui l'obiettivo è raggruppare oggetti in gruppi che abbiano un certo grado di omogeneità ma che, al tempo stesso, abbiamo un certo grado di eterogeneità rispetto agli altri gruppi.

I **problemi di clustering** sono istanze di problemi di apprendimento non supervisionato.

Un **problema di clustering** ha l'obiettivo di classificare i dati, ma senza assegnare loro un'etichetta. In questo caso, infatti, non abbiamo classi predefinite, ma ogni **cluster** può essere interpretato come una classe di oggetti simili che hanno caratteristiche simili.

Gli **algoritmi di clustering** si basano sul concetto della **similarità**, la quale è interpretata in maniera diversa quando parliamo di *somiglianza intra-gruppo* ed *inter-gruppo*.

La qualità di un *algoritmo di clustering* dipenderà dalla **misura di similarità** utilizzata e dall'algoritmo stesso.

Da un punto di vista esterno, la qualità del *clustering* è misurato in base alla sua abilità di scoprire alcuni o tutti i pattern nascosti, ovvero le caratteristiche che legano elementi simili. Questo è però qualcosa che non potremo calcolare in maniera automatica, poiché non abbiamo le "etichette" di partenza.

Altre proprietà che rendono un *algoritmo di clustering* buono possono essere la **scalabilità, robustezza agli outlier, o interpretabilità dei risultati**.

Ma cosa significa che due elementi sono simili? Da che punto di vista? Inoltre, esistono anche diversi tipi di dati (strutturati, non strutturati, ecc.) e la misura di **similarità** dipende anche da questo.

Dovremmo anche considerare la **dimensionalità dei dati**: quando gli attributi di un elemento aumentano, i dati diventano sempre più sparsi e difficili da raggruppare.

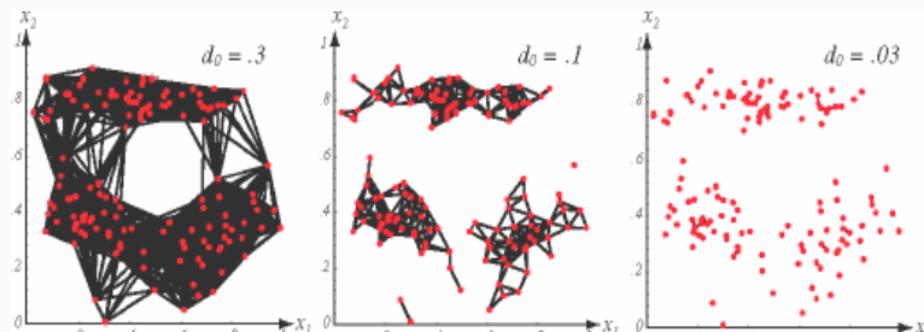
12.2 Misure di similarità

Una **misura di similarità** (o di diversità) tra due pattern è la **distanza** fra essi. Ma non sempre la **distanza** tra due pattern è importante per indicare diversità.

Se la **distanza** rappresenta una buona **misura di similarità**, allora possiamo imporre che la **distanza** tra due pattern nello stesso *cluster* sia più piccola della distanza tra due pattern appartenenti a *cluster* diversi.

In questo modo, fare *clustering* sarebbe semplice, poiché potremmo definire una soglia sulla distanza e raggruppare i pattern al di sotto di tale soglia.

La scelta della soglia influenzerebbe sia il numero che la forma dei cluster.



12.3 Misure metriche

Per capire qual è la giusta soglia da utilizzare, iniziamo con il definire una **misura metrica**, ovvero una quantità calcolabile di distanza tra più elementi di una popolazione.

Misura metrica: dato un insieme S di campioni, una distanza d è **metrica** se valgono le seguenti proprietà:

1. **Identità:** $\forall x \in S, d(x, x) = 0$;
2. **Positività:** $\forall x \neq y \in S, d(x, y) > 0$;
3. **Simmetria:** $\forall x, y \in S, d(x, y) = d(y, x)$;
4. **Disuguaglianza triangolare:** $\forall x, y, z \in S, d(x, z) \leq d(x, y) + d(y, z)$.

Una misura è detta **semi-metrica** quando le proprietà 1, 2 e 3 sono soddisfatte, mentre è **detta pseudo-metrica** quando 1, 3 e 4 sono soddisfatte.

Un esempio di metrica è la distanza euclidea, che calcola la distanza tra due punti in un piano cartesiano:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Altre *metriche di similarità* sono le seguenti:

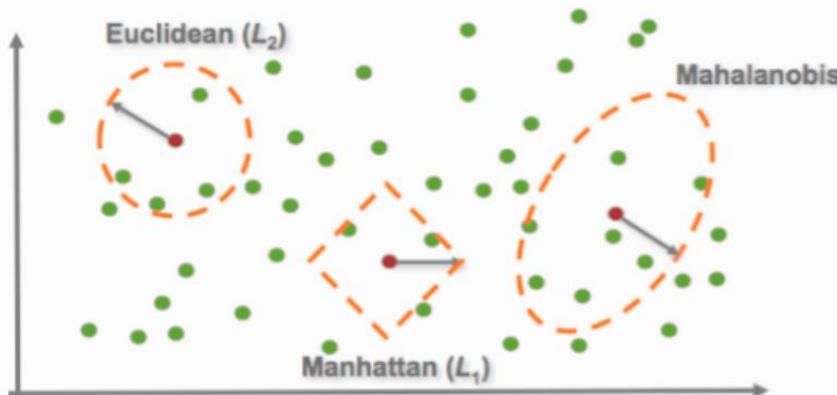
- **Distanza di Manhattan:** la distanza tra due punti è la somma del valore assoluto delle differenze delle loro coordinate.
- **Distanza di Mahalanobis:** è una forma di distanza standardizzata, in cui si tiene conto non solo della diversa dispersione delle variabili, ma anche della loro correlazione.
- **Distanza di Jaccard:** l'indice misura la similarità tra insiemi campionari, ed è definito come la dimensione dell'intersezione divisa per la dimensione dell'unione degli insiemi campionari. La **distanza di Jaccard** è interessante poiché può essere facilmente utilizzata per valutare non solo la similarità tra due insiemi numerici, ma anche tra due stringhe. Vediamo perché:

$$J_\delta(X, Y) = \frac{X \cap Y}{X \cup Y}$$

$S_1 = \text{"Mi piace andare al mare."}$
 $S_2 = \text{"Ieri sono stato al mare."}$
Quanto simili sono queste due stringhe?

$\rightarrow \frac{\{\text{al, mare}\}}{\{\text{mi, piace, andare, al, mare, ieri, sono, stato}\}} = \frac{2}{8} = 0,25$

- **Distanza di Levenshtein:** l'indice misura il numero minimo di modifiche elementari (cancellazione, sostituzione, inserimento) che consentono di trasformare una stringa X in una stringa Y.



Sebbene ci siano molte altre metriche, *non solo strutturali*, è impossibile dire quale sia meglio di un'altra. Questo dipende dal problema e, generalmente, il clustering ottenuto con varie distanze viene sperimentato e validato a posteriori.

12.4 Algoritmi di clustering

Gli algoritmi di clustering si suddividono innanzitutto in varie tipologie:

- **Esclusivi vs non esclusivi.** Un *algoritmo di clustering* è **esclusivo** se ogni pattern appartiene solo ad un cluster. Al contrario, se ogni pattern può essere assegnato a più di un cluster, allora parleremo di **algoritmo non esclusivo**.
- **Gerarchico vs partizionale.** Un algoritmo di clustering è detto **gerarchico** se mira a costruire delle gerarchie di cluster, anche dette *sequenze innestate di partizioni*. Al contrario, un **algoritmo partizionale** effettua solo una partizioni dei pattern.

Gli algoritmi di clustering si suddividono poi in categorie:

- **Agglomerativi vs divisivi.** Un algoritmo di clustering è **agglomerativo** se parte da cluster atomici che puntano ad unire iterativamente in cluster più grandi. Un algoritmo **divisivo** parte invece da ampi cluster per dividerli poi in cluster più piccoli.
- **Seriali vs simultanei.** Un algoritmo di clustering è **seriale** se elabora i pattern uno alla volta. Un algoritmo è **simultaneo** se invece elabora i pattern insieme.
- **Graph-theoretic vs algebrici.** Un algoritmo di clustering è **graph-theoretic** se elabora i pattern sulla base della loro collegabilità. Un algoritmo è **algebrico** se invece elabora i pattern sulla base di criteri di errore.

Infine, la maggior parte degli algoritmi si basa su:

1. **partizionamento iterativo ad errore quadratico**;
2. **clustering gerarchico agglomerativo**.

Vari algoritmi differiscono tra di loro per la misura di similarità usata o per il criterio di ottimizzazione.

Algoritmo k-means

L'algoritmo delle **k-medie** è a partizionamento iterativo ad errore quadratico.

1. Si selezionano k *centroidi* in maniera casuale: k *pattern* sono eletti come rappresentanti;
2. Genera un partizionamento assegnando ogni campione al *centroide* più vicino.
3. Calcola i nuovi *centroidi* del cluster, considerando la media dei valori dei cluster generati al punto 2.
4. Ripeti lo step 2 fin quando i *centroidi* non cambiano.

L'algoritmo **k-means** si può definire come **iterativo** poiché costruisce iterativamente i cluster.

È un algoritmo di **partizionamento** poiché fornisce un'unica partizione degli elementi.

È ad **errore quadratico** poiché mira a minimizzare l'errore rispetto ai *centroidi*.

Tale algoritmo ha una **buona efficienza**. Fornisce buoni risultati se i cluster sono **compatti**, **ipersferici** e **ben separati** nelle caratteristiche. Dovremo però impostare un valore k di cluster a priori ma, non avendo conoscenza del numero di classi del problema, sarà difficile stimare questo valore. Questo ci potrebbe portare ad un ottimo locale.

Per verificare il valore migliore di k , dovremo affidarci a dati empirici o alla conoscenza del dominio oppure agli **algoritmi di ricerca**, in particolare, potremmo utilizzare un **algoritmo di ricerca locale** per

restituire il valore k che ottimizza una funzione obiettivo come, ad esempio, la metrica di valutazione della bontà dei cluster generati. Ulteriori miglioramenti potrebbero coinvolgere la generazione iniziale dei *centroidi*, ad esempio passando da una generazione casuale ad una pseudo-casuale.

Alcune attività di *pre-processing* e *post-processing* sono assolutamente vitali. La **normalizzazione dei dati** e la **rimozione degli outlier** sono necessari per consentire a **k-means** di lavorare su dati più uniformemente distribuiti e, quindi, ridurre il rischio di ottenere risultati sub-ottimi.

Inoltre, il risultato del clustering può essere manipolato a posteriori dall'utente utilizzatore: l'eliminazione dei cluster piccoli (che possono rappresentare degli outlier) o l'unione dei cluster "vicini" sono operazioni che aiutano ad ottenere un migliore risultato. Questi passi possono anche essere implementati durante l'esecuzione.

Infine, per i dati categorici, l'algoritmo **k-means** non può funzionare. In questi casi si può utilizzare l'algoritmo **k-medoids**. Identificati i k *medoidi*, l'algoritmo è identico. L'unica particolarità è identificare una funzione di similarità adatta ai dati.

Medoide: il punto *meno dissimile* di una distribuzione, ad esempio la moda di una distribuzione.

Algoritmo Clustering gerarchico

Il **clustering gerarchico** cerca di considerare raggruppamenti *multi-livello* (ovvero, a diversi livelli di similarità).

In altri termini, nel **clustering gerarchico** non bisogna stabilire un numero k di cluster da generare, ma l'algoritmo andrà a raggruppare elementi sulla base della loro *(de)crescente similarità*.

Questo dà all'utente utilizzatore una maggiore capacità di interpretazione dei risultati, consentendo di scegliere a posteriori il livello di similarità ideale per i dati di input.

Il primo livello conterrà n *cluster*, ovvero ogni cluster avrà un solo elemento.

Il secondo conterrà $n - 1$ *cluster*, ovvero un cluster sarà formato sulla base della similarità delle caratteristiche.

Il processo andrà avanti fino a quando tutti gli elementi non formeranno un unico cluster.



Questa rappresentazione può essere vista come un albero e prende il nome di **dendrogramma** e, come un *albero decisionale*, può essere navigato per decidere quanti cluster sono necessari per i dati di input.

- Parliamo di **clustering gerarchico divisivo** se, nella definizione del *dendrogramma*, partiamo dalla similarità più bassa per arrivare agli elementi atomici.
- Parliamo di **clustering gerarchico agglomerativo** se, nella definizione del *dendrogramma*, partiamo dalla similarità più alta per arrivare ad un unico cluster.

La scelta principale nel *clustering gerarchico* consiste nella misura di distanza tra cluster, che sarà utilizzata per dividere o agglomerare cluster.

NB: Prima abbiamo visto le distanze metriche tra due elementi, qui parliamo di distanze tra cluster!

A prescindere dalla misura metrica utilizzata, possiamo determinare la distanza tra cluster in vari modi. Le più note sono le seguenti:

$$d_{min}(D_i, D_j) = \min_{x \in D_i, x' \in D_j} |x - x'|$$

Minima distanza tra due punti nei cluster D_i e D_j .

$$d_{max}(D_i, D_j) = \max_{x \in D_i, x' \in D_j} |x - x'|$$

Massima distanza tra due punti nei cluster D_i e D_j .

Quando utilizziamo d_{min} per determinare la distanza tra cluster, allora parliamo di **clustering del nearest neighbor**.

Quando utilizziamo d_{max} per determinare la distanza tra cluster, allora parliamo di **clustering del farthest neighbor**.

Algoritmo Density-based clustering

Il *clustering basato sulla densità* raggruppa pattern considerando la loro densità nella distribuzione.

DBSCAN



k-means



Questi algoritmi si basano su due parametri per stabilire il criterio di raggruppamento dei cluster:

- il parametro ***minPts*** stabilisce il numero minimo di punti per considerare un intorno di un punto denso;
- il parametro ϵ definisce un intorno circolare di ciascun punto dai suoi vicini.

Aumentando o riducendo il valore di questi parametri, potremo migliorare la qualità del clustering.

Inoltre, questo tipo di clustering è in grado di scoprire **forme arbitrarie**. L'algoritmo più noto basato sulla densità è chiamato **DBSCAN**.

12.5 Valutazione dei risultati

L'ultimo punto da affrontare quando si parla di clustering è relativo alla **valutazione della bontà dei cluster**. Questa dipenderà da vari fattori, come la misura di similarità utilizzata, dall'algoritmo stesso, o da fattori esterni.

Il vero problema è che essendo **algoritmi non supervisionati**, l'assenza di etichette rende impossibile stimare con esattezza l'accuratezza/precisione dei cluster che sono stati formati da un algoritmo.

Pertanto, dobbiamo utilizzare delle **metriche di stima** tra cui:

1. il **punto di gomito**, detto **Elbow point**, è spesso utilizzato per valutare il numero migliore di cluster da generare negli *algoritmi di clustering partizionale*, ad esempio *k-means*.

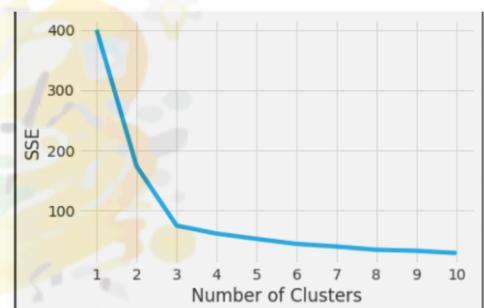
- il **coefficiente di forma**, detto **Silhouette coefficient**, misura la consistenza dei cluster, andando a misurare quanto simili sono gli elementi che compongono un singolo cluster.
- MoJo distance** può essere calcolato solo se abbiamo a disposizione delle etichette per poter valutare la bontà del clustering: in alcuni casi, le etichette vengono ignorate in fase di costruzione del modello per poi essere usate in fase di validazione.

Embow point

Il **punto di gomito** è un *metodo empirico*, che consiste nel graficare i valori candidati del parametro k rispetto alla somma degli errori quadratici ottenuti dall'algoritmo configurato per generare k cluster.

Consideriamo questa figura, in cui vengono rappresentati come, al variare del numero di cluster, varia la *somma degli errori quadratici*. Da qui, possiamo vedere come l'errore diminuisce drasticamente quando si passa da 2 a 3 cluster. L'errore decresce ancora man mano che il numero di cluster aumenta.

Sebbene l'errore venga minimizzato quando i cluster aumentano, avere un numero eccessivo di cluster (in figura, 10) implica avere tanti gruppi formati da pochissimi elementi. Nel caso estremo, avremo l'errore minimo quando ci sarà un cluster per ogni elemento, il che significa non fare *clustering*.



Questa è perciò una situazione da evitare. Vogliamo avere il giusto compromesso tra errore e capacità di raggruppamento. Il **punto di Elbow** consente di identificare questo compromesso. Nel caso di esempio, un buon compromesso sarebbe quello di avere $k = 3$ o $k = 4$.

Silhouette coefficient

Il **silhouette coefficient** è una misura della coesione e separazione tra i dati. In particolare, quantifica quanto i dati siano ben disposti nei cluster generati.

Il **coefficiente** si basa su due parametri:

- quanto bene i dati sono *ammassati* nel cluster di riferimento;
- quanto è *distante* ciascun campione da qualsiasi altro cluster.

Il coefficiente varia tra **-1** e **+1**. Valori alti indicano condizioni maggiori di coesione e di separazione dei dati. Il coefficiente, per un punto i -esimo appartenente al cluster c , è calcolato tramite la seguente formula:

$$s(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))}$$

- $a(i)$ rappresenta la distanza media dell' i -esimo punto rispetto a tutti gli altri punti appartenente allo stesso cluster;
- $b(i)$ rappresenta la distanza media dell' i -esimo punto rispetto a tutti gli altri punti appartenente al cluster più vicino del cluster a cui è stato assegnato;
- $s(i)$ rappresenta il coefficiente di silhouette dell' i -esimo punto. Il valore finale di silhouette è dato dalla media dei coefficienti di silhouette calcolati per ogni elemento del problema.

Mojo distance

La **Move-Join (MoJo) distance** è una metrica che calcola il numero minimo di operazioni di spostamento e raggruppamento di elementi necessari per passare dal clustering, identificato da un algoritmo, al clustering ideale degli elementi.

Ma se ho a disposizione un oracolo che riporta il raggruppamento ideale, perché dovrei fare clustering con un algoritmo?

Per valutare un algoritmo di clustering prima di utilizzarlo su nuovi dati sconosciuti. Inoltre, una valutazione fatta tramite il calcolo della **MoJo distance** ci dà maggiore confidenza nell'utilizzare i risultati del clustering e può abilitare più facilmente l'utilizzo del clustering come mezzo per altre operazioni.

