

L'Intelligenza Artificiale è l'abilità di una macchina di mostrare capacità umane quali il ragionamento, l'apprendimento, la pianificazione e la creatività.

Gli esseri umani fanno riferimento a sé stessi con il termine "*Homo Sapiens*", poiché ritengono che la loro intelligenza sia importante. Per migliaia di anni abbiamo cercato di comprendere come pensiamo, ovvero come riusciamo a percepire, capire, predire e manipolare un mondo così grande e complicato. Il campo dell'intelligenza artificiale va ancora più in là: il suo obiettivo non è solo quello di comprendere, ma anche di **costruire entità intelligenti**.

Intelligenza Artificiale, cos'è?

L'IA ha diverse interpretazioni riguardo la sua definizione: alcune definizioni riguardano i processi di pensiero e il ragionamento e quindi si misura il successo dell'IA in termini di fedeltà alle prestazioni umane; altre, invece, riguardano il comportamento quindi si misura il successo dell'IA in termini di prestazione ideale cioè in termini di razionalità. Un sistema è razionale se fa la cosa giusta, dato quello che sa.

Tutti questi approcci sono stati storicamente seguiti, ciascuno con metodi diversi. Un approccio centrato sull'uomo deve essere in parte una scienza empirica, coinvolgendo osservazioni e ipotesi sul comportamento umano. Un approccio razionalista, invece, comporta una combinazione di matematica e ingegneria.

La definizione di Intelligenza Artificiale comprende diversi aspetti e, pertanto, esistono più definizioni capaci di descriverla.

- **Pensare umanamente:** il tentativo di far sì che i computer arrivino a pensare. In termini pratici, *automatizzare le attività che associamo al pensiero umano*, come il processo decisionale, la risoluzione di problemi, l'apprendimento, e altro;
- **Pensare razionalmente:** lo studio delle facoltà mentali attraverso l'uso di modelli computazionali. Ovvero, lo studio dei processi di calcolo che rendono possibile percepire, ragionare e agire;
- **Agire umanamente:** l'arte di creare macchine che eseguono attività che richiedono intelligenza quando vengono svolte da persone. Lo studio di come fare eseguire ai computer attività che l'essere umano, al momento, svolge meglio;
- **Agire razionalmente:** l'Intelligenza Computazionale è lo studio della progettazione di agenti intelligenti. L'Intelligenza Artificiale riguarda il comportamento intelligente di questi artefatti.

Pensare umanamente

L'approccio della modellazione cognitiva

Quando diciamo che un determinato programma *ragiona* come un essere umano, dobbiamo prima di tutto determinare come *noi pensiamo*, ovvero capire il funzionamento effettivo della mente umana e ci sono tre modi per farlo:

- **Introspezione**: cercando di catturare i nostri pensieri “al volo”;
- **Sperimentazione psicologica**: osservando una persona in azione;
- **Imaging cerebrale**: osservando il cervello in azione.

Una volta che abbiamo una teoria della mente sufficientemente precisa, diventa possibile esprimere la teoria come un programma per computer.

I veri avamposti dell’Intelligenza Artificiale: la psicologia e le neuroscienze

Psicologia: Come pensano ed agiscono gli esseri umani e gli animali?

Neuroscienze: Come avviene l’elaborazione dell’informazione da parte del cervello?

Psicologia cognitiva

La psicologia cognitiva vede il cervello come un dispositivo di elaborazione delle informazioni.

Secondo Kenneth Craik, un **agente intelligente** possiede tre requisiti fondamentali:

1. Lo stimolo deve essere tradotto in una rappresentazione interna;
2. La rappresentazione deve essere manipolata da processi cognitivi per ottenere nuove rappresentazioni interne;
3. Tali nuove rappresentazioni devono essere a loro volta trasformate in azioni.

“Se l’organismo porta nella sua testa un “modello in scala” della realtà esterna e delle proprie possibili azioni allora sarà in grado di provare varie alternative, decidere quali di esse sia la migliore, reagire a situazioni future prima che si manifestino, utilizzare la conoscenza di eventi passati per gestire quelli presenti e futuri, e sotto ogni aspetto reagire in modo molto più completo, affidabile e competente alle emergenze che si troverà ad affrontare.”

(Definizione di intelligenza artificiale di Craik)

Il lavoro di Craik fu continuato da Broadbent, il quale fu tra i primi a modellare i fenomeni psicologici basandosi sull’elaborazione dell’informazione, dando vita alla **scienza cognitiva**.

All’interno delle scienze cognitive, un ruolo importante è svolto dalle cosiddette “neuroscienze”.

Neuroscienze: il cervello come causa della mente

Le neuroscienze si occupano dello studio del sistema nervoso e, in particolare, del cervello. Il modo in cui si origina un pensiero è tuttora ignoto, pertanto nelle neuroscienze il cervello è considerato il mezzo tramite il quale i pensieri vengono resi possibili.

Secondo Searle, risulta stupefacente come “una collezione di semplici cellule può condurre al pensiero, all’azione e alla consapevolezza: [...] il cervello è causa della mente”. Queste semplici cellule di cui parla Searle non sono altro che i **neuroni**, cellule nervose tipicamente collegate tra loro tramite punti di congiunzione chiamati **sinapsi**. I segnali si propagano da un neurone all’altro

grazie ad una complicata reazione elettrochimica e controllano l'attività cerebrale nel breve periodo, ma permettono anche dei cambiamenti a lungo termine nelle connessioni tra i neuroni: si ritiene che questo meccanismo formi la base dell'apprendimento.

Pensare razionalmente

L'approccio delle leggi del pensiero

I sillogismi aristotelici hanno rappresentato il primo tentativo di codificare formalmente il pensiero corretto, ovvero i processi di ragionamento inconfutabili. Si ritiene che queste leggi abbiano dato origine alla disciplina della *logica*.

La tradizione logicista

I logicisti del XIX secolo hanno sviluppato una notazione precisa per formulare enunciati riguardanti tutti gli oggetti del mondo e le relazioni tra essi.

L'idea dei logicisti è quella di partire dagli enunciati logici per poter costruire sistemi intelligenti e razionali. Tuttavia, sussistono due problemi:

1. Non è facile esprimere una conoscenza non formalizzata in termini strettamente formali, specialmente in casi in cui la conoscenza non è sicura al 100%;
2. Forse più importante, c'è una grande differenza tra l'essere in grado di risolvere un problema "in linea di principio" e farlo nella pratica.

Pertanto, anche problemi con poche centinaia di fatti possono esaurire le risorse computazionalmente disponibili, *a meno che* non siano forniti all'elaboratore degli strumenti con cui poter guidare i passi del ragionamento. Questa rappresenta una sfida aperta nel contesto dell'Intelligenza Artificiale.

Agire umanamente

L'approccio al test di Turing

Il test di Turing è stato concepito con l'obiettivo di fornire una soddisfacente definizione operativa dell'intelligenza. Il test è anche noto come "The Imitation Game".

Supponiamo l'esistenza di tre protagonisti: una macchina (A), una donna (B) e un interrogante (C). Consideriamo anche l'esistenza di due stanze: C sarà isolato in una, mentre A e B condivideranno l'altra stanza.

L'identità di A e B non è nota a C e, infatti, quest'ultimo li conoscerà come due persone ignote.

Obiettivo di C: determinare quale delle due entità sia la macchina e quale sia la donna attraverso le domande che porrà.

Obiettivo di A: ingannare C e indurlo in errore.

Obiettivo di B: aiutare C nella corretta identificazione.

In altri termini, C porrà delle domande e, sulla base delle risposte ottenute, dovrà capire chi tra A e B è la macchina. A, invece, assumerà il ruolo dell'imitatore, ovvero colui che dovrà rispondere come se fosse B.

Sfortunatamente Turing ricevette diverse obiezioni in merito. (erano tutti dei cristiani cacasotto)

Ma quindi: può una macchina agire umanamente?

Esistono diverse limitazioni matematiche a riguardo e, principalmente, limitazioni alle capacità delle macchine a stati discreti. Il più conosciuto è noto come il *teorema di Gödel*: esso dimostra che in qualsiasi sistema logico sufficientemente potente si possono formulare proposizioni di cui non si riesce a dare una dimostrazione né di esse, né della loro negazione, all'interno del sistema, derivandone così la possibilità dell'incoerenza dello stesso sistema logico.

Lo stesso Turing ottenne risultati simili. Tuttavia, vale la pena sottolineare che le stesse limitazioni evidenziate dal teorema di Gödel potrebbero valere anche per l'intelletto umano. Non esiste, ad oggi, nessuna dimostrazione a proposito.

Agire razionalmente

L'approccio degli agenti razionali

Un agente è semplicemente qualcosa che agisce, che fa qualcosa. Tutti i programmi fanno qualcosa, ma si suppone che gli agenti facciano di più: operare autonomamente, essere in grado di percepire l'ambiente esterno e raggiungere obiettivi.

Agenti razionali

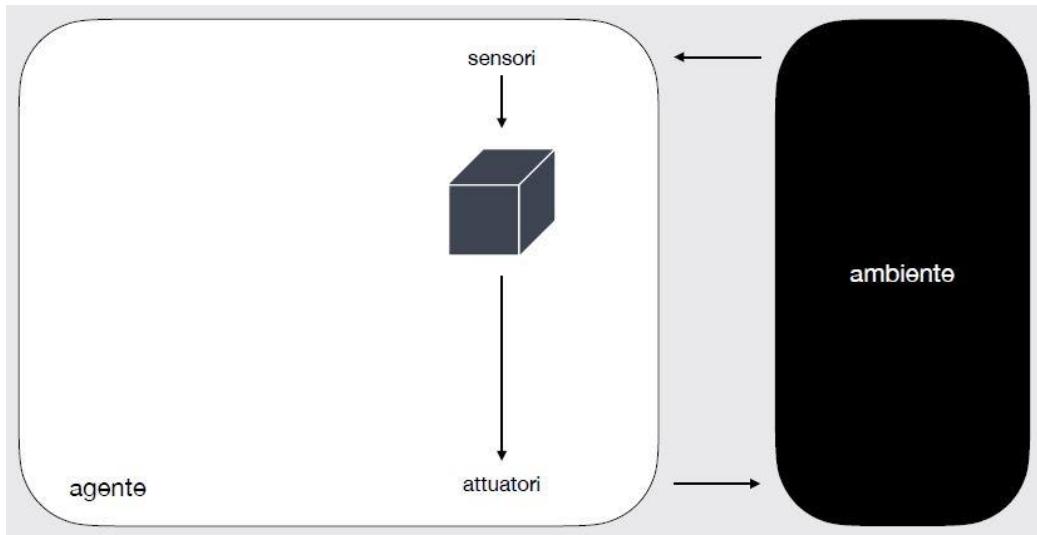
Un agente razionale agisce in modo da ottenere il miglior risultato o, in condizione di incertezza, il miglior risultato atteso. Tuttavia, essere in grado di formulare deduzioni corrette rappresenta solo una parte di un agente razionale. In alcune situazioni non si può dimostrare l'esistenza di un'azione "giusta" da fare seppur qualcosa andrebbe fatto.

Le abilità richieste dal test di Turing consentono ad un agente di agire razionalmente.

In parole poche, la razionalità è un concetto che **include** gli aspetti precedenti, ma che in più propone l'idea di "adattarsi" al contesto riuscendo ad agire nel miglior modo possibile sulla base delle informazioni disponibili.

Agenti intelligenti

Un **agente** è qualsiasi cosa possa esser vista come un sistema che *percepisce* il suo **ambiente** attraverso dei **sensori** ed *agisce* su di esso tramite degli **attuatori**.



La **percezione** è un insieme di input percettivi dell'agente in un dato istante. Mentre la **sequenza percettiva** è la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza.

Più in generale, è bene dire che la scelta di un'azione da parte dell'agente può dipendere dall'intera sequenza percettiva osservata fino a quel momento, ma **non** da qualcosa che abbia percepito. Se volessimo dirla in termini matematici, allora il comportamento di un agente è descritto da una **funzione agente**, che descrive la corrispondenza tra una qualsiasi sequenza percettiva ed una specifica azione.

La sequenza percettiva e la corrispondente azione attuata dall'agente può essere espressa attraverso una tabella a due colonne. Il ché ci porta ad una domanda: Qual è il modo corretto di progettare la tabella? O in altri termini: Cosa rende un agente buono o cattivo, intelligente o stupido?

La **razionalità** di un agente dipende da quattro fattori:

- La misura delle prestazioni che definisce il criterio di successo;
- La conoscenza pregressa dell'ambiente da parte dell'agente;
- Le azioni che l'agente può eseguire;
- La sequenza percettiva dell'agente fino ad oggi.

Per ogni possibile sequenza di percezioni, un agente razionale dovrebbe scegliere un'azione che massimizzi il valore atteso della sua misura di prestazione, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente. Questa **misura di prestazione** però dovrebbe essere progettata sulla base dell'*effetto che si desidera osservare sull'ambiente e non su come dovrebbe comportarsi l'agente*. Altrimenti, un agente otterebbe una razionalità perfetta semplicemente "illudendosi" di fare la cosa giusta.

Però bisogna ricordare che razionalità non implica onniscienza. Un **agente onnisciente** conosce il risultato effettivo delle sue azioni e può agire di conseguenza. Sfortunatamente, nel mondo reale l'onniscienza è impossibile. Un **agente razionale** fa la cosa giusta sulla base del contesto in cui opera. Razionalità, pertanto, non significa perfezione ma piuttosto **massimizzazione del risultato atteso** che può essere raggiunta grazie ad azioni di *information gathering*, ovvero azioni che hanno la sola finalità 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 far "conoscenza" di un ambiente sconosciuto.

Oltre che l'esplorazione, la definizione di razionalità prevede il concetto di **apprendimento**, ovvero la capacità di imparare coppie di percezione-azione sulla base delle azioni e dei rispettivi 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*.

Definizione di ambiente

Un **ambiente** è essenzialmente un'istanza di un problema di cui gli agenti razionali rappresentano le soluzioni.

Un ambiente viene generalmente descritto tramite la formulazione **PEAS**:

- **Performance**: misura di prestazione adottata 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, un agente, riceve gli input percettivi.

Proprietà degli ambienti

La gamma di ambienti di attività che potrebbero sorgere nell'IA è molto vasta. Possiamo, tuttavia, identificare un numero abbastanza ridotto di dimensioni con le quali possiamo classificare gli ambienti di attività. Queste dimensioni determinano, in larga misura, l'appropriata progettazione dell'agente e l'applicabilità di ciascuna delle principali famiglie di tecniche per l'implementazione dell'agente.

- **Completamente osservabili vs parzialmente osservabili**

Se i sensori di un agente gli danno un accesso allo stato completo dell'ambiente, in ogni momento, allora diciamo che l'ambiente dell'attività è completamente osservabile. Questi tipi di ambienti sono convenienti perché l'agente non ha bisogno di tenere uno stato interno per tenere traccia del mondo.

Un ambiente, invece, potrebbe essere parzialmente osservabile a causa di sensori rumorosi e imprecisi o semplicemente perché mancano dei dati, il sensore non riesce a catturare tutti i dati necessari.

- **Deterministico vs stocastico**

Se il prossimo stato dell’ambiente è completamente determinato dallo stato corrente e dall’azione corrente eseguita dall’agente, allora diciamo che l’ambiente è deterministico; altrimenti, l’ambiente è stocastico. In linea di principio, un agente non deve preoccuparsi dell’incertezza in un ambiente deterministico completamente osservabile. Se l’ambiente è parzialmente osservabile, tuttavia, potrebbe sembrare stocastico. La maggior parte delle situazioni reali sono così complesse che è impossibile tenere traccia di tutti gli aspetti non osservati; ai fini pratici, devono essere trattati come stocastici.

Diciamo che un ambiente è *incerto* se non è completamente osservabile o non deterministico.

L’uso della parola “stocastico” implica generalmente che l’incertezza sui risultati è quantificata in termini di probabilità; un ambiente non determinista è quello in cui le azioni sono caratterizzate dai loro possibili esiti, ma ad esse non sono associate probabilità. Le descrizioni di un ambiente non determinista sono solitamente associate a misure di prestazione che richiedono all’agente di avere successo per tutti i possibili risultati delle sue azioni.

- **Episodico vs Sequenziale**

In un ambiente episodico, l’esperienza dell’agente è divisa in episodi atomici, in cui l’agente riceve una percezione ed esegue una singola azione. Fondamentalmente, il prossimo episodio non dipende dalle azioni intraprese negli episodi precedenti.

In ambienti sequenziali, invece, la decisione attuale potrebbe influenzare tutte le decisioni future.

Un ambiente episodico è molto più semplice di un ambiente sequenziale perché l’agente non deve pensare alle azioni che dovrà compiere.

- **Statico vs dinamico**

Se l’ambiente può cambiare mentre l’agente sta deliberando, allora diciamo che l’ambiente è dinamico per quell’agente; in caso contrario, è statico.

Gli ambienti statici sono facili da gestire perché l’agente non deve continuare a guardare il mondo mentre sta decidendo un’azione, né deve preoccuparsi del passare del tempo. Gli ambienti dinamici, d’altra parte, chiedono continuamente all’agente cosa vuole fare; se non ha ancora deciso, conta come se avesse preso la decisione di non fare nulla.

Se l’ambiente stesso non cambia con il passare del tempo, ma cambia il punteggio di prestazione dell’agente, allora diciamo che l’ambiente è semidinamico.

- **Discreto vs continuo**

La distinzione discreto/continuo si applica allo stato dell’ambiente, al modo in cui viene gestito il tempo e alle percezioni e azioni dell’agente. Ad esempio il gioco degli scacchi è un ambiente discreto, mentre l’auto a guida autonoma è un esempio di ambiente continuo.

- **Singolo vs multi-agente**

La distinzione tra ambienti ad agente singolo e multi-agente può sembrare abbastanza semplice. Ad esempio, un agente che risolve un cruciverba da solo si trova chiaramente in un ambiente con un solo agente, mentre un agente che gioca a scacchi si trova in un ambiente con due agenti. Ci sono, tuttavia, alcuni problemi sottili.

In primo luogo, abbiamo capito come un'entità la si può considerare un agente, ma non abbiamo capito quali entità devono essere viste come agenti. Un agente A (un taxi a guida autonoma) deve trattare un oggetto B (un altro veicolo) come un agente, oppure può essere trattato semplicemente come un oggetto che si comporta secondo le leggi della fisica?

La distinzione chiave è se il comportamento di B è meglio descritto come la massimizzazione di una misura di performance il cui valore dipende dal comportamento dell'agente A. Pertanto, gli scacchi sono un ambiente **competitivo** multi-agente.

Nell'ambiente di guida dei taxi, invece, evitare le collisioni massimizza la misura di performance di tutti gli agenti, quindi è un ambiente multi-agente parzialmente **cooperativo**. È anche parzialmente competitivo perché, ad esempio, una sola auto può occupare un posto auto.

I problemi di progettazione degli ambienti multi-agente sono spesso molto diversi da quelli degli ambienti a singolo agente.

Struttura degli agenti

Il compito dell'IA è progettare il **programma agente**, il quale consiste nell'implementazione della funzione agente.

Possiamo dire quindi che un agente è formato da architettura + programma, dove per architettura si intendono i sensori e gli attuatori dell'agente, mentre il programma prende in input la percezione corrente dei sensori e restituisce 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.

Più in generale, la sfida principale dell'IA è trovare il modo di scrivere programmi che, nella massima misura possibile, producano comportamento razionale con una **piccola quantità di codice** invece che con la **rappresentazione di tutti gli input percettivi possibili**.

Proprio alla luce di ciò, un approccio basato su tabelle è fallimentare, basti pensare di risolvere il gioco degli scacchi con le tabelle (cosa allucinante). Fare uso quindi di **learning agents**, ovvero agenti capaci di migliorare le loro prestazioni e attuare migliori azioni attraverso l'apprendimento.

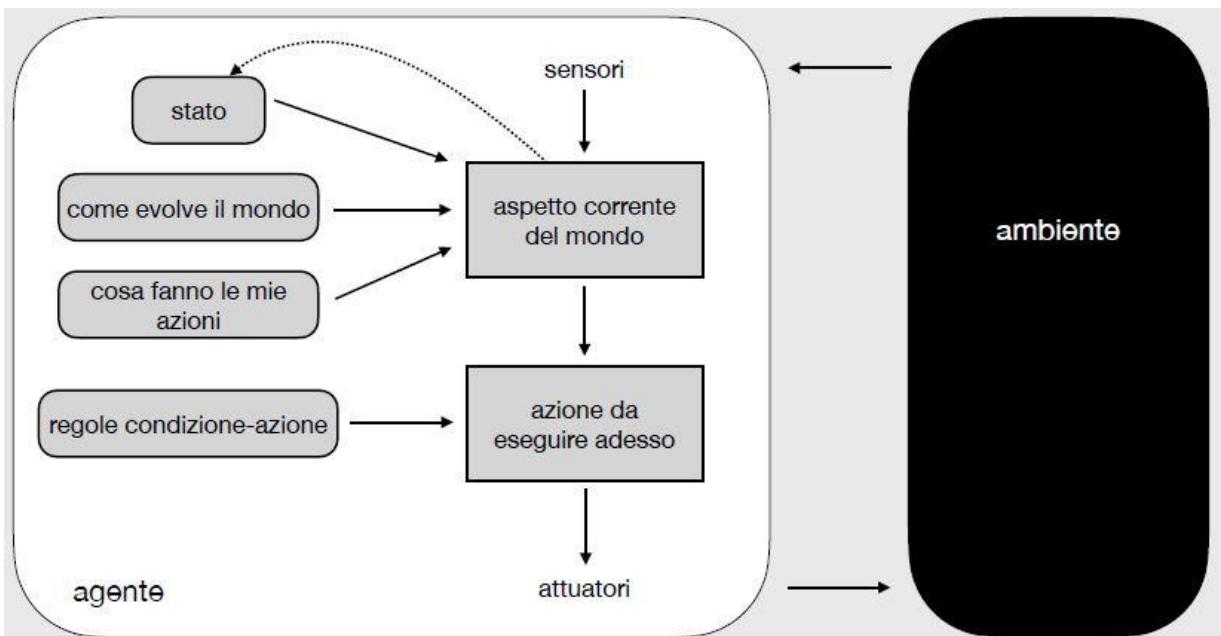
Agenti reattivi semplici

Questi agenti intraprendono azioni sulla base della percezione corrente, ignorando la storia percettiva pregressa. Pertanto, non hanno memoria.

In alcuni casi, un agente reattivo semplice che opera in ambienti parzialmente osservabili può incappare in cicli 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. Una componente casuale porta solitamente ad un comportamento razionale, ma non è questo il caso: la componente aleatoria serve solo ad un agente reattivo semplice di poter evitare cicli infiniti, ma non “crea” intelligenza.

Agenti reattivi basati su modello

Un agente con due tipi di conoscenza: come evolve il mondo, indipendentemente dal suo stato; informazioni sull'impatto delle sue azioni sull'ambiente.



La conoscenza del mondo, sviluppata mediante una teoria scientifica completa, viene chiamata “*modello*” del mondo.

```
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;

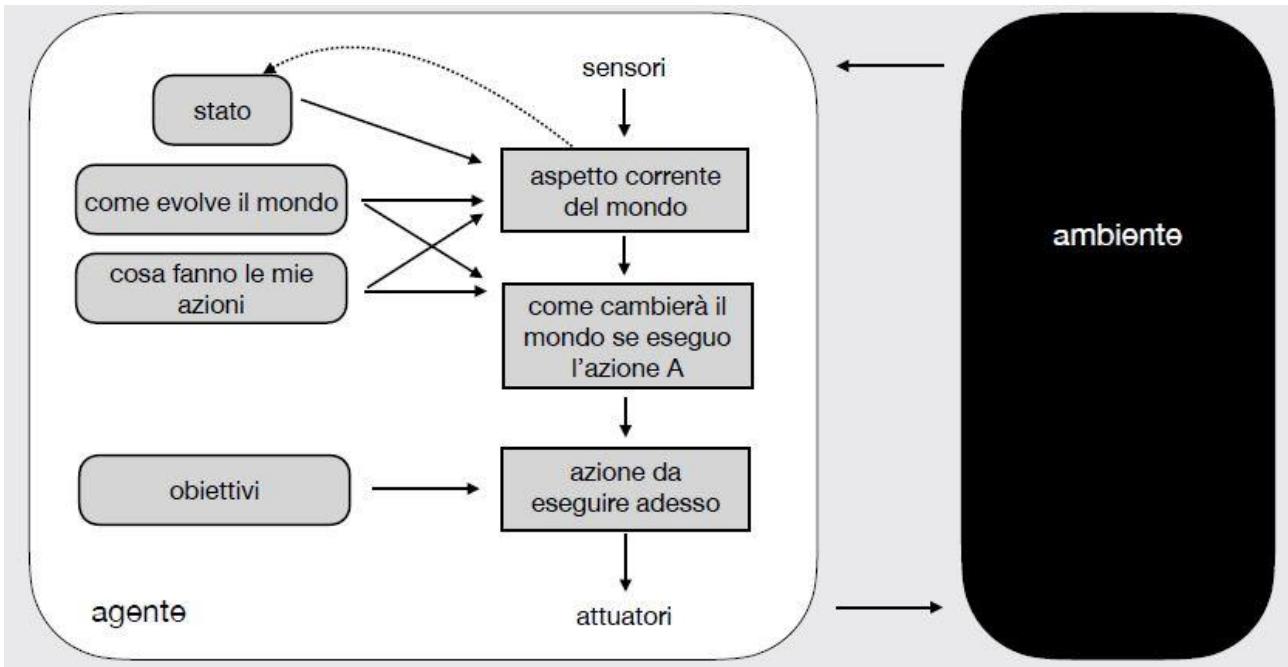
  stato <- AGGIORNA-STATO(stato, azione, percezione, modello)
  regola <- REGOLA-CORRISPONDENTE(stato, regole)
  azione <- regola.AZIONE

  return azione
```

È importante notare che la variabile “stato” non necessariamente rappresenta con esattezza lo stato del mondo, ma è la migliore ipotesi che l’agente può fare sul suo stato.

Agenti basati su obiettivi

Un agente che aggiunge al modello del mondo, informazioni sugli obiettivi specifici che si desidera raggiungere.



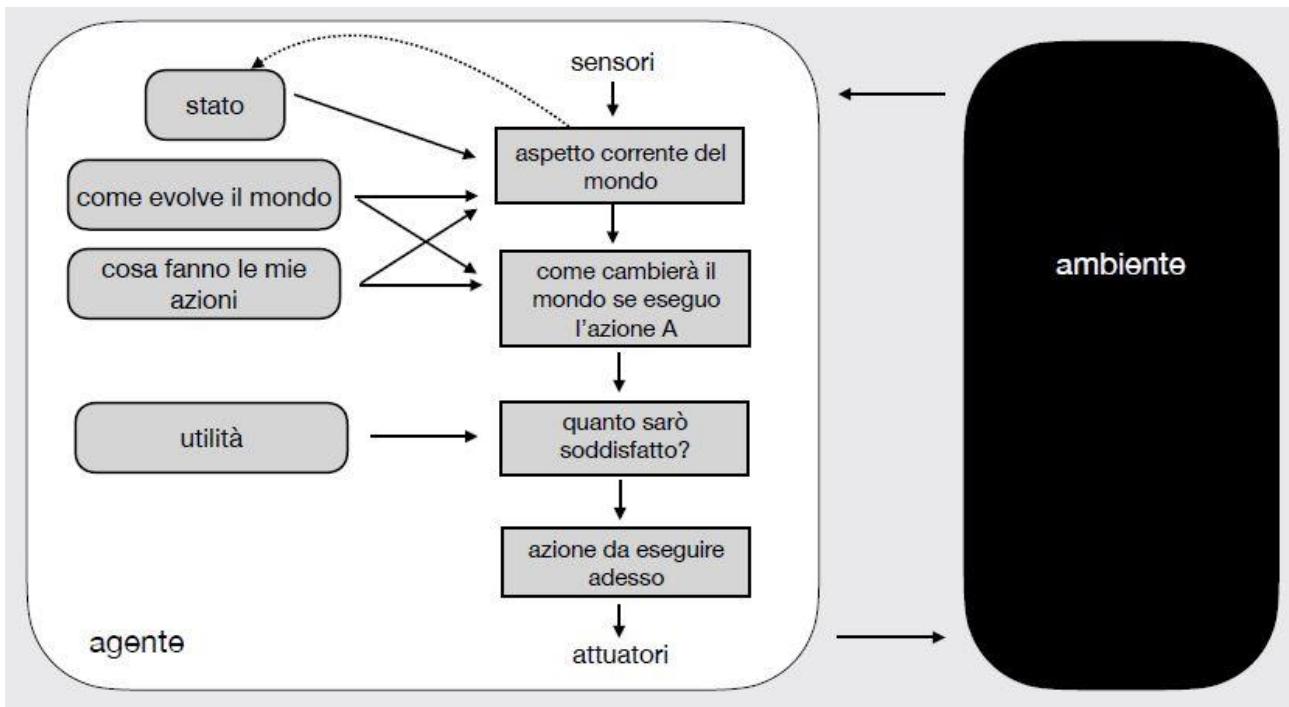
In altri termini, conoscere lo stato corrente dell’ambiente non sempre basta a decidere cosa fare. Per questo incorporare degli obiettivi può aiutare l’agente a compiere azioni più intelligenti e razionali che portino ad una soluzione più rapida.

Attenzione però: il tipo di decisioni prese da un agente di questo tipo non ha niente a che vedere con le regole condizione-azione viste per gli agenti basati su modello. Infatti, in questo caso dobbiamo prendere in considerazione il futuro sotto due aspetti: *Cosa accadrà se faccio così? E se faccio questo verrò 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

Gli obiettivi non bastano in quanto consentono di esprimere in maniera binaria i risultati “buoni” da quelli “cattivi”. In situazioni reali, esistono situazioni “desiderabili” e non.



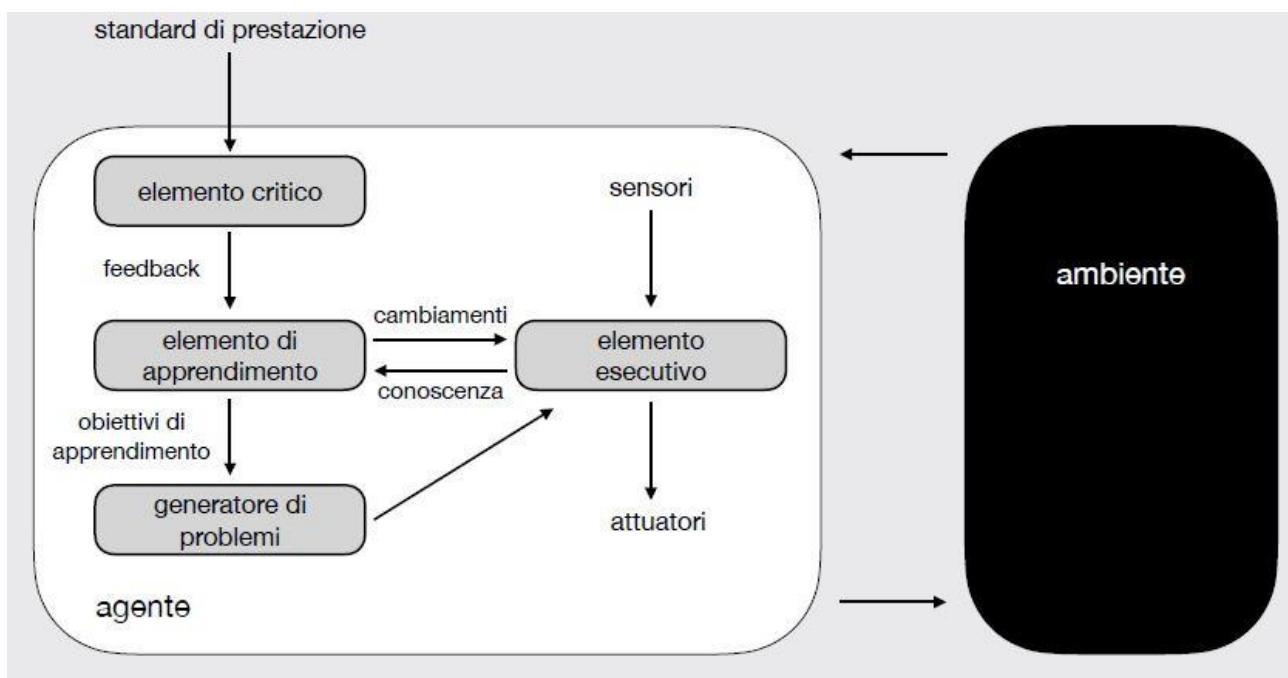
Una funzione di utilità assegna ad uno stato un numero reale che quantifica il grado di “desiderabilità” ad esso associato. Quando ci sono più obiettivi tra di loro contrastanti, una funzione di utilità consente di determinare quale obiettivo preferire in base al punteggio ad esso assegnato dalla funzione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

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;
- **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 significative che, chiaramente, portino l'agente ad apprendere nuove conoscenze da sfruttare poi per migliorare le sue azioni.



Come funzionano i componenti dei programmi agente

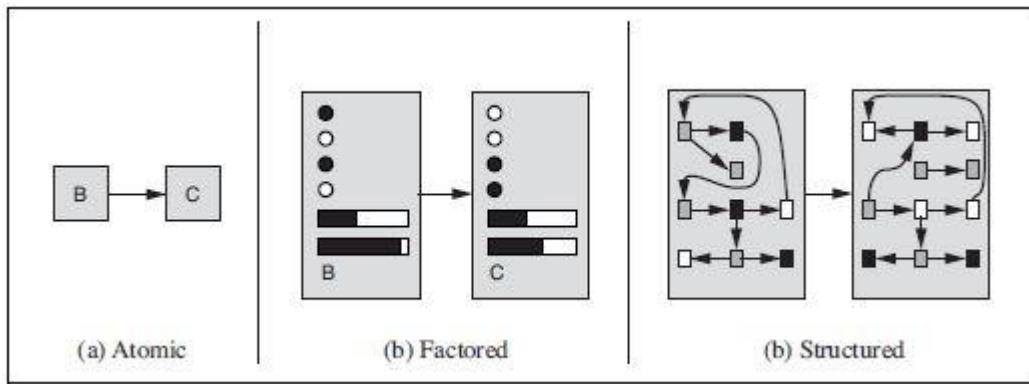
Abbiamo descritto finora i programmi agente come costituiti da vari componenti la cui funzione è di rispondere a domande quali: "Com'è il mondo adesso?", "Che azione dovrei fare ora?", "Cosa fanno le mie azioni?". Ora la prossima domanda che deve porsi uno studente di IA è: "Come cazzo funzionano questi componenti?".

In parole poche, possiamo rappresentare il mondo esterno in tre modi differenti: **atomico**, **fattorizzato** e **strutturato**.

In una rappresentazione atomica ogni stato del mondo è indivisibile, non ha una struttura interna. L'unica proprietà riconoscibile di uno stato è il fatto di essere uguale o diverso da un altro. Gli algoritmi alla base di ricerca e giochi lavorano tutti con rappresentazioni atomiche o, quantomeno, considerano le rappresentazioni come se fossero atomiche.

Una rappresentazione **fattorizzata** suddivide ogni stato in un insieme fisso di **variabili** o **attributi**, ognuno dei quali può avere un **valore**. Sebbene due diversi stati atomici non abbiano nulla in comune, due diversi stati fattorizzati possono condividere alcuni attributi e questo rende molto più facile capire come trasformare uno stato in un altro. Con le rappresentazioni fattorizzate, possiamo anche rappresentare l'incertezza: ad esempio, se non si conosce un valore di un determinato attributo lo si può lasciare vuoto.

Infine, si usa una rappresentazione **strutturata** quando c'è da descrivere uno stato complesso che include degli oggetti i quali, oltre ad avere i propri attributi, sono relazionati con altri oggetti sempre interni allo stato.



Agenti risolutori di problemi di ricerca

Gli **agenti risolutori di problemi** sono un particolare tipo di agenti intelligenti. Essi si pongono uno o più obiettivi da raggiungere ed eseguono azioni che massimizzino le chance del loro raggiungimento. Utilizzano, pertanto, rappresentazioni atomiche del mondo, i cui stati verranno usati, via via, per trovare una soluzione accettabile al problema trattato.

Supponiamo che un agente X sia in vacanza ad Arad, in Romania. La sua **misura di prestazione** sarà composta da diversi fattori: vorrà farsi una bella tintarella, migliorare il suo rumeno, ammirare i panorami più belli, passare notti divertenti, evitare i postumi di una sbronza, e così via.

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. Dovrà evitare eccessi come, ad esempio: tornare a casa alle 7 del mattino completamente ubriaco; passare troppo tempo a Palazzo Cenad nel tentativo di apprezzare la combinazione di stili architettonici di esso, ecc.

Supponiamo adesso che l'agente abbia un biglietto aereo non rimborsabile per la partenza da Bucarest il giorno successivo. Dato questo presupposto, sembra sensato che l'agente adotti l'**obiettivo** di arrivare a Bucarest in tempo utile per potersi imbarcare. Tutte le azioni che non consentiranno all'agente di raggiungere Bucarest in tempo possono essere scartate. In termini pratici, questo implica che il cosiddetto **spazio delle soluzioni** potrà essere sostanzialmente ridotto, velocizzando il raggiungimento di una soluzione.

Quindi abbiamo capito che il primo passo da fare nella risoluzione dei problemi è quello della **formulazione dell'obiettivo**, che è basata sulla situazione corrente e sulla misura di prestazione

dell'agente. L'**obiettivo** è rappresentato da tutti e soli quegli stati (del mondo) in cui l'obiettivo è soddisfatto.

Il compito dell'agente è determinare come agire, ora e nel futuro, per raggiungere uno stato obiettivo. Per fare ciò, l'agente deve decidere quali tipi di azioni e stati prendere in considerazione. Se l'agente cercasse di considerare azioni di "basso livello", quali:

- "muovi il piede sinistro di un centimetro"
- "gira il volante a sinistra di un grado"

Allora con ogni probabilità non riuscirebbe neanche ad uscire dal parcheggio. A quel grado di dettaglio c'è troppa incertezza e una soluzione sarebbe composta da un numero eccessivo di passi. Pertanto l'agente dovrà scegliere una **giusta granularità**.

Altro passo per la risoluzione di un problema è la **formulazione del problema**, ovvero il processo che porta a decidere, dato un obiettivo, quali azioni e quali stati considerare.

Supponiamo, quindi, che l'agente esaminerà azioni al livello di guidare da una grande città ad un'altra. Ogni stato corrisponderà a trovarsi in una particolare città. Ci sono tre strade che portano fuori città: una in direzione Sibiu, una verso Timisoara, la terza verso Zerind. Chiaramente nessuna opzione soddisfa l'obiettivo; pertanto, l'agente non saprà dove andare. In altri termini, in questo stato l'agente **non possiede abbastanza informazioni sullo stato risultante da un'azione** (cioè non sa cosa succede dopo aver compiuto una determinata azione).

Se l'ambiente fosse ignoto, l'agente dovrebbe scegliere un'azione a caso. Se supponessimo invece che l'agente abbia una mappa della Romania, allora potrebbe collezionare informazioni sull'ambiente circostante. Una volta ottenute tali informazioni, l'agente può usarle per considerare i *passi successivi*.

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**.

Cosa significa esaminare le azioni future?

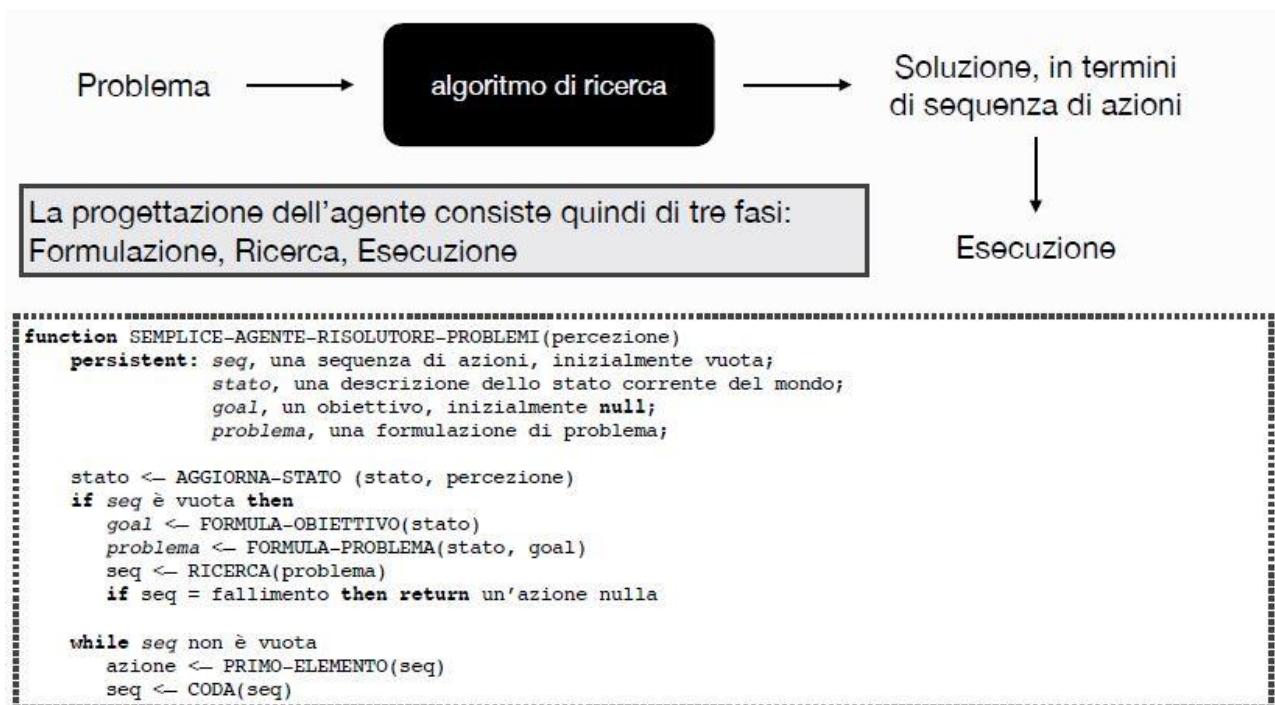
Ipotizziamo che l'ambiente sia **osservabile**, pertanto l'agente conosce *sempre* lo stato corrente del mondo. Ipotizziamo che l'ambiente sia **discreto**, pertanto in qualsiasi stato esiste un numero finito di azioni tra cui scegliere. Ipotizziamo che l'ambiente sia **noto**, pertanto l'agente saprà sempre quali stati saranno raggiunti da ciascuna azione. Ipotizziamo che l'ambiente sia **deterministico**, pertanto ogni azione avrà uno ed un solo risultato.

Sotto queste ipotesi, 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 giunte.

In una situazione del genere, l'agente in esempio saprebbe esattamente dove si troverà dopo la prima zione e che cosa percepirlà, e così via. Così facendo, potrebbe identificare una soluzione specificata da una sequenza di azioni che porti da una città all'altra fino a Bucarest.

Ricerca

Il processo che cerca una sequenza di azioni che porti al raggiungimento dell'obiettivo è detto **ricerca**.



1. La funzione prende in input la percezione corrente dell'agente.
2. Nella prima fase, le variabili necessarie alla risoluzione del problema sono instanziate.
3. Aggiornato lo stato, dato lo stato precedente e la percezione attuale, l'agente formulerà un obiettivo e un problema. Dopodiché, avvierà l'algoritmo di ricerca.
4. 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.

NB: 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.

Problemi ben definiti e soluzioni

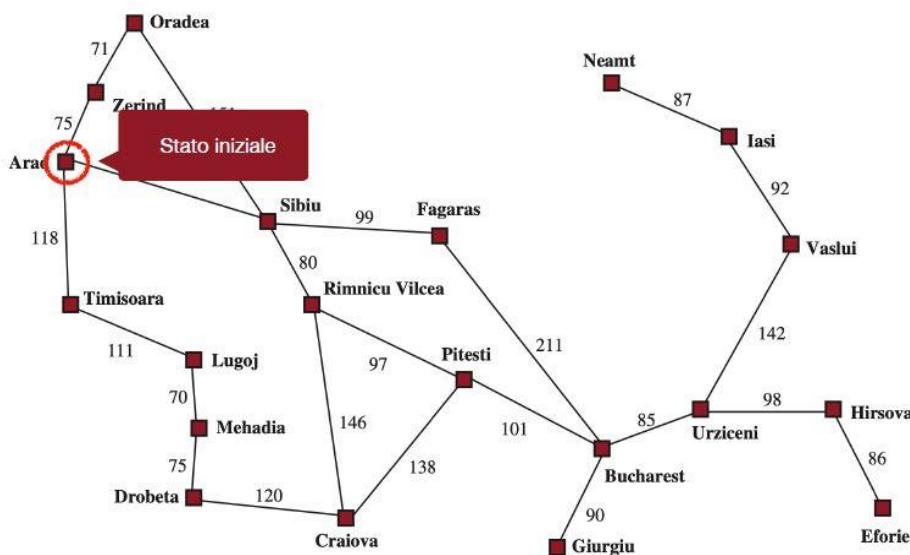
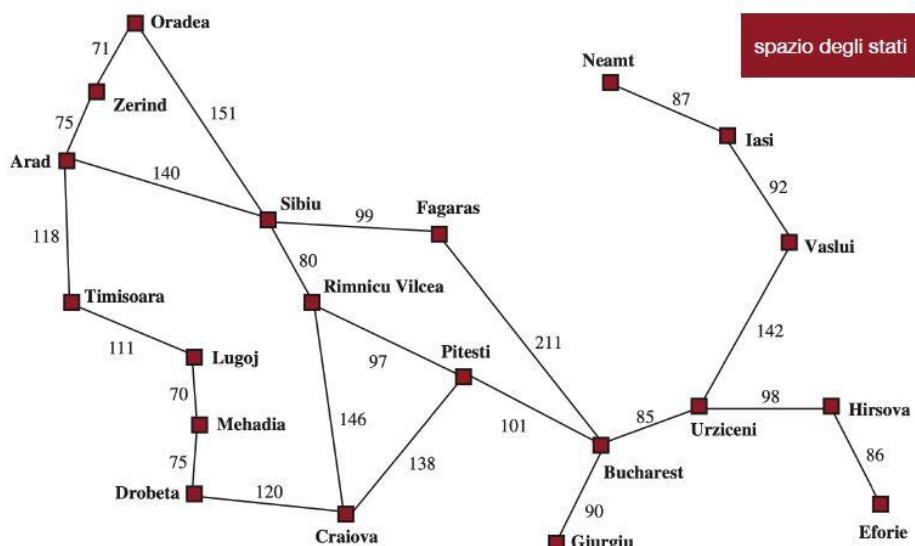
Un **problema** può essere definito formalmente da cinque componenti:

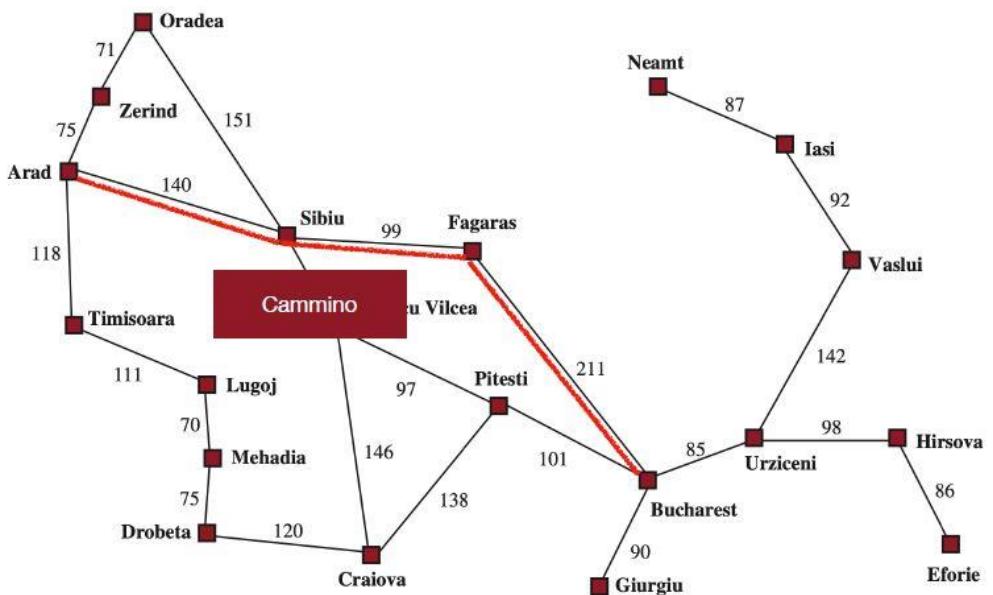
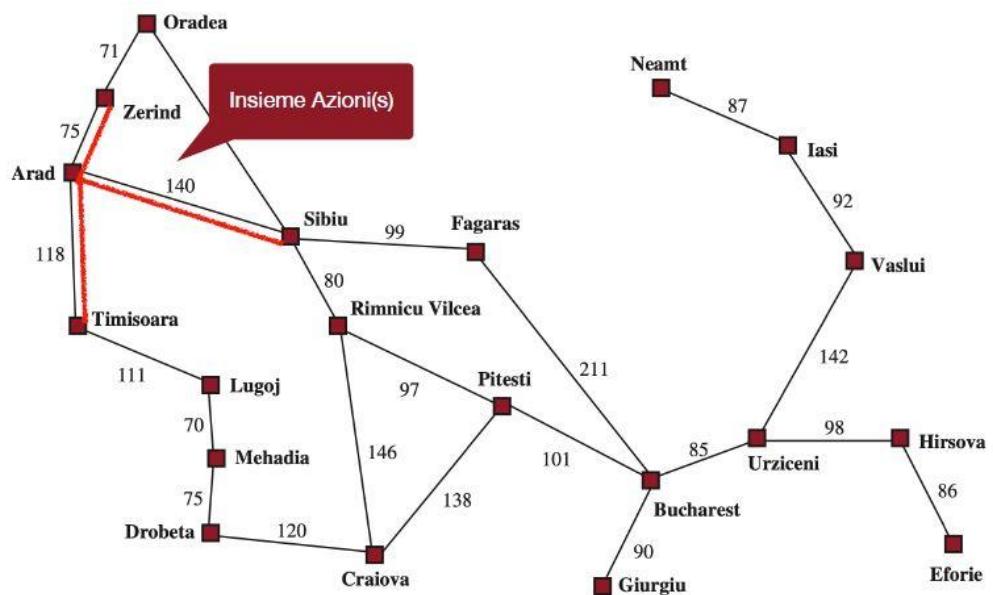
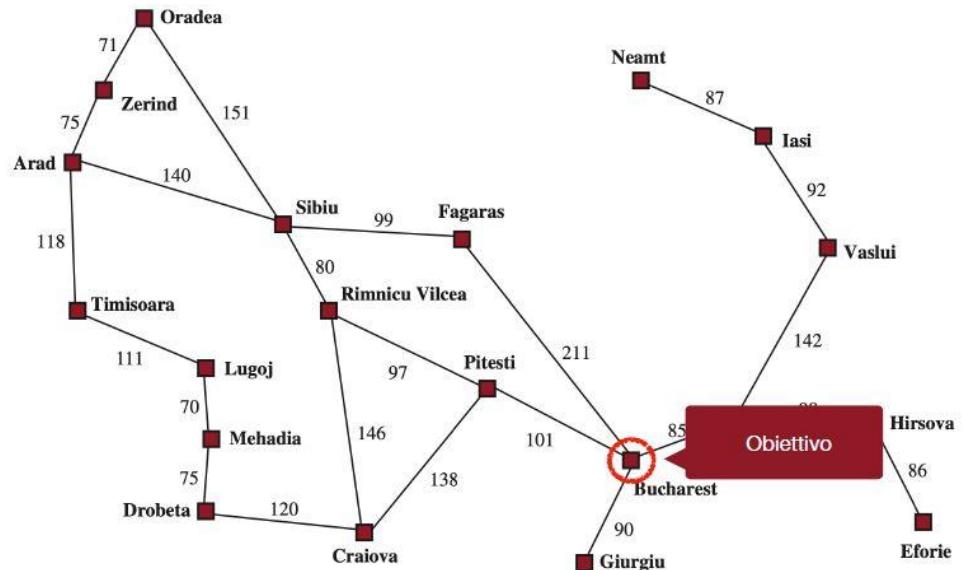
- **Stato iniziale:** rappresenta il punto di partenza dell'agente. Nell'esempio precedente. Lo stato iniziale potrebbe essere descritto come *in(Arad)*;
- **Azioni:** questo insieme include la descrizione delle possibili operazioni attuabili dall'agente. Più formalmente, dato uno stato *s*, *Azioni(s)* 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)*};
- **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

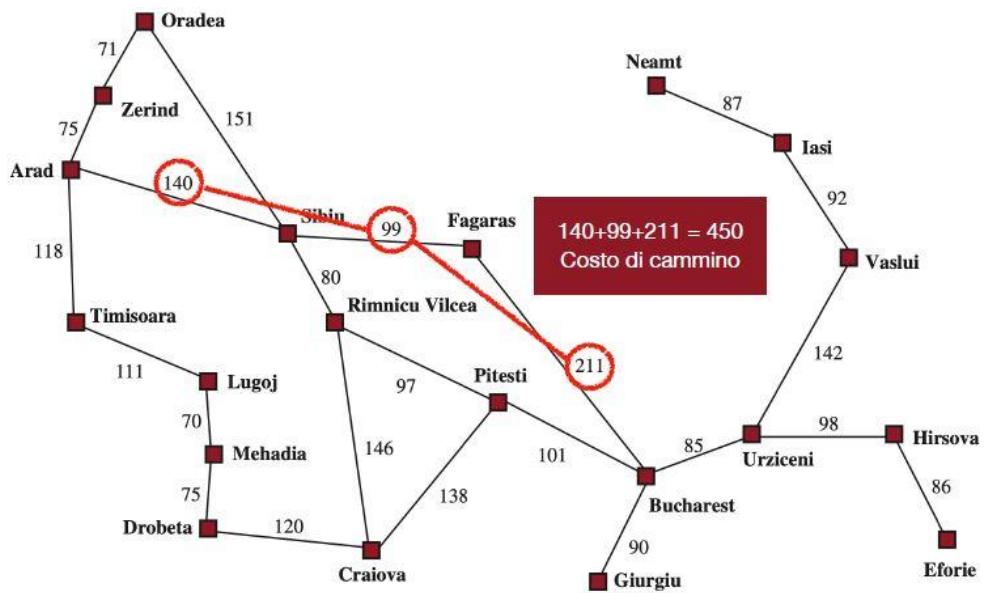
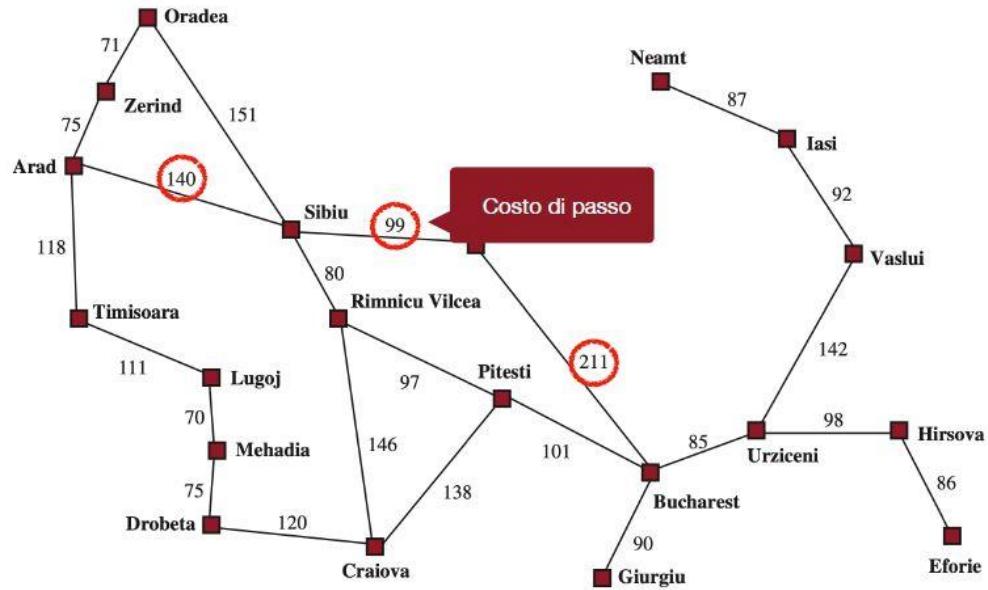
dell'azione a nello stato s . Utilizziamo anche il termine **successore** per indicare qualsiasi stato raggiungibile da uno stato mediante una singola azione. Ad esempio, potremmo avere che: $Risultato(in(Arad), Go(Zerind)) = In(Zerind)$;

- **Test obiettivo:** questo determina se un particolare stato è uno stato obiettivo. In alcuni casi, esiste un insieme esplicito di possibili stati obiettivo: in questo caso il test si limiterà a verificare se uno stato s appartiene all'insieme. Nell'esempio, il test obiettivo sarà dato dal singoletto $\{in(Bucarest)\}$;
- **Costo di cammino:** funzione che determina il costo numerico ad ogni cammino. Nell'esempio, il costo potrebbe essere rappresentato dai chilometri da percorrere per arrivare a Bucarest.

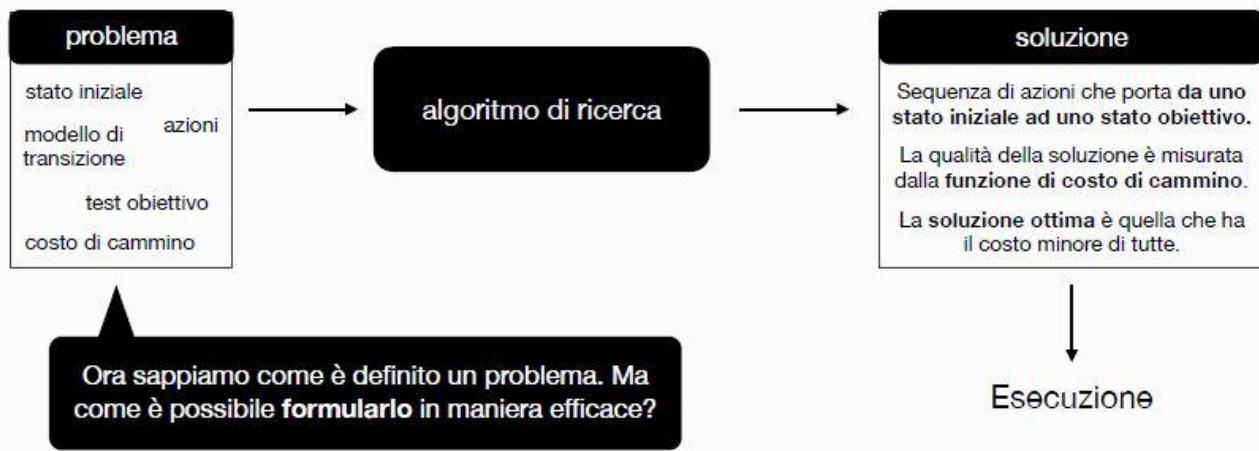
Stato iniziale, azioni e modello di transizione definiscono lo **stato degli spazi** del problema, l'insieme di tutti gli stati raggiungibili da quello iniziale mediante qualsiasi sequenza di azioni. Questo può essere rappresentato sotto forma di **grafo**, in cui i nodi rappresentano gli stati e gli archi (pesati) rappresentano l'azione che porta da un nodo ad un altro, etichettata con il relativo **costo di passo** $c(s, a, s')$, ovvero il costo dell'azione a che porta dallo stato s a quello s' .







Formulazione di problemi



Nell'esempio precedente, abbiamo proposto una formulazione del problema (andare da Arad a Bucarest). Sebbene sembri ragionevole, questa formulazione rappresenta una *descrizione matematica astratta*, ovvero un **modello**. Basto confrontare la descrizione distata scelta, *in(Arad)*, con un vero viaggio in macchina. In un contesto del genere, dovremmo passare necessariamente per una serie di altri paesi e città. Lo stato del mondo include varie altre particolarità che influenzano l'agente, come i compagni di viaggio, la presenza di poliziotti nelle vicinanze, le condizioni del tempo e della strada, ecc.

Queste considerazioni sono state omesse poiché sono irrilevanti al fine ultimo di raggiungere l'obiettivo Bucarest. Il processo di rimozione dei dettagli da una rappresentazione è detta **astrazione**.

Oltre ad astrarre le descrizioni di stato, dobbiamo astrarre anche le azioni – un'azione di guida è influenzata e può avere molti effetti (ad esempio, la benzina viene consumata). È possibile allora essere più precisi nella definizione di un livello di astrazione *appropriato*? 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*.

In altri termini, con questa formulazione del problema il viaggio a Bucarest sarebbe scomponibile in sotto-problemi:

1. Da Arad a Sibiu;
2. Da Sibiu a Faragas;
3. Da Faragas a Bucarest.

Un'astrazione si dice **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 prevede, quindi, che si rimuova quanto più dettaglio possibile mantenendo la validità e assicurandosi che le azioni astratte siano più facili da eseguire.

Formulazione di problemi tramite esempi - Il Problema delle 8 Regine

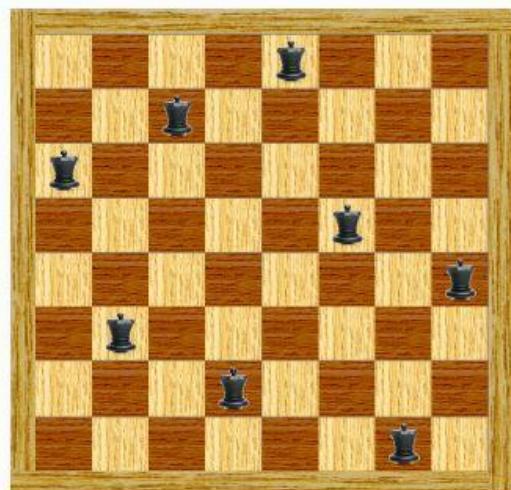
Una volta stabilite le basi di un agente risolutore di problemi così come di un algoritmo di ricerca, consideriamo alcuni casi tipici e/o più pratici di formulazione di problemi.

Scopo. Piazzare 8 regine su una scacchiera in modo tale che nessuna possa attaccarne un'altra. Negli scacchi, una regina attacca i pezzi che si trovano sulla sua stessa colonna, riga o diagonale.

Soluzione. Una delle possibili soluzioni è quella mostrata in figura.

Nell'ambito del corso, vedremo come risolvere il problema utilizzando due particolari tipologie di algoritmi di ricerca, ovvero gli *algoritmi genetici* e il *backtracking*. Per il momento, il nostro obiettivo è quello di definire una corretta formulazione del problema.

In particolare, possiamo parlare di due tipologie di formulazione, che prendono il nome di **formulazione incrementale** e **formulazione a stato completo**.



Formulazione di problemi tramite esempi - Il Problema delle 8 Regine

La formulazione incrementale utilizza operatori che estendono progressivamente la descrizione di stato, cominciando dallo stato vuoto.

Stati. Ogni piazzamento sulla scacchiera di un numero da 0 a 8 regine è uno stato.

Stato iniziale. Scacchiera vuota.

Azioni. Aggiungere una regina in una casella vuota.

Modello di transizione. Restituisce la scacchiera con una regina aggiunta nella casella specificata.

Test obiettivo. Sulla scacchiera ci sono 8 regine e nessuna è attaccata.

In questa formulazione abbiamo circa $1,8 \cdot 10^{14}$ possibili sequenze da investigare. Da notare che nessun vincolo è stato specificato nella formulazione.

Stati. Tutte le possibili configurazioni di regine, una per colonna a partire da sinistra, tali che nessuna regina ne attacchi un'altra.

Azioni. Aggiungere una regina in una qualsiasi casella della colonna vuota più a sinistra, in modo tale che essa non sia attaccata da nessun'altra regina.

Questa formulazione riduce lo spazio degli stati a 2.057, rendendo il problema più semplice da risolvere. Ma attenzione, se parliamo del problema generico delle N regine con N=100, la riduzione porterebbe a 10^{52} stati, il ché renderebbe il problema intrattabile.

Formulazione di problemi tramite esempi - Il Problema delle 8 Regine

La formulazione a stati completi prevede che le otto regine siano già sulla scacchiera, per poi spostarle in maniera iterativa.

Stati. Ogni piazzamento sulla scacchiera di un numero da 0 a 8 regine è uno stato.

Stato iniziale. Scacchiera con una regina piazzata per colonna.

Azioni. Sposta una regina nella colonna, se minacciata.

Modello di transizione. Restituisce la scacchiera con una regina la cui posizione è modificata.

Test obiettivo. Sulla scacchiera ci sono 8 regine e nessuna è attaccata.

In questa formulazione abbiamo circa $1,6 \cdot 10^7$ possibili sequenze da investigare, notevolmente di meno rispetto all'equivalente formulazione incrementale senza vincoli.

Formulazione di problemi tramite esempi - Il Problema del Commesso Viaggiatore

Il problema del commesso viaggiatore è un altro tipico esempio di utilizzo di algoritmi di ricerca. A differenza del caso precedente, parliamo di problemi di ottimizzazione *reali*.

Scopo. Un commesso deve consegnare dei pacchi e, pertanto, dovrà visitare ogni città esattamente una volta: l'obiettivo è trovare il percorso più breve. Il problema è spesso implementato in molte applicazioni logistiche.

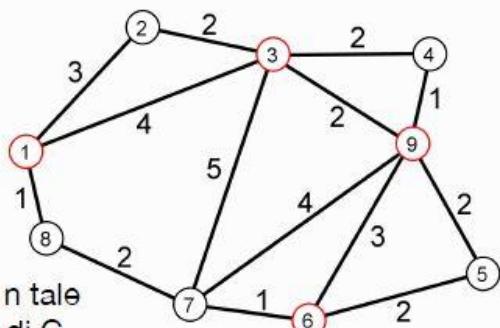
Un ciclo che passa su tutti i nodi del grafo una e una sola volta è detto *ciclo Hamiltoniano*. Determinare se un grafo contiene un ciclo di questo tipo rappresenta un problema NP-Completo (non è risolvibile in tempo polinomiale).

Grafo $G(N, A)$ completo - ovvero un grafo che ha un arco tra ogni coppia di vertici.

c_{ij} = costi sugli archi.

La risposta al problema è ‘Si’ se esiste un ciclo hamiltoniano nel grafo G , ‘No’ altrimenti.

Formalmente, una permutazione π dei nodi $1, 2, \dots, n$ tale che per ogni $i = 1, \dots, n-1$, $(\pi(i), \pi(i+1))$ è un arco di G .



Formulazione di problemi tramite esempi - Il Problema del Commesso Viaggiatore

Il problema del commesso viaggiatore è un altro tipico esempio di utilizzo di algoritmi di ricerca. A differenza del caso precedente, parliamo di problemi di ottimizzazione *reali*.

Stati. Posizione corrente + insieme di città già visitate.

Stato iniziale. Il commesso si trova nella prima città.

Azioni. Spostamenti tra i nodi (città).

Modello di transizione. Restituisce un grafo modificato in cui il commesso si è spostato da una città ad un'altra.

Test obiettivo. Il commesso arriva all'ultima città passando per tutte le città.

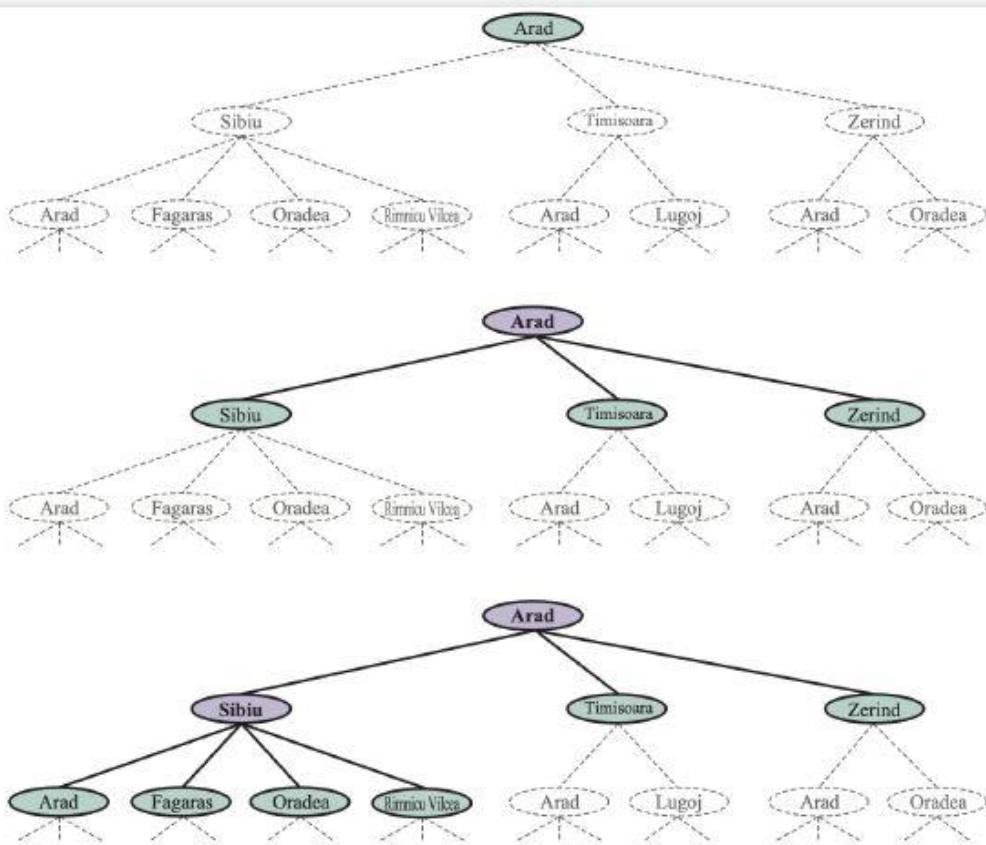
Cercare soluzioni

Abbiamo imparato come formulare i problemi, ora dobbiamo risolverli. Una **soluzione** è una **sequenza di azioni**, quindi gli algoritmi di ricerca lavorano considerando varie possibili sequenze di azioni. Tali sequenze di azioni formano un **albero di ricerca** con uno stato iniziale alla radice, i rami sono aioni e i nodi corrispondono a stati nello “spazio degli stati” del problema.

Dato lo stato iniziale, il primo passo sarà verificare che l'obiettivo non sia stato raggiunto; dopodiché, si considerano le varie azioni. Lo facciamo espandendo lo stato attuale: vale a dire, applicare ogni azione legale allo stato attuale, generando così un nuovo insieme di stati.

Nell'esempio del viaggio a Bucarest, la radice sarà Arad e l'espansione dello stato corrente non sarà altro che aggiungere alla radice tre nodi che rappresentano le città che si possono raggiungere; in questo caso Sibiu, Timisoara e Zerind. Poi bisogna scegliere quale di queste tre possibilità considerare ulteriormente.

Questa è l'essenza della ricerca: approfondire un'opzione e mettere per il momento a parte le altre, pronti a riprendere nel caso la prima non porti ad una soluzione.



Uno stato può essere ripetuto più volte: in questo caso parliamo di **cammino ciclico**. (vedi in basso a destra nella figura sopra: da Sibiu c'è un nodo che lo riporta ad Arad). In alcuni casi, i cicli possono causare il fallimento di alcuni algoritmi, poiché non vi è limite al numero di volte in cui è possibile percorrere un ciclo. Tuttavia, i cammini sono additivi e i costi dei passi non negativi: un cammino ciclico per qualsiasi stato non è perciò mai migliore di quello che si ottiene dallo stesso cammino rimuovendo un ciclo.

I nodi privi di figli in un determinato momento della ricerca sono detti nodi **foglia** (vedi Timisoara e Zerind in basso a destra nella figura sopra). L'insieme di tutti i nodi foglia che possono essere espansi in un dato punto è detto **frontiera**.

Il processo di espandere i nodi su una frontiera

```
function RICERCA-ALBERO(problema) 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 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
```

Se volessimo essere più precisi, allora l'algoritmo dovrebbe prendere in considerazione una **strategia** in base alla quale poter espandere una frontiera:

```
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

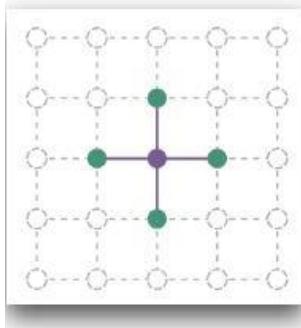
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        espandi il nodo scelto, aggiungendo i nodi risultati alla frontiera
```

La **strategia** definisce la politica che l'algoritmo utilizzerà per decidere come procedere la ricerca.

Cammini ciclici e ridondanti

I cammini ciclici, ovvero quelli in cui uno stato è ripetuto nell'albero di ricerca, sono un caso particolare di **cammino ridondante**: un cammino in cui esistono due o più modi per passare da uno stato all'altro.

I cammini ridondanti possono talvolta causare il fallimento degli algoritmi di ricerca. In alcuni casi, tuttavia, non è possibile evitare i cammini ridondanti; basti pensare, ad esempio, ai problemi in cui le azioni sono *reversibili*.



Ricerca di un itinerario su griglia orizzontale: Ogni stato ha 4 successori, perciò l'albero di ricerca di profondità d che include stati ripetuti avrà 4^d foglie, nonostante ci siano "solo" $2 \cdot d^2$ stati distinti a d passi da qualsiasi stato dato.

Per $d = 20$, ciò significa che ci sono circa mille miliardi di nodi ma soltanto 800 stati distinti.

In altri termini, seguendo cammini ridondanti si può trasformare un problema trattabile in uno intrattabile.

Per evitare di addentrarsi in cammini ridondanti, è perciò necessario ricordare dove si è passati. Possiamo quindi migliorare l'algoritmo di ricerca introducendo una struttura dati denominata **insieme esplorato** (o lista chiusa).

```
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 direttamente come un grafo dello spazio degli stati.

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

Strutture dati per algoritmi di ricerca

Gli algoritmi di ricerca richiedono una struttura dati per tenere traccia dell'albero di ricerca costruito. Per ogni nodo n dell'albero abbiamo una struttura contenente i seguenti quattro componenti:

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

Dati i componenti di nodo padre, è poi facile calcolare i 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)
```

Memorizzati i singoli nodi, è necessaria una struttura dove conservare tutti i nodi che saranno generati nell'albero di ricerca. La frontiera deve essere memorizzata in modo da consentire all'algoritmo di ricerca di scegliere facilmente il successivo nodo da espandere in base alla propria strategia. In tal senso, la struttura dati più adatta è una **coda** (LIFO, FIFO, Coda con priorità).

Valutare un algoritmo di ricerca

Una strategia di ricerca è definita *scegliendo l'ordine in cui i nodi sono espansi*. Tali strategie vengono valutate in base a quattro indicatori:

- **Completezza:** l'algoritmo *garantisce* di trovare una soluzione se questa esistesse?
- **Ottimalità:** l'algoritmo garantisce di trovare la *soluzione ottima*?
- **Complessità temporale:** quanto tempo *impiega* l'algoritmo per trovare una soluzione?
- **Complessità spaziale:** di 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, poiché la struttura dati viene esplicitamente passata come input di un algoritmo, in IA il grafo è spesso *implicitamente* rappresentato dallo stato iniziale, dalle azioni e dal modello di transizione: per questa ragione, è spesso *indefinito*.

Ne consegue che la complessità viene espressa in termini diversi: si parla di *fattore di ramificazione b*; *profondità della soluzione a costo minimo d*; *massima profondità dello spazio degli stati m*.

Algoritmi di ricerca non informata

Gli algoritmi di ricerca non informata fanno riferimento alle strategie di ricerca che **non dispongono di informazioni aggiuntive sugli stati** oltre a quella fornita nella definizione del problema: tutto ciò che possono fare è generare successori e distinguere gli stati obiettivo dagli altri.

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

Data la loro natura, 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 e euristica, i quali fanno uso di informazioni riguardo la distanza stimata dalla soluzione.

La ricerca non informata è un metodo di ricerca (spesso) non efficiente poiché, non utilizzando alcuna informazione ulteriore del problema da risolvere, implica l'esplorazione di tutti gli stati possibili.

Ricerca in ampiezza

La ricerca in ampiezza è una semplice strategia in cui i nodi dell'albero sono analizzati secondo un ordine di vicinanza al nodo radice. Sono espansi dapprima i nodi più vicini alla radice e successivamente tutti gli altri nodi successori.

Da un punto di vista formale **ogni nodo successore viene aggiunto alla coda FIFO** dei nodi ancora da analizzare (sarebbero i nodi della frontiera) **non appena viene espanso**.

La ricerca in ampiezza è **sempre in grado di trovare una soluzione**, se esiste, **realizzabile con il cammino più breve**: tuttavia, cammino più breve non è necessariamente quello con il costo minore.

```
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) 1

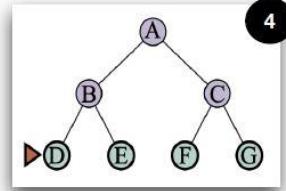
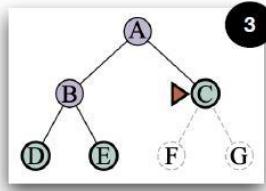
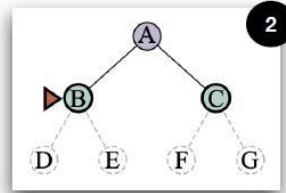
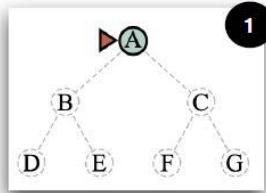
    frontiera <- una coda FIFO con nodo come unico elemento
    esplorati <- un insieme vuoto 2

    loop do
        if la frontiera è vuota then return fallimento 3
        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) 5
```

1. All'inizio, non facciamo altro che assegnare la radice del grafo alla variabile *nodo* e controllare se la soluzione è la radice.
2. Se la soluzione non è la radice, allora inizializziamo la frontiera (con la radice, che è l'unico nodo esplorato finora) e la lista dei nodi esplorati.
3. Se, durante la ricerca, la frontiera dovesse essere vuota e la soluzione non dovesse essere trovata, allora non resta che restituire un fallimento.
4. La funzione POP di una coda estrae il primo elemento della frontiera, di fatto procedendo alla sua espansione; il nuovo nodo verrà aggiunto a quelli esplorati.
5. 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.

Strategie di ricerca non informata - Ricerca in ampiezza, un esempio pratico



Ad ogni iterazione, viene espanso il nodo indicato. Si può pensare alla ricerca in ampiezza come un pendolo che, di volta in volta, visita i nodi in maniera orizzontale.

Performance ricerca in ampiezza

Denotiamo con b il fattore di diramazione (ovvero, il numero massimo di successori); e con d la profondità della soluzione minima.

Completezza: è facilmente intuibile che l'algoritmo è completo. Se il nodo obiettivo più vicino alla radice si trova alla profondità d , la ricerca lo troverà dopo aver espanso tutti i nodi che lo precedono.

Ottimalità: se il costo di cammino è una funzione monotona non decrescente della profondità del nodo, allora possiamo dire che la ricerca è ottima. Più in generale, però, l'algoritmo non lo è – restituisce la soluzione più vicina, indipendentemente dal costo.

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 nodi al secondo livello, ecc. Nel caso pessimo la soluzione è l'ultima ad essere analizzata e si trova ad una profondità d . Quindi $O(b^d)$, la complessità temporale è esponenziale.

Complessità spaziale: l'algoritmo memorizza ogni nodo espanso nell'insieme *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 $O(b^d)$, ovvero è dominata dalla dimensione della frontiera.

Ricerca a costo uniforme

La ricerca a costo uniforme è una strategia di ricerca non informata in cui l'algoritmo espande il nodo sulla frontiera con il costo di cammino più basso dal nodo radice. Questo è possibile semplicemente memorizzando la frontiera come **coda a priorità ordinata secondo il costo di cammino $g(n)$** .

Questa tipologia di ricerca si presta ad essere utilizzata in caso di costi di passo non uguali nell'albero di ricerca. Questa strategia permette di trovare la soluzione più economica senza dover necessariamente analizzare l'intero albero.

Ci sono due sostanziali 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) 1

    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) 2
    aggiungi nodo.STATO a esplorati

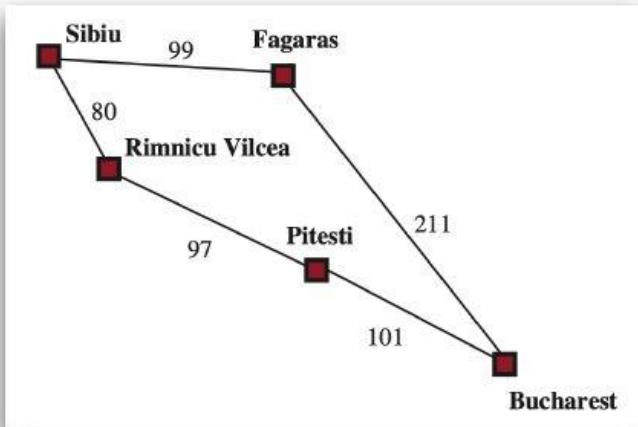
    for each azione in problema.AZIONI(nodo.STATO) do
        figlio <- NODO-FIGLIO(problema, stato, azione)
        if figlio.STATO non è in esplorati e figlio non è in frontiera then
            if problema.TEST-OBIETTIVO(figlio.STATO) then return SOLUZIONE(figlio)
            frontiera <- INSERISCI(figlio, frontiera)
    else if figlio.STATO è in frontiera con COSTO-CAMMINO più alto then
        sostituisci quel nodo frontiera con figlio 3
```

(in rosso sono le linee di codice appartenenti all'algoritmo di ricerca per ampiezza che sono rimosse dall'algoritmo a costo uniforme; in verde, invece, le linee che implementano le differenze con l'algoritmo di ricerca per ampiezza).

1. Se il test fosse fatto a questo punto, rischieremmo di ritornare una soluzione sub-ottima, poiché il primo nodo generato potrebbe trovarsi su un cammino con costo maggiore.
2. Ricordiamo che la funzione POP di una coda a priorità restituisce il nodo di costo minimo in frontiera: pertanto, se il test ha successo, allora abbiamo trovato la soluzione migliore.
3. Per la stessa ragione, non effettuiamo il test qui perché il figlio potrebbe trovarsi su un cammino sub-ottimo.

4. L'ultimo *if* consente di modificare l'ordinamento della lista e di conseguenza l'esplorazione dei nodi – in questo modo, si potrà disporre sempre dei nodi in ordine crescente di costo.

Strategie di ricerca non informata - Ricerca a costo uniforme, un esempio pratico



Obiettivo: Raggiungere Bucarest da Sibiu.

I successori di Sibiu sono Vilcea e Fagaras, che distano 80 e 99 km, rispettivamente, da Sibiu.

Viene espanso il nodo con costo minore, Vilcea. A questo punto, viene generato il nodo figlio, Pitesti.

Il costo di cammino sarà quindi pari a $80 + 97 = 177$.

Ora il nodo con costo minore sarà Fagaras, con 99 (< 177). La ricerca procederà quindi con la sua espansione verso Bucarest: il costo di cammino sarà di $99 + 211 = 310$.

Sebbene sia stato raggiunto l'obiettivo, la ricerca a costo uniforme continua poiché esiste un nodo con costo minore di 310. Il nodo Pitesti verrà espanso, il che porterà al raggiungimento di Bucarest con costo $177 + 101 = 288$.

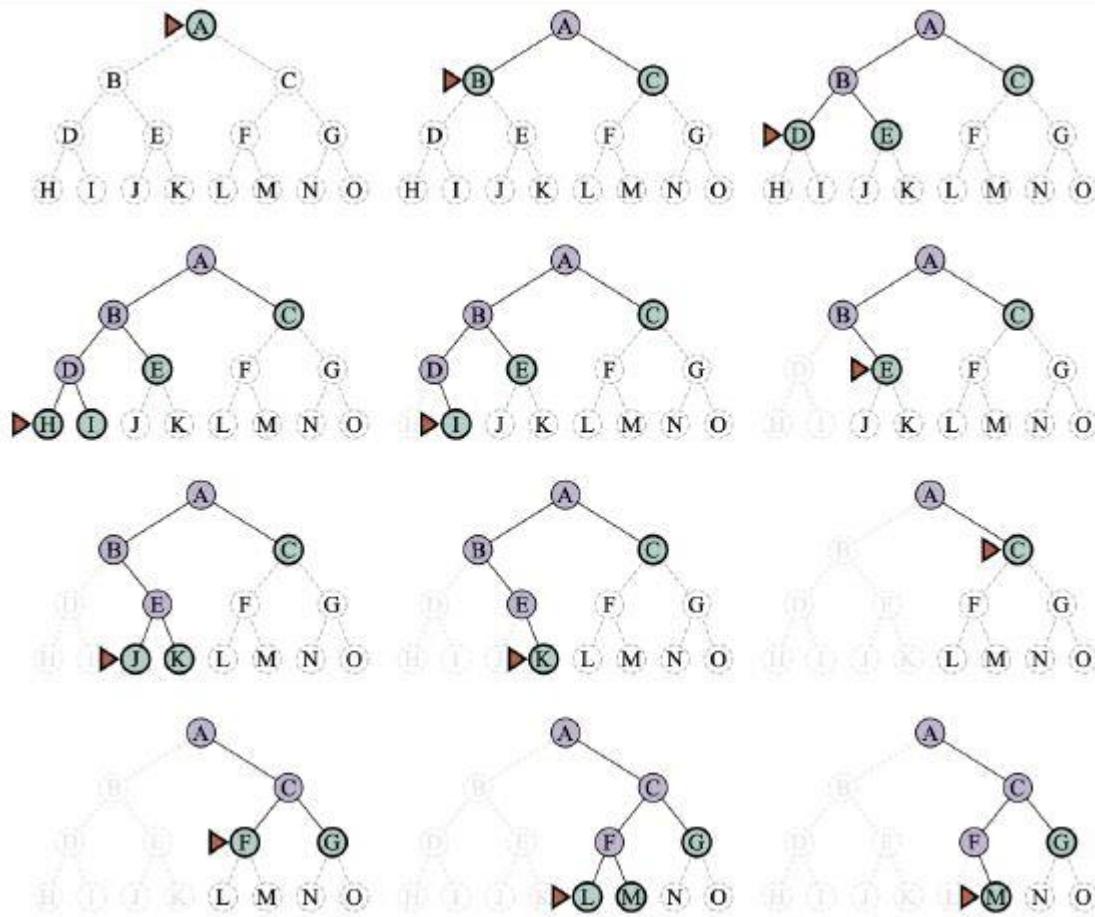
L'algoritmo verificherà che questo cammino è migliore del precedente ($288 < 310$) e restituirà, quindi, la soluzione finale.

Da un punto di vista di performance, la ricerca a costo uniforme è un algoritmo completo e, per natura, è anche ottimo. Non è semplice, però, definire esattamente la complessità temporale e spaziale di questo algoritmo perché l'algoritmo di base sui costi di passo e non sulla profondità dell'albero. In generale possiamo dire che la ricerca a costo uniforme è più efficace della ricerca in ampiezza ma potrebbe essere meno efficiente.

Ricerca in profondità

Questo tipo di ricerca espande i nodi analizzando un ramo dell'albero alla volta. Da un punto di vista pratico, la ricerca in profondità raggiunge immediatamente il livello più profondo dell'albero, dove i nodi non hanno successori. Dopo ciò la ricerca "torna indietro" per considerare il nodo più profondo che ha successori ma che non è stato ancora espanso.

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



Il problema di questo algoritmo è che potrebbe presentare dei **cammini ridondanti**, ovvero cammini in cui esistono due o più modi per passare da uno stato all'altro.

A livello di performance non è per niente efficiente infatti **non è completo** perché nel caso in cui l'albero abbia profondità infinita o presente dei cicli, l'algoritmo non terminerà. Inoltre, **non è nemmeno ottimo** in questo potrebbe ritornare un nodo obiettivo molto lontano dalla radice.

Ha un enorme vantaggio in termini di complessità spaziale in quanto è sufficiente memorizzare il numero dei rami da analizzare a partire dal nodo radice e i singoli nodi del cammino all'interno del ramo. Una volta analizzata la ramificazione di un ramo, la memoria può essere liberata per fare spazio all'analisi del ramo successivo.

La complessità temporale, invece, rimane sempre esponenziale.

Per migliorare l'algoritmo, almeno dal punto di vista della completezza, possiamo imporre un limite massimo alla profondità dei cammini in modo da evitare che l'algoritmo non termini. Questo tipo di algoritmo si chiama **ricerca in profondità limitata**.

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: o per via di un fallimento (ovvero, nessuna soluzione) o tramite il valore speciale *taglio* (ovvero, nessuna soluzione entro il limite stabilito).

Il problema di questa strategia consiste nel selezionare una soglia tale da poter garantire il successo. Talvolta questa soglia deriva dall'esperienza del designer o da dati empirici; altre volte, invece, non è possibile definirla.

Da un punto di vista di performance abbiamo visto che l'algoritmo **non è completo** nei casi in cui la soluzione si trova ad una profondità maggiore del limite stabilito. Questa strategia **non può essere ottima** nei casi in cui il limite stabilito sia minore della profondità massima (quindi sempre). La complessità temporale è limitata dalla dimensione dello spazio degli stati; nel caso pessimo è comunque esponenziale. La complessità spaziale invece è buona per gli stessi motivi della ricerca in profondità.

```
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
```

Ricerca ad approfondimento iterativo

L'algoritmo di ricerca ad approfondimento iterativo integra in sé sia la ricerca in ampiezza che la ricerca in profondità.

Inizialmente l'algoritmo effettua le operazioni di ricerca nei nodi vicini alla radice, aumentando progressivamente la profondità di scansione nei cicli di ricerca successivi. Nel momento in cui l'algoritmo trova una soluzione interrompe il ciclo di ricerca senza dover scandagliare ulteriormente l'albero alla ricerca di altre soluzioni.

Questo tipo di ricerca è il metodo da preferire quando lo spazio di ricerca è grande e la profondità della soluzione non è nota.

Da un punto di vista delle performance possiamo dire che, come la ricerca in ampiezza, è un algoritmo **completo**. Per quanto riguarda l'ottimalità, l'algoritmo **non lo è**. Nel caso pessimo la complessità temporale è esponenziale. Invece la complessità spaziale è uguale a quella dell'algoritmo di ricerca in profondità.

Ricerca bidirezionale

L'idea alla base di questa strategia di ricerca è quella di effettuare due ricerche in parallelo: una prima in avanti dallo stato iniziale e l'altra all'indietro dallo stato obiettivo.

Si è pensato di procedere in questo modo perché sia la complessità temporale che spaziale sono migliori.

$$O(b^{d/2}) + O(b^{d/2}) < O(b^d)$$

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

Non è però così semplice da implementare.

Con questa strategia si definisce il concetto di **predecessore**: i predecessori di uno stato x sono tutti gli stati che hanno x come successore. Definire un predecessore però non è sempre così immediato.

Algoritmi di ricerca informata

La ricerca informata è una strategia di ricerca per trovare una o più soluzioni ad un problema utilizzando una **funzione di conoscenza**, la quale determinerà l'ordine di ricerca dell'algoritmo al fine di ridurre il tempo e il costo della ricerca.

Questa funzione di conoscenza (o euristica) stima la probabilità di trovare la soluzione nei vari nodi da selezionare e, sulla base di queste stime, modifica l'ordine dei nodi nella coda di ricerca anticipando l'analisi su quelli a maggiore probabilità di successo.

Quindi non elimina il processo di ricerca ma ne riduce fortemente i tempi di esecuzione. Pur non essendo una ricerca completa, la ricerca informata consente di raggiungere comunque soluzioni accettabili con maggiore efficienza rispetto ad una ricerca non informata.

Esistono diverse tipologie di ricerca informata: **Best-First**, **A***, **Beam Search**, **IDA***, **Best-First ricorsiva**, **SMA***, **ricerca euristica**.

Ricerca Best-First

La ricerca Best-First è una strategia in cui ad ogni passo viene espanso sempre il nodo migliore, ovvero **quello più vicino al nodo obiettivo**, sulla base del fatto che è probabile che questo porti rapidamente ad una soluzione.

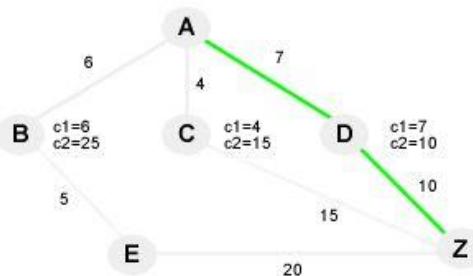
Per valutare la qualità dei nodi, l'algoritmo si avvale di una funzione euristica, la quale, per esempio, potrebbe considerare la distanza tra i nodi di partenza e il nodo obiettivo. Ovviamente, però, questa euristica dovrebbe conoscere a priori tutte le distanze tra i nodi. (questa è un'informazione che rende, appunto, l'algoritmo di ricerca *informato*).

In termini di performance possiamo dire che l'algoritmo **non è completo** perché potrebbe tranquillamente incappare in cicli infiniti. **Non è nemmeno ottimo** in quanto, ad ogni passo, cerca sempre di arrivare più vicino all'obiettivo senza considerare cammini alternativi di costo inferiore. Nel caso pessimo, l'algoritmo ha complessità temporale di tipo esponenziale, tuttavia con una buona euristica, è possibile ridurre drasticamente tale complessità. Invece, in termini di complessità spaziale, è esponenziale anch'essa perché conserva tutti i nodi in memoria.

Ricerca A*

La ricerca A* è una strategia di ricerca informata in cui la scelta del cammino si basa sia sul costo effettivo $g(n)$, ovvero il costo per raggiungere il nodo intermedio n dal nodo di origine, e sia sulla stima del costo di $h(n)$, ovvero il costo del nodo intermedio n al nodo obiettivo. La funzione di ricerca è quindi composta da due funzioni: $F(n) = g(n) + h(n)$.

La ricerca A* è una variante della ricerca a costo uniforme, a differenza che la ricerca A* analizza il costo del cammino $g(n)$ soltanto fino ad un livello di profondità n (nodo intermedio) e da qui stima il cammino $h(n)$ per raggiungere il nodo obiettivo. In tal modo la ricerca A* permette di analizzare dapprima i cammini presumibilmente migliori, ossia quelli in grado di minimizzare $F(n)$.



Come si può facilmente osservare, a partire dal nodo iniziale A è possibile intraprendere tre cammini per raggiungere il nodo finale Z. Se l'algoritmo prendesse in considerazione soltanto il costo effettivo C_1 dovrebbe analizzare dapprima il nodo C in quanto ha un costo C_1 inferiore (4) rispetto ai nodi B (6) e D (7). Ciò porterebbe ad arrivare al nodo finale Z mediante il cammino ACZ con un [costo di cammino](#) pari a 19 (4+15). Pur essendo una soluzione efficace, in quanto l'obiettivo viene raggiunto, non è anche una soluzione efficiente dal punto di vista dei costi. L'**algoritmo di ricerca A-star** riduce il rischio di selezionare un cammino inefficiente poiché non si limita a prendere in considerazione soltanto il costo C_1 , ma anche una stima (ricerca informata) del costo C_2 . Seguendo tale logica il nodo intermedio migliore risulta essere il nodo D, il quale pur essendo il più lontano dal nodo iniziale ($C_1=7$) consente di minimizzare la somma dei costi C_1+C_2 (7+10) rispetto ai cammini alternativi. Il costo del cammino individuato dall'algoritmo A-star ADZ è pari a 17 ed è il cammino più efficiente per raggiungere il nodo Z a partire dal nodo A.

L'ottimalità della strategia di ricerca A* è determinata dall'euristica utilizzata per stimare il costo di $h(n)$, la quale deve essere sempre un'euristica ammissibile e consistente, cioè che non sbagli mai per eccesso la stima del costo di cammino. Inoltre l'algoritmo è completo se esiste un numero finito di nodi da analizzare di costo minore o uguale al costo della soluzione ottima.

In termini di complessità temporale, l'algoritmo è ottimamente efficiente, in quanto nessun altro come lui espande meno nodi (senza rinunciare all'ottimalità). Non riusciamo però a migliorare la complessità spaziale in quanto resta sempre esponenziale, però si può lavorare su questo aspetto per tentare di ottimizzare l'algoritmo.

Migliorare l'occupazione in memoria, la 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*: definita in base ad una euristica.

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.

Da un punto di vista pratico, ad ogni livello dell'albero di ricerca l'algoritmo genera tutti i successori degli stati a quel livello, ordinandoli in maniera decrescente in base all'euristica scelta. Conserverà poi solo i primi k di ogni livello che saranno poi espansi.

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

Migliorare l'occupazione in memoria, la Iterative Deepening A* (IDA*)

Un altro approccio per migliorare la complessità spaziale dell'algoritmo di ricerca A* è quello di adottare l'idea dell'approfondimento iterativo ad esso.

La differenza principale tra IDA* e l'approfondimento iterativo standard sta nel valore del taglio, che non è più basato sulla profondità ma sul costo f , ovvero $g+h$.

Ad ogni iterazione, il nuovo valore di taglio è l' f -costo minimo tra quelli di tutti i nodi che hanno superato il valore di taglio nell'iterazione precedente.

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

Il vero vantaggio si ottiene in termini di occupazione di memoria, poiché l'algoritmo mantiene le proprietà dell'algoritmo di ricerca ad approfondimento iterativo standard: la ricerca dovrà 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 $O(bd)$.

Migliorare l'occupazione di memoria, la ricerca Best-First ricorsiva

Questa rappresenta una variante della ricerca Best-First classica, usando tuttavia 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 della funzione euristica).

In primo luogo, la ricerca Best-First ricorsiva è ottimale se la funzione euristica è ammissibile. L'algoritmo vanta un livello di complessità spaziale molto basso poiché utilizza uno spazio lineare. La scarsa memoria su cui si basa l'algoritmo non elimina il rischio di analizzare più volte sempre gli stessi percorsi o di incappare in cammini ridondanti. Ciò equivale a dire che la complessità temporale dell'algoritmo può diventare anche molto elevata.

Il problema della ricerca Best-First ricorsiva, così come l'IDA*, consiste nel fatto che utilizzano troppa poca memoria. Tra un'iterazione ad un'altra, IDA*, ricorda solo il limite corrente all' f -

costo, mentre la ricerca Best-First ricorsiva usa solo uno spazio lineare. Anche se fosse disponibile più memoria, i due algoritmi non riuscirebbero ad usarla. Entrambi gli algoritmi dimenticano la maggior parte di ciò che fanno, **rischiando di espandere più e più volte gli stessi stati già visitati in precedenza.**

Migliore occupazione della memoria, la ricerca Simplified Memory Bounded A* (SMA*)

L'idea alla base è quella di utilizzare meglio la memoria disponibile. Molto semplicemente, 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, però, 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 sottoalbero: SMA* rigenera un sottoalbero dimenticato solo quando tutti gli altri cammini promettono di comportarsi peggio di quello.

SMA* è completo se la soluzione è raggiungibile, ovvero se la profondità del nodo obiettivo più vicino alla radice è inferiore alla dimensione della memoria espressa in nodi. Inoltre, è anche ottimale se c'è una soluzione ottima raggiungibile.

Il problema di questo algoritmo consiste nel fatto che, per problemi difficili, la ricerca sarà spesso 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: la complessità temporale potrebbe esplodere, rendendo il problema non risolvibile nella pratica.

Funzioni euristiche

Molti problemi dell'IA 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. Migliorando anche di poco un'euristica si riesce ad esplorare uno spazio degli stati molto più vasto.

Per la risoluzione di problemi, spesso si dovranno inventare delle euristiche: *rilassamento dei vincoli, massimizzazione di euristiche, apprendere dall'esperienza, combinare euristiche.*

Algoritmi di ricerca locale

Gli algoritmi precedenti sono progettati per esplorare sistematicamente lo spazio degli stati e 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 problemi reali il cammino che porta alla soluzione è *irrilevante* proprio perché lo stato obiettivo è esso stesso la soluzione al problema indipendentemente da come ci si è arrivati.

Quindi possiamo considerare algoritmi diversi che ignorano il cammino che porta alla soluzione, i quali mantengono in memoria solo lo stato corrente e tentano di migliorarlo: questi algoritmi sono detti **algoritmi di miglioramento iterativo**.

Di questi algoritmi fanno parte gli algoritmi di ricerca locale.

Algoritmi di ricerca “tradizionali”

Considerano interi cammini dallo stato iniziale a quello obiettivo

Hanno una complessità temporale e spaziale generalmente esponenziale

Non possono essere applicati in problemi con spazio degli stati grandi/infiniti

Algoritmi di ricerca locale

Considerano lo stato corrente con l'obiettivo di migliorarlo

Usano poca memoria, molto spesso avendo una complessità costante

Possono trovare soluzioni ragionevoli (sub-ottimali) a problemi complessi

Funzione obiettivo

Gli algoritmi di ricerca locale non hanno un testo 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, come ad esempio la disposizione dei tavoli di un ristorante: in questi casi, infatti, non esiste un vero e proprio test obiettivo.

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*.

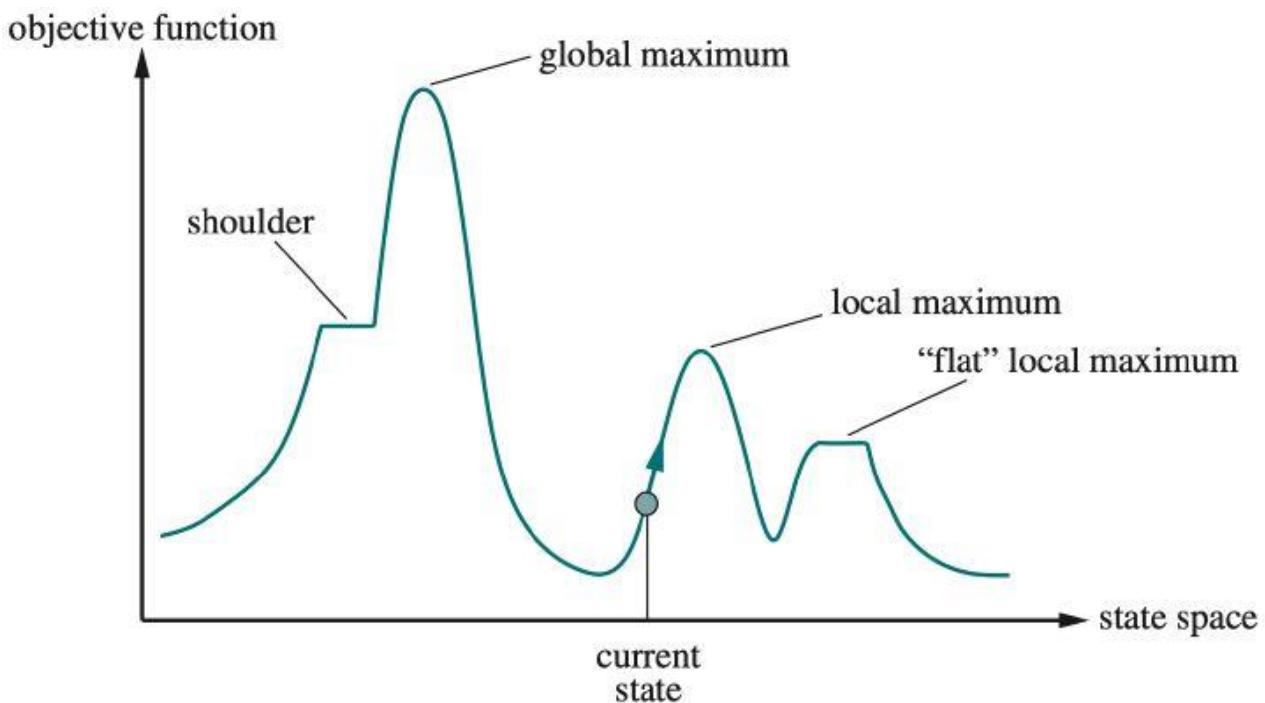
Tipicamente, la funzione obiettivo è definita attraverso le possibili mosse che l'algoritmo può effettuare. È da notare che la ricerca locale si basa sull'esplorazione iterativa delle "soluzioni vicine" che possono migliorare quella corrente mediante modifiche locali.

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

È da 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**. In altri termini, la soluzione può essere migliore possibile date le circostanze.

Il panorama dello spazio degli stati

Per poter meglio comprendere il funzionamento degli algoritmi di ricerca locale, è utile visualizzare lo spazio degli stati in maniera diversa rispetto ad un albero.



L'asse delle x rappresenta i vari stati raggiungibili nel processo di ricerca.

L'asse delle y indica il valore della funzione obiettivo per un determinato stato.

Lo stato corrente indica la posizione in cui si trova la ricerca rispetto al panorama degli stati.

Un punto piatto dello stato degli spazi indica una regione dello spazio dove gli stati vicini avranno lo stesso valore (anche detto *plateau*).

Una spalla è un plateau che presenta uno spigolo in salita (o in discesa).

Un massimo locale è una soluzione sub-ottima ad un problema di ricerca.

L'ottimo globale è invece la soluzione al problema migliore in assoluto.

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.

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

Algoritmo Hill-Climbing

L'algoritmo di 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.

Questo tipo di algoritmo segue una strategia simile agli algoritmi di ricerca Best-First Greedy: si sceglie un nodo iniziale, in base ad una funzione euristica oppure a caso, e si procede con l'elaborazione dei nodi vicini. Quando un nodo vicino è migliore del nodo di riferimento, quest'ultimo viene sostituito con il nuovo nodo. Il ciclo di elaborazione termina quando viene raggiunto il nodo con il valore più alto ("picco") ossia quando nessun nodo vicino ha valore superiore a quello di riferimento.

Tuttavia, in un algoritmo Hill Climbing la probabilità di fallire la ricerca è molto elevata. L'algoritmo presenta i seguenti vantaggi e svantaggi:

- **Minore complessità.** L'algoritmo Hill Climbing utilizza poca quantità di memoria. Non dovendo analizzare tutti i nodi e cammini possibili è caratterizzato da minori tempi di elaborazione. Quindi è un algoritmo molto rapido e veloce.
- **Massimo locale.** Uno dei principali handicap di questo algoritmo è il rischio di incappare in un massimo locale. Quando l'algoritmo trova un valore di massimo locale, si blocca; tuttavia, ciò non esclude la presenza lontana di massimi globali con valore superiore. L'algoritmo si muove velocemente verso il nodo migliore utilizzando una strategia troppo miope.

Per ridurre questo rischio è possibile ricorrere a diverse tecniche. In primo luogo si può cercare di **migliorare la funzione euristica di individuazione dei nodi iniziali**. In secondo luogo è consigliabile eseguire più volte l'algoritmo seguendo la logica dell'algoritmo **Hill Climbing iterativo**: elaborando più volte l'algoritmo da nodi iniziali differenti, tenendo in memoria il migliore risultato ottenuto. Facendo ciò è possibile migliorare l'efficacia della ricerca senza penalizzare eccessivamente la velocità e il tempo di elaborazione dell'algoritmo Hill Climbing.

Un'altra tecnica computazionale per ridurre il rischio del massimo locale sub-ottimale è l'algoritmo **Hill Climbing stocastico** in cui la scelta del nodo successore è determinata casualmente da una funzione stocastica e non è sempre quella migliore possibile.

- **Massimo locale piatto.** Un altro rischio dell'algoritmo Hill Climbing è quello di imbattersi in un massimo locale piatto: quando tutti i nodi vicini hanno un valore uguale al nodo corrente l'algoritmo si blocca.

Per ridurre il rischio di imbattersi in un massimo locale piatto è, invece, possibile ammettere una percentuale di **spostamenti laterali** tra nodi vicini a parità di valore per esplorare lo spazio di ricerca locale quando non sono presenti nelle vicinanze altri nodi con valori superiori a quello corrente. È sconsigliabile però consentire sempre gli spostamenti laterali in quanto potrebbe generare un ciclo infinito nell'elaborazione dell'algoritmo.

Algoritmo Simulated Annealing

Un algoritmo che non scende mai a “valle” verso stati con valore inferiore sarà sicuramente incompleto, poiché potrebbe restare bloccato in un massimo/minimo locale.

Di contro, un’esplorazione completamente casuale è completa ma estremamente inefficiente – dovrà continuamente navigare lo spazio di ricerca senza una strategia.

Sembra perciò sensato combinare una ricerca Hill Climbing con un’esplorazione casuale, per migliorare sia efficienza che completezza. Questo è quello che si propone di fare il **Simulated Annealing**.

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, sceglie i k successori migliori dalla lista e ricomincia.

La ricerca è “informata”, nel senso che l’informazione riguardo i successori analizzati viene passata dall’uno all’altro thread di ricerca paralleli.

Questo aumenta le chance di navigare il panorama degli stati verso stati più “promettenti”, possibilmente aumentando le chance di trovare un ottimo globale.

Da un altro punto di vista, la Local Beam potrebbe soffrire di una carenza di diversificazione tra i k stati, concentrando quindi la ricerca troppo rapidamente in una piccola regione dello spazio.

Un’alternativa è rappresentata dalla ricerca **Local Beam stocastica**: invece di scegliere i k successori migliori, si scelgono k successori a caso. Tuttavia, ai migliori di questi si assegna una probabilità maggiore di scelta.

Algoritmi Genetici

Per *Algoritmi Genetici* (GA) si intende una procedura di alto livello (meta-euristica) ispirata alla genetica per definire un algoritmo di ricerca. (Quindi è un approccio alla creazione di algoritmi)

I GA hanno alcune proprietà:

- **I GA non sono problem-specific.** Non risolvono una singola classe di problemi di ricerca, ma molteplici, anche di natura molto differente.
- **I GA sono flessibili.** È semplice modificarli, adattarli e creare varianti.
- **I GA non garantiscono l'ottimalità.** Di norma, producono soluzioni sub-ottimali.

Evolvono una **popolazione** di **individui** (soluzioni candidate) producendo di volta in volta soluzioni sempre migliori rispetto ad una **funzione obiettivo**, fino a raggiungere **l'ottimo** o un'altra **condizione di terminazione**. La creazione di nuove **generazioni** di individui avviene applicando degli **operatori genetici**, precisamente **selezione, crossover e mutazione**.

I quattro aspetti caratterizzanti di un GA sono:

1. **Codifica** gli individui come stringhe di lunghezza finita. Può capitare che in alcune situazioni, per ragioni di efficienza e semplicità, si codifichi in altri modi.
2. È **population-based**: non fa evolvere una singola soluzione, ma un insieme di esse contemporaneamente.
3. È **cieco**: la funzione obiettivo non necessita di informazioni di basso livello del problema che si sta risolvendo. Talvolta, però qualche informazione di basso livello può aiutare la ricerca.
4. Ha elementi di **casualità** che guidano la ricerca, pur non arrivando ai livelli di una random search.

Un tipico algoritmo genetico, nel corso della sua esecuzione, provvede a far evolvere degli individui secondo il seguente schema:

1. Generazione casuale della prima popolazione di individui (codificati con una stringa di bit). Un individuo non è altro che una possibile soluzione al problema in questione.
2. Applicazione della **funzione di fitness** agli individui appartenenti all'attuale popolazione.
3. Selezione degli individui considerati migliori in base al risultato della funzione di fitness.
4. Procedimento di crossover per generare degli individui ibridi a partire dagli individui scelti dal procedimento di selezione.
5. Creazione di una nuova popolazione a partire dagli individui identificati dopo il crossover.
6. Riesecuzione della procedura a partire dal punto 2 ed utilizzando la nuova popolazione creata al punto 5.

L'iterazione dei passi precedenti permette l'evoluzione verso una soluzione ottimizzata del problema considerato.

Poiché l'algoritmo di base soffre del fatto che alcune soluzioni ottime potrebbero essere perse durante il corso dell'evoluzione e del fatto che l'evoluzione potrebbe ricadere e stagnare in "ottimi locali", spesso, questo algoritmo, viene integrato con la tecnica dell'**elitarismo** e/o con la tecnica

delle **mutazioni casuali**. La prima consiste nella copia degli individui migliori della popolazione precedente nella nuova popolazione; la seconda, invece, introduce nelle soluzioni individuate delle occasionali mutazioni casuali in modo da permettere l'uscita da eventuali ricadute in ottimi locali. La mutazione avviene dopo il crossover e prima della creazione della nuova popolazione.

Convergenza

Per *convergenza* si intende la capacità di un GA di migliorare iterativamente le soluzioni candidate verso il punto di ottimo. Quando gli individui tendono a somigliarsi troppo, bloccando troppo presto il progresso globale dell'intera popolazione, parliamo di *convergenza prematura*.

Diversità

Per *diversità* si intende la capacità di un GA di definire individui che possano navigare il panorama degli stati in maniera efficace, evitando la stagnazione verso punti di ottimo locale.

Funzione di fitness

È una funzione in grado di associare un valore ad ogni soluzione. In parole povere, misura il livello di adeguatezza degli individui rispetto al problema considerato e, inevitabilmente, guida il processo di selezione.

Setup di un GA

Un algoritmo genetico si può setizzare in modi differenti a seconda del tipo di problematica che si deve affrontare. Abbiamo a disposizione diversi parametri che consentono al progettista di modificare la struttura dell'algoritmo.

Size popolazione

Rappresenta il numero di individui di ciascuna generazione. Se la size è fissa, bisogna decidere il numero; se, invece, la size è variabile è opportuno decidere una dimensione massima perché la ricerca potrebbe diventare eccessivamente lenta.

Size mating pool

Rappresenta il numero di individui che partecipano alla riproduzione. Più è grande la size, più lento sarà l'algoritmo. Più piccola è la size, più limitata sarà la diversità che l'algoritmo garantirà.

Probabilità crossover

Indica con quale probabilità avviene il crossover tra due genitori.

Possiamo stabilire una probabilità che l'algoritmo userà per valutare se effettuare l'operazione di crossover o meno. Più è alta la probabilità, più spesso i genitori produrranno nuovi individui e, quindi, più diversità ci sarà in termini di soluzioni.

Però bisogna stare attenti perché più sarà alta la probabilità e più è probabile che l'algoritmi arrivi ad una convergenza prematura.

Probabilità di mutazione

Indica con quale probabilità avviene una mutazione. Vale lo stesso discorso per le probabilità di crossover.

Inizializzazione

Con quale criterio di crea la prima generazione. Possiamo creare una popolazione iniziale in maniera totalmente casuale oppure possiamo basarci su *euristiche*. Se riusciamo a rendere i GA meno ciechi (dando informazioni problem-specific) possiamo evitare di iniziare con individui quasi sicuramente pessimi.

Rappresentazione individui

Tipo di codifica degli individui. Gli individui si possono codificare in una stringa binaria o possiamo rappresentarli come meglio crediamo per il problema in esame.

Algoritmo di selezione

Come avviene la selezione degli individui. Può avvenire in modi differenti:

- Casuale: sconsigliata in molti casi, poiché aumenta le chance di non convergenza;
- Roulette Wheel: gli individui ricevono una probabilità di selezione pari al valore della loro fitness relativa all'intera popolazione. Molto fedele alla natura, ma un individuo "molto forte" verrà selezionato troppe volte, rischiano la convergenza prematura;
- Rank: si compie un ordinamento totale degli individui rispetto alla fitness e si assegna a ciascuno di essi il rango in base alla posizione. Ciascun individuo riceve una probabilità di selezione inversamente proporzionata al rango;
- Truncation: si compie un ordinamento totale in maniera analoga ad una Rank Selection, ma la selezione non avviene su base casuale quanto con una selezione rigida dei migliori M individui. Non possono esserci selezioni ripetute;
- K-way tournament: K individui sono selezionati casualmente (formando un *torneo*) e il migliore tra questi K passa la selezione definitivamente. Si ripete finché non si arriva ad M tornei (e quindi M vincitori). Si possono avere selezioni ripetute.
 - Si può adottare una pressione di selezione: invece di far vincere sempre il migliore (assoluto) di un torneo, ogni individuo ha una probabilità di vittoria proporzionata alla sua fitness. In sostanza, i singoli tornei diventano delle piccole Roulette Wheel.
 - Si possono impedire le ripetizioni, in tal caso, se il mating pool è grande quanto la popolazione, ogni individuo ha la garanzia di partecipare ad esattamente K tornei.
 - Osserviamo che con $K=1$, si ha una Truncation Selection. Di fatti, si procederebbe selezionando gli individui migliori fino a M , escludendo gli altri.

Algoritmo di crossover

Come avviene il crossover tra individui. Può avvenire in modi differenti:

- Single Point: si seleziona un punto del patrimonio genetico dei genitori e si procede alla generazione di due figli tramite scambio di cromosomi;
- Two Point: si considerano due punti di incrocio aumentando chiaramente la diversità dei geni degli individui generati;
- K-Point: si usano un numero K di incroci. Poco utilizzato dal momento che richiede la definizione di una strategia di combinazione dei K cromosomi dei genitori;
- Uniform: ciascun gene i -esimo viene scelto casualmente tra i due geni i -esimi dei genitori. Ha il massimo grado di casualità;
- Arithmetic: ciascun gene i -esimo è il valore medio dei geni i -esimi dei genitori. Ne consegue che i due figli saranno *gemelli*. Per differenziare i due figli, è possibile assegnare un peso (fisso o casuale) ai due genitori e fare la media pesata invece che quella aritmetica.

Algoritmo di mutazione

Come avviene la mutazione degli individui. Può avvenire in modi differenti:

- Bit Flip: consiste nella modifica di un singolo gene binario;
- Random Resetting: cambio casuale di un gene ad un altro valore ammissibile;
- Swap: scambio di due geni scelti casualmente;
- Scramble: si sceglie un subset di geni in modo casuale e lo si permuta casualmente;
- Inversion: si sceglie un subset di geni in modo casuale e lo si ribalta.

Stopping condition

Con quale criterio decidiamo di terminare l'evoluzione oppure proseguire.

- Tempo di esecuzione: l'evoluzione termina se si è superato un tempo di esecuzione T , allo stop o si decide di restituire l'ultima generazione ottenuta o la migliore ultima;
- Costo: se è presente una funzione di costo allora al raggiungimento o superamenti di quel costo l'evoluzione termina;
- Numero di iterazioni: si itera per un massimo di X generazioni;
- Assenza di miglioramenti: se per Y generazioni consecutive non ci sono miglioramenti, allora l'evoluzione termina;
- Ibride: ovvero, basate su una combinazione delle precedenti;
- Problem specific: conoscendo i dettagli del problema, possiamo definire condizioni ad-hoc.

Tecniche di improvement degli algoritmi genetici

Improvement #1: Il concetto di elitismo

In fase di selezione, un individuo con alta fitness ha alte chance di sopravvivenza, ma la selezione può essere comunque crudele contro di lui. Allora per non perdere questo individuo si adotta il concetto di **elitismo**: salviamo il migliore (o i migliori k) individuo e lo copiamo della generazione successiva.

Improvement #2: Uso di euristiche problem-specific

È vero che i GA sono ciechi, ma ciò non toglie che l'utilizzo di informazioni aggiuntive possano rivelarsi utili. Queste possono velocizzare i miglioramenti, aumentare la diversità e ridurre il rischio di convergenza prematura.

Improvement #3: Problemi multi-obiettivo

Nel parlare della funzione di fitness, si è sempre discusso di funzioni mono-obiettivo. In realtà però ci troveremo di fronte a problemi con n funzioni di fitness da massimizzare o minimizzare. Un individuo quindi non avrà più un singolo valore di fitness, ma n . O meglio, un *vettore di fitness lungo n* .

Di conseguenza, in fase di valutazione dovremmo ragionare diversamente rispetto a quanto fatto nei problemi mono-obiettivo. Quindi si ricorre al **fronte di Pareto**: insieme di individui di una popolazione non migliori tra loro secondo, due regole, ma migliori di tutti gli individui al di fuori dal fronte.

Le due regole del fronte di Pareto, riguardante gli individui del fronte, sono: tutte le fitness corrispondenti non peggiori e almeno una fitness strettamente migliore.

Dati due individui del fronte di Pareto, come decide quale tra questi sia il migliore?

Preference Sorting: tecnica che permette di fornire un ordinamento totale tra gli individui nel fronte di Pareto attraverso l'uso di una funzione di preferenza.

La funzione di preferenza è diversa dalla funzione di fitness. Quasi sempre, il criterio che governa questa funzione *deriva necessariamente dalla conoscenza del problema*.

Possiamo introdurre quanti criteri di preferenza vogliamo. Tutto dipende da quante informazioni riusciamo a ricavare dal problema e da ciò che desideriamo.

Improvement #4: Il concetto di archivio

Archive strategy: strategia che mantiene una popolazione aggiuntiva che *non evolve*, contenente gli individui che riescono a soddisfare degli obiettivi mai soddisfatti nelle precedenti iterazioni.

Dopo la stopping condition dell'algoritmo, invece di restituire soltanto l'ultima generazione, restituisco l'archivio, perché so già che contiene un insieme di individui molto forti quanto unici.

Improvement #5: Il concetto di dynamic target selection.

In problemi complessi, potrei avere centinaia/migliaia di funzioni di fitness. Il fronte di Pareto inizia ad ingrandirsi; un criterio di preferenza "debole" non mi sarà d'aiuto.

In generale, per evitare di avere troppe fitness, è consigliato usare *funzioni indipendenti tra loro*, ma dove ciò non è possibile si ha una situazione del genere: "se non minimizzo prima f1 non potrò mai pensare di minimizzare f2".

Di conseguenza, se nella mia popolazione non ho alcun individuo che minimizzi f1, a che serve che nella successiva iterazione consideri anche f2 durante la valutazione?

Dynamic Target Selection: tecnica che, ad ogni iterazione, ci permette di restringere l'insieme delle funzioni di fitness da valutare a seconda del risultato dell'iterazione precedente.

Teoria dell'apprendimento

L'apprendimento consiste in qualunque processo tramite il quale un sistema migliora le sue prestazioni sulla base dell'esperienza. In questo caso si parla di agenti intelligenti che migliorano automaticamente operando in un ambiente.

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 E.

Un agente capace di apprendere presenta il vantaggio di saper operare in ambienti inizialmente sconosciuti; esso è formato da quattro componenti principali:

- **Elemento di apprendimento:** l'elemento responsabile del miglioramento interno;
- **Elemento esecutivo:** l'elemento responsabile della selezione delle azioni esterne;
- **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 significative che, chiaramente, portino l'agente ad apprendere nuove conoscenze da sfruttare poi per migliorare le sue azioni.

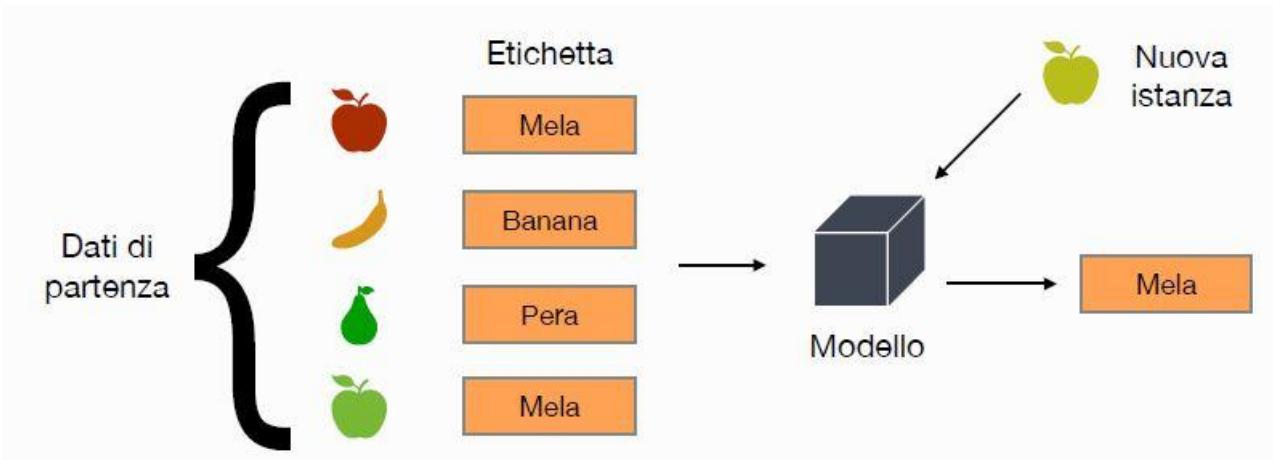
Machine learning

Il **machine learning** è una branca dell'intelligenza artificiale che si occupa dello studio e della costruzione di algoritmi che possano *imparare dai dati* e, sulla base di questi, fare delle previsioni.

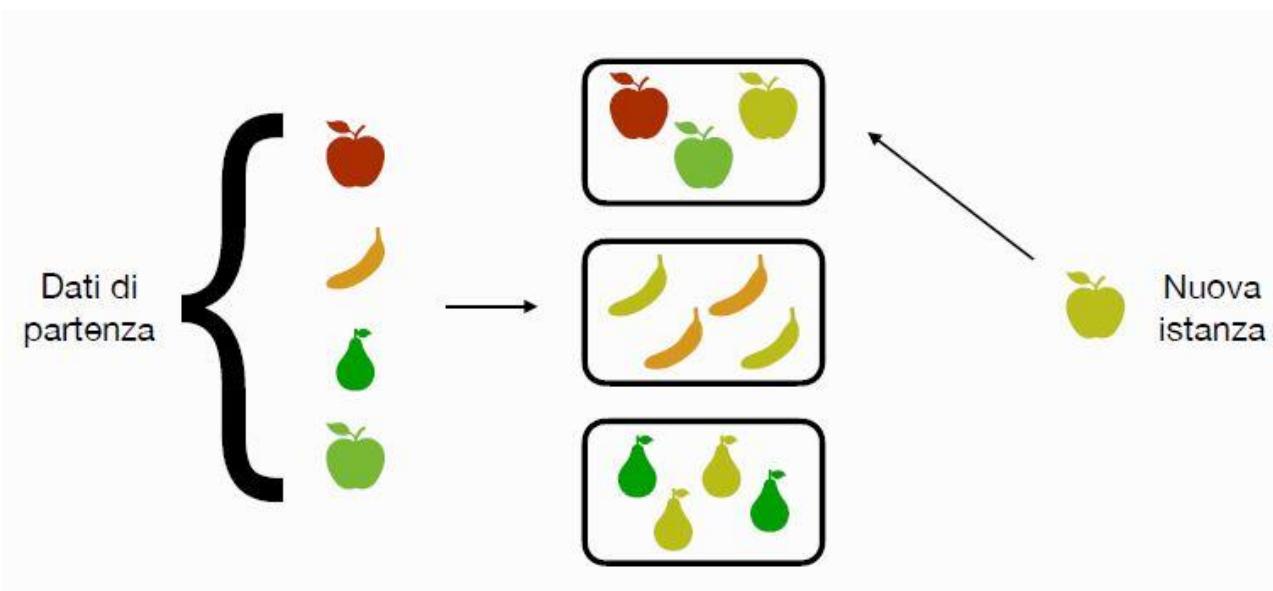
Infatti, gli algoritmi di machine learning, non si limitano ad eseguire dei comandi esplicativi, ma sono in grado di prendere **decisioni data-driven**, ossia prendere decisioni in base ai dati, alle informazioni di cui si dispone, e non in base a dinamiche soggettive e a sensazioni personali.

Esistono due tipi di apprendimento:

- **Apprendimento supervisionato:** è quello in cui l'agente apprende usando dei *dati etichettati*. Le etichette determinano la *variabile dipendente*, ovvero quello che l'agente dovrà apprendere. (*variabile dipendente* è la *y* in una funzione, quindi il risultato finale dato un certo input detto *variabile indipendente*);



- **Apprendimento non supervisionato:** è quello in cui l’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.



Oltre alla classica distinzione tra apprendimento supervisionato e non, 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 compirà azioni in maniera sequenziale e, al termine di ogni sequenza, gli verrà assegnata una “ricompensa” che ha lo scopo di incoraggiare comportamenti corretti.

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

- Se l'output è continuo, allora si parla di **regressione** (es. stimare lo stipendio di una persona in base al titolo di studio);
- Se l'output è discreto, allora si parla di **classificazione** (es. valutare se una e-mail è spam in base al suo oggetto);
- Se l'output è la suddivisione dei dati in gruppi, allora si parla di **clustering** (es. 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 vasto insieme di dati. Oppure c'è il *mining delle associazioni* che serve ad identificare dei pattern comuni in un insieme di transizioni. Ad esempio, il "market basket analysis".

Qual è l'algoritmo migliore?

La risposta è ovviamente: *dipende*. Esistono centinaia di algoritmi di machine learning che si possono utilizzare per risolvere problemi. A prescindere dallo specifico algoritmo da utilizzare, è importante innanzitutto caratterizzare il problema da trattare, quindi capire di che genere è (supervisionato, non supervisionato, di clustering, di classificazione, di regressione, ...).

Dato che per ogni problema ci sono vari algoritmi, come facciamo a scegliere quale usare?

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.

Un modo per capire se un algoritmo è adatto ad un determinato problema è quella di **misurare l'errore**: ovvero si costruiscono diversi modelli, si calcola l'errore per ciascuno di essi e si sceglie quello con i migliori risultati.

In altri termini, la misura dell'errore ci fornisce una base di confronto tra più modelli. Per capire meglio tutto ciò, consideriamo l'esempio di un problema di apprendimento supervisionato di regressione: la costruzione di un modello di stima dell'altezza di una persona in base al peso, genere ed età.

Errore, Bias e Varianza

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

Nell'esempio che tratteremo, l'errore è la differenza tra l'altezza stimata e quella reale. Possiamo rappresentare i dati di input per ciascuna persona del nostro dataset con un vettore del tipo:

$$\vec{x} \equiv (x_1, x_2, \dots x_p)_i$$

Nel nostro caso $p=3$ poiché abbiamo tre parametri sulla base dei quali fare predizione e, quindi, la terna $(x_1, x_2, x_3)_i$ rappresenta genere, peso ed età dell'i-esima persona.

La nostra variabile dipendente, di tipo continuo, sarà data dall'altezza stimata e verrà chiamata y_i .

Costruire un modello di machine learning significa trovare quella funzione f per cui:

$$y_i = f(\vec{x}_i)$$

Due esempi:

$$f(u, 70\text{kg}, 35) = 170\text{cm}; \\ f(d, 60\text{kg}, 35) = 163\text{cm}.$$

Se l'altezza di queste due persone fosse proprio quella riportata dalla funzione, allora il nostro errore sarebbe pari a zero.

Tuttavia, sappiamo che persone dello stesso peso, genere ed età possono avere altezze diverse. Ad esempio, nel nostro dataset potremmo avere:

$$\vec{x}_3 = (u, 70, 35); y_3 = 170;$$

$$\vec{x}_4 = (u, 70, 35); y_4 = 180;$$

$$\vec{x}_5 = (u, 70, 35); y_5 = 175;$$

La nostra funzione f , a parità di input, restituisce sempre lo stesso output. Questo implica che il fenomeno ha una variabile intrinseca che rende impossibile la creazione di un modello perfetto.

Per questa ragione, si introduce il concetto di *errore irriducibile*, ovvero una misura di variabilità intrinseca del fenomeno in esame —> in altri termini, quell'errore che avremo sempre e comunque.

$$y_i = f(\vec{x}_i) + \epsilon_{irr}$$

Ma l'errore irriducibile non è l'unico errore da considerare... abbiamo anche l'errore dovuto al modello di machine learning.

Se l'errore irriducibile *dipende esclusivamente dai dati* che abbiamo a disposizione, dobbiamo considerare che i modelli di machine learning *non sono infallibili* e potrebbero produrre predizioni errate.

Se indicassimo con \tilde{y}_i l'altezza stimata dal modello, allora possiamo dire che:

$$y_i = \tilde{y}_i + \epsilon_{irr} + \epsilon$$

Tutti gli sforzi relativi alla ricerca del miglior modello di machine learning puntano a minimizzare l'errore riducibile. Questo errore dipende da due fattori principali.

Bias. Il modello ha un certo bias se, quando viene addestrato su diversi dataset, l'output che restituisce è sistematicamente sbagliato.

Il bias indica l'insieme di assunzioni usate dal modello per prediri un valore di output dati degli input che non ha ancora incontrato (anche detto underfitting).

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 (approfondiremo più avanti).

Tutti gli sforzi relativi alla ricerca del miglior modello di machine learning puntano a minimizzare l'errore riducibile. Questo errore dipende da due fattori principali.

$$\epsilon = \text{bias} + \text{varianza}$$

Quindi, l'obiettivo è quello 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.

Che implicazioni ha tale compromesso sulle prestazioni dei modelli di machine learning?

Underfitting e overfitting

Underfitting

Un modello con bias elevato è più semplice di quanto dovrebbe essere e quindi tende a sotto-dimensionare i dati. In altre parole, è uno studente superficiale: il modello non riesce ad apprendere e acquisire gli schemi intricati del set di dati.

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

Altrettanto chiaramente, un modello di questo tipo **non potrà risolvere problemi complessi**, ovvero problemi per i quali l'insieme di dati di input è particolarmente intricato (come, ad esempio, il problema visto in precedenza, in cui più persone con le stesse caratteristiche hanno altezze diverse).

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

Fatte queste premesse, un modello di questo tipo tenderà a risolvere **problemi semplici utilizzando soluzioni complesse**, non essendo capace di generalizzare le competenze acquisite sull'insieme di dati di input.

Underfitting e overfitting, l'importanza della progettazione e della valutazione empirica

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 mitigare i rischi di underfitting e overfitting lavorando sulla configurazione degli algoritmi di machine learning.

Alcune operazioni tipicamente utilizzate riguardano:

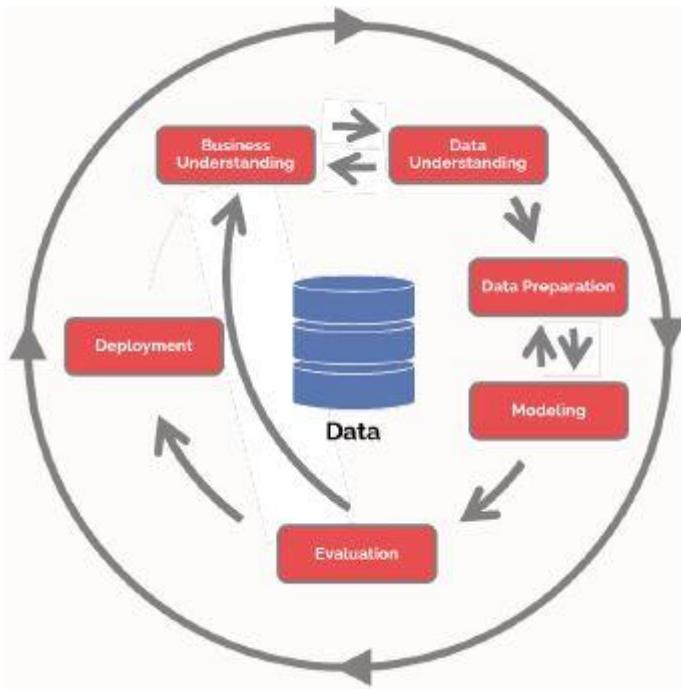
- La selezione delle caratteristiche rilevanti, tramite la quale un algoritmo di machine learning riesce ad apprendere "meglio", focalizzando l'attenzione sui soli dati che rappresentano la variabile dipendente;
- La convalida incrociata, tramite la quale vengono generate una serie di insiemi di dati 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.

Ingegneria del Machine Learning

L'ingegneria del software è la chiave di tutto il software che c'è al mondo... senza questa, semplicemente, non potremmo sviluppare alcun sistema software affidabile o nessun sistema che consenta di fornire funzionalità agli utenti.

Quando si vuole progettare una soluzione basata su machine learning bisogna pensare a due cose fondamentali: data e software engineering. Il tutto può essere descritto dal cosiddetto modello **CRISP-DM** (Cross-Industry Standard Process for Data Mining), che rappresenta il ciclo di vita di progetti basati su intelligenza artificiale e data science.

Il CRISP-DM è un modello *non* sequenziale in cui le diverse fasi possono essere eseguite un numero illimitato di volte. Quindi, il modello è di fatto flessibile e può essere considerato sia tradizionale che agile.



La prima fase è chiaramente quella di raccolta dei requisiti e definizione degli obiettivi di business che si intende raggiungere.

La fase di **business understanding** definisce il Piano di progetto, un documento che spiega l'esecuzione del progetto dal punto di vista di gestione tecnica e socio-tecnica. In questo documento vengono riportati:

- I *business success criteria*, ovvero criteri che stabiliscono se il sistema che si sta sviluppando è in linea con gli obiettivi di business definiti precedentemente;
- Gli obiettivi tecnici;
- Viene determinata la disponibilità delle risorse;
- Vengono stimati i rischi;
- Vengono definiti i *piani di contingenza*, ovvero dei programmi operativi che delineano preventivamente le azioni di determinati soggetti nel caso in cui si verifichi un evento dannoso;
- Viene condotta un'*analisi costi-benefici*;
- Vengono selezionate le tecnologie e i tool necessari allo sviluppo del sistema.

La fase di **data understanding** definisce il Documento di analisi dei dati che riporta i metodi di estrazione ed analisi dei dati, oltre che le relazioni tra essi ed eventuali problemi di qualità. Innanzitutto, quindi, vengono *acquisiti i dati necessari* al raggiungimento degli obiettivi prefissati. I dati poi vengono caricati in un tool di analisi dei dati dove vengono *esaminati e documentati*. Poi vengono *esplorati*, ovvero vengono visualizzati e vengono identificate eventuali relazioni tra essi.

Infine vengono identificati e documentati possibili problemi di qualità dei dati (ad esempio, dati mancanti).

La fase di **data preparation** definisce l'Insieme di dati di input, ovvero l'insieme dei dati che verranno considerati in fase di modellazione dell'algoritmo di machine learning. Questo include il *feature engineering*, una tecnica di apprendimento automatico che sfrutta i dati per creare nuove variabili che non sono presenti nel dataset di addestramento. L'obiettivo è quello di semplificare e accelerare le trasformazioni dei dati migliorando al contempo l'accuratezza del modello.

La fase di **data modeling** definisce il modello di machine learning, ovvero l'algoritmo addestrato e configurato sui dati a disposizione. In primo luogo va selezionata la tecnica o l'algoritmo da utilizzare. 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.

La fase di **evaluation** 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.

La fase di **deployment** definisce il report finale del progetto, descrivendo tutte le fasi condotte, oltre che il piano di manutenzione e monitoraggio.

SCRUM

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 determinati requisiti vengono definiti, analizzati, progettati e sviluppati.

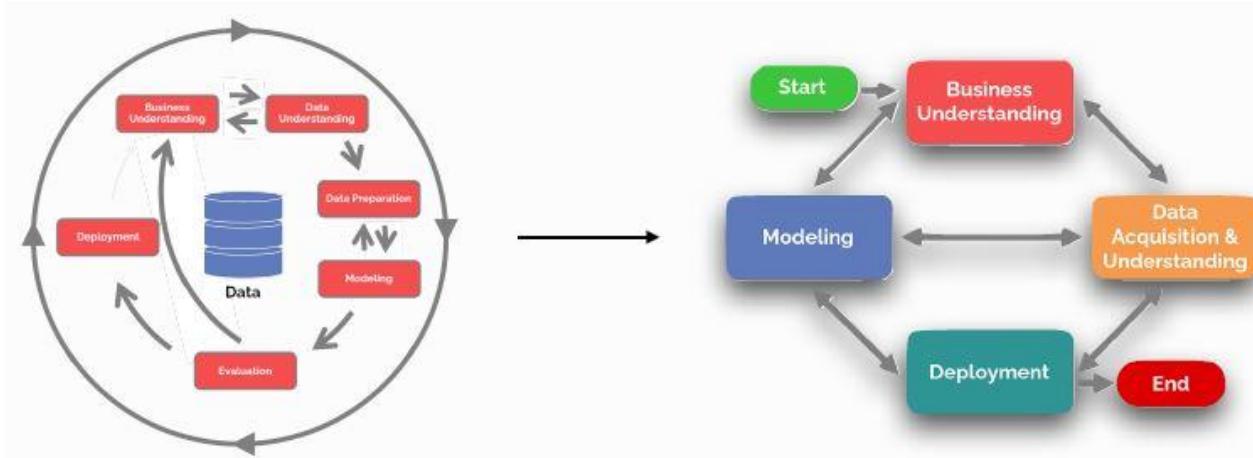
A differenza dei cicli di vita tradizionali, SCRUM prevede lo sviluppo incrementale di porzioni del sistema. L'incrementalità serve da un lato ad avere una migliore interazione con gli stakeholder (dovuta al rilascio di piccole parti del sistema) e dall'altro a minimizzare i rischi (che nel processo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

SCRUM prevede tre ruoli fondamentali:

- Il *product owner*, colui il quale rappresenta gli stakeholder;
- Lo *SCRUM Master*, colui che coordina le attività di sviluppo (da non confondere con il project manager che ha la responsabilità anche di gestione del personale);
- Il *team di sviluppo*.

In SCRUM ogni sprint inizia con uno *sprint planning*, in cui vengono definite le attività da compiere. Il lavoro è monitorato tramite i *daily scrum*, una riunione quotidiana in cui vengono riportati i progressi. Alla fine dello sprint, viene effettuata una *sprint review*, dove il team mostra i progressi effettuati. Infine, lo sprint viene chiusa con una *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 **TDSP** (Team Data Science Process).



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, il TDSP definisce sei ruoli esplicativi:

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

TDSP, pro e contro

By design, TDSP enfatizza la necessità di rilasci incrementali, il ché minimizza i rischi connessi a potenziali misunderstandings dei requisiti.

Più importante, TDSP riconosce esplicitamente 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, oltre che esperti di data engineering e science.

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

Di contro, la necessità di definire degli sprint 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!

Qualità dei dati e Feature Engineering

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

I *leaky predictor* sono quelle caratteristiche che ci aiutano sicuramente a caratterizzare il problema, ma che nella pratica non saranno disponibili il più delle volte, potenzialmente causando quindi il fallimento del modello.

Tipologie di dati

Un **dato**, in termini di machine learning, è un qualsiasi elemento di cui si dispone per formulare un giudizio o risolvere un problema.

Esistono diversi tipi di dati:

- **Dati strutturati:** sono dati tabulari per cui righe e colonne sono ben definite. Sappiamo per esattezza il significato del dato. Spesso questi dati sono memorizzati in basi di dati che rappresentano anche le relazioni tra i dati. In questo caso, i dati possono essere recuperati tramite query;
- **Dati non strutturati:** sono 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;
- **Dati semi-strutturati:** in questo caso il formato è a metà tra lo strutturato e il non strutturato. Mentre il formato è fissato, la struttura non ha una definizione stretta. Ad esempio, dati tabulari potrebbero avere dati mancanti o dati espressi tramite formato non strutturato. I dati semi-strutturati sono generalmente memorizzati come file. Alcuni però potrebbero anche essere presenti all'interno di basi di dati document-oriented.

Ingegneria dei dati

La fase di “*data preparation*”, quella in cui avviene la preparazione dei dati da utilizzare in input nel modello di machine learning, viene suddivisa in quattro fasi: **data cleaning, feature scaling, feature selection e data balancing**.

Il “*data cleaning*” è la fase in cui si stabiliscono tecniche per ovviare a problemi di mancanza dei dati o problemi di dati rumorosi. L'insieme di queste tecniche è detto **Data Imputation**.

Per il problema dei dati mancanti esistono due soluzioni abbastanza banali: *scartare le righe del dataset* che presentano dati mancanti, soluzione facile ma non sempre applicabile. Se non si hanno tante osservazioni, scartare le righe diventa un problema; *scartare le colonne del dataset* che presentano dati mancanti, soluzione facile ma non sempre applicabile. Se la colonna rappresenta una caratteristica rilevante per il problema in esame, non possiamo scartarla.

In alternativa, possiamo stimare il valore dei dati mancanti basandoci su applicazioni di semplici tecniche statistiche. Questa soluzione però, non può essere applicata su dati non-numerici e non considera l'incertezza quando va ad imputare i dati.

L'imputazione dei dati può essere:

- *Most frequent imputation*: i dati mancanti sono sostituiti dal valore più frequente contenuto in una colonna. Però se ci sono troppi dati mancanti, questa tecnica rischia di influenzare eccessivamente la distribuzione della variabile.
- *Imputazione deduttiva*: il progettista definisce una regola di imputazione sulla base di una deduzione logica. Non è sempre facile trovare una regola valida e sicuramente non è un approccio scalabile.

In alcuni casi ci può decidere di combinare le tecniche.

Quando ci troviamo di fronte a dati non strutturati, il caso tipico è quello di un testo scritto in linguaggio naturale. Abbreviazioni, errori di battitura, errori grammaticali, ed altro, rendono il mining di testi molto complicato. Si necessita quindi di alcuni step dedicati ad *estrarre della semantica da testi dove la semantica è nascosta*.

Innanzitutto, sono tre i principali problemi da affrontare quando si ha a che fare con il testo: difficoltà di estrazione; ambiguità del linguaggio; esistono molti modi di esprimere concetti simili.

Un processo standard di normalizzazione di un testo prevede i seguenti step:

```
connect = DBConnection.getConnection();
```

↓ Se ci interessa dare un senso ai simboli, allora dovremo prima di tutto procedere a sostituirli con delle parole.

```
connect equals DBConnection calls getConnection();
```

Nell'esempio, non ci interessa sostituire il simbolo '()' perché non aggiunge semantica!

La sostituzione è possibile definendo un dizionario, ovvero una struttura che restituisce la parola associata ad un simbolo.

↓ Eliminiamo quindi tutto ciò che può solo creare rumore.

```
connect equals DBConnection calls getConnection();
```

↓ Per consentire ad un algoritmo di "comprendere" il testo, non possiamo avere situazioni in cui più parole siano concatenate.

```
connect equals DB Connection calls get Connection
```

Questo operazione è fatta sulla base si euristiche (camelCase, underscore, ecc.)

↓ Tramite l'utilizzo di dizionari, possiamo procedere poi a quello che definiamo *contraction expansion*.

```
connect equals database Connection calls get Connection
```

In inglese, la contraction expansion consente spesso di disambiguare il significato delle frasi (ad esempio, l'espansione di "wouldn't" in "would not" consente di definire una negazione).

↓ Connect e connection fanno riferimento alla stessa semantica, ma sono scritte in maniera diversa!

```
connect equals database Connect calls get Connect
```

Questo step si definisce *stemming* e consiste nel processo di sostituzione di una parola nella sua radice.

Anche qui, si fa riferimento ad un dizionario.

NB: Nell'esempio specifico, potremmo anche considerare i nomi dei metodi come nomi propri, ma dipende da quello che vogliamo fare!

↓ Connect, Connect e connect sono la stessa cosa? Non proprio! Riduciamo tutto in maiuscolo/minuscolo.

```
connect equals database connect calls get connect
```

A volte sono necessari altri step:

1. Spelling correction;
2. Filtrare verbi e sostantivi;
3. Singolarizzazione;
4. Rimozione delle ripetizioni;
5. Rimozione di documenti;
6. Stopword removal.

Il “*feature scaling*” è l'insieme di tecniche che consentono di normalizzare o scalare l'insieme di valori di una caratteristica. Questa normalizzazione serve a non far confondere l'algoritmo di apprendimento, perché potrebbe sottostimare/sovrastimare l'importanza di una caratteristica poiché questa ha una scala di valori molto inferiore/superiore rispetto alle altre.

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]).

Lunghezza del testo	$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$	Lunghezza del testo
300		0,38
180	Il minimo della distribuzione è 180; Il massimo è 492. Vogliamo normalizzare in una scala da 0 a 1. Prendiamo il caso di 300:	0
222	$x' = 0 + \frac{(300 - 180)(1 - 0)}{492 - 180} = \frac{120}{312} = 0,38$	0,13
492	Nella distribuzione normalizzata, 180 sarà uguale a 0, mentre il valore 492 sarà uguale ad 1.	1
300		0,38

Una alternativa spesso considerata è quella della *z-score normalization*.

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

Dove x è il valore originale, \bar{x} è la media della distribuzione, σ la deviazione standard. Questa è la normalizzazione di default implementata da molti dei tool di machine learning.

Il “*feature selection*” è il processo tramite il quale vengono selezionate le caratteristiche più correlate al problema in esame, a partire da un insieme di caratteristiche esistenti.

Il modo più semplice per approcciare il problema della selezione delle feature 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. (se una variabile è simile nell’intero dataset, è probabile che non sia discriminante);
- *Eliminazione univariata di feature*: questo metodo prevede la selezione delle variabili sulla base di test statistici. Ogni variabile indipendente viene correlata ad una variabile dipendente, ottenendo quindi una classifica delle variabili basata sulla correlazione. Si sceglieranno solo le k migliori variabili.

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).

Se pensiamo ad esempio ai problemi di carattere medico, come classificare una polmonite, per questi problemi il numero di pazienti affetti dalla malattia è molto molto inferiore al numero di pazienti malati. Se non consideriamo il problema dello sbilanciamento dei dati, definiremmo molto probabilmente 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 (ci interessa trovare i malati, non chi sta bene).

E qui entra in gioco il “*data balancing*”, ovvero l’insieme di tecniche che servono a convertire un dataset sbilanciato in un dataset bilanciato. Sono due principalmente le tecniche applicabili:

- **Undersampling**: metodo tramite il quale vengono causalmente eliminate un numero di istanze del dataset della classe di maggioranza.
Il problema è che se ho un numero eccessivamente 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 nel fatto che si potrebbero rimuovere istanze particolarmente rilevanti per l’apprendimento del modello.
- **Oversampling**: metodo tramite il quale vengono casualmente aggiunte un numero di istanze del dataset della classe di maggioranza.
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.

Problemi di classificazione

Un problema di classificazione porta alla costruzione di un **classificatore** per classificare i nuovi elementi sulla base del training set.

La classificazione ha come scopo l’unico obiettivo di predire il valore di una variabile categorica, chiamata “variabile dipendente” (o target) tramite l’utilizzo di un training set, ovvero un insieme di osservazioni per cui la variabile target è nota.

Esistono diversi tipi di classificatori, in particolare il *Naive Bayes* e il *Decision Tree*.

Naive Bayes

L’algoritmo considera le caratteristiche della nuova istanza da classificare e ne 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. Di conseguenza, l’algoritmo non andrà a valutare in fase di classificazione la potenziale utilità data dalla combinazione di più caratteristiche. Quindi considererà tutte le proprietà come indipendenti: 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.

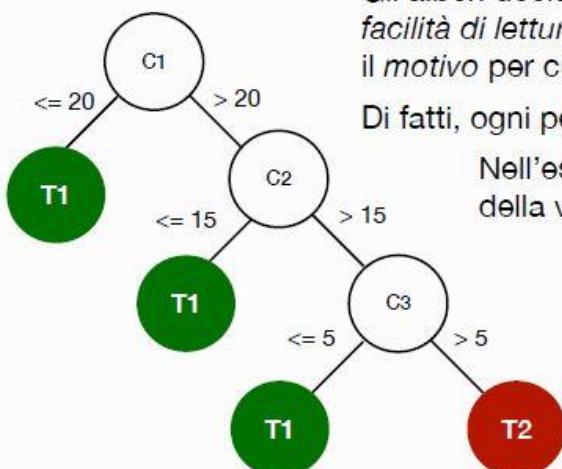
La classificazione con Naive Bayes avviene secondo tre step:

1. **Calcolo della probabilità della classe:** le probabilità di classe sono semplicemente le *frequenze* delle istanze che appartengono a ciascuna classe divisa per il numero totale di istanze;
2. **Calcolo della probabilità condizionata:** si applica il teorema di Bayes, per determinare le probabilità condizionate delle caratteristiche del problema;
3. **Decisione:** viene identificata nella classe ottiene il valore di probabilità più elevato.

Decision Tree

L'algoritmo mira a creare un albero i cui nodi rappresentano un sotto-insieme di caratteristiche del problema e i cui archi rappresentano delle decisioni.

L'obiettivo di un albero decisionale è predire il valore di una variabile target apprendendo semplici *regole di decisione* inferte 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.

Di fatti, ogni percorso dell'albero corrisponde ad una regola.

Nell'esempio, potremmo motivare la predizione della variabile target T2 nel seguente modo:

$$C1 > 20 \wedge C2 > 15 \wedge C3 > 5 \Rightarrow T2$$

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 tre passi.

1. Posiziona la miglior caratteristica del training set come radice dell'albero;
2. Dividi il training set in sotto-insiemi. Ogni sotto-insieme dovrebbe essere composto da valori simili per una certa caratteristica;
3. Ripetere gli step 1 e 2 su ogni sotto-insieme fin quando non viene raggiunto un nodo foglia in ogni sotto-albero.

Per decidere con quale attributo dividere il dataset si ricorre all'**information gain**: una misura che indica il *grado di purezza* di un attributo, ovvero quanto un certo attributo sarà in grado di dividere adeguatamente il dataset.

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).

Nella teoria dell'informazione, l'entropia indica in che misura un messaggio è ambiguo e difficile da capire:

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

(Maggiore è l'entropia, minore sarà l'ammontare di informazione del messaggio)

In particolare, per ogni attributo del dataset calcoleremo il suo gain:

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

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. Calcola l'entropia per ogni attributo del dataset;
2. Dividi il training set in sotto-insiemi, utilizzando l'attributo per cui l'entropia sia minimizzata o, in maniera equivalente, l'information gain è massimizzata;
3. Crea un nodo dell'albero decisionale contenente quell'attributo;
4. Ripeti i passi precedenti fin quando tutti i sottoinsiemi definiti non siano puri.

Come decidere quale classificatore va usato per un determinato problema?

La soluzione è quella di sperimentare più classificatori e valutare le prestazioni di classificazione ottenute. Chiaramente, non possiamo valutare un classificatore su dati che non conosciamo, altrimenti non potremmo misurare le sue prestazioni. Ma, avendo un dataset di partenza etichettato, possiamo sfruttare questa conoscenza per capire come un classificatore classifica questi dati.

Dovemmo però simulare un'esecuzione reale dell'algoritmo simulando la presenza di istanze la cui etichetta non è nota.

Addestrare e validare un modello di machine learning sullo stesso dataset è un grosso errore metodologico che porta ad avere risultati totalmente inaffidabili.

Possiamo però dividere il dataset di partenza in maniera tale da considerare alcune delle istanze come non note. Creeremo quindi due insiemi:

- Il **training set**, che sarà composto dalle istanze che l'algoritmo utilizzerà per l'addestramento;
- Il **test set**, che sarà composto dalle istanze per cui l'algoritmo addestrato dovrà predire la classe di appartenenza.

Esistono diversi modi di dividere un dataset, tra cui c'è il metodo chiamato convalida incrociata.

La **convalida incrociata** è un metodo statistico che consiste nella ripetuta partizione e valutazione dell'insieme dei dati di partenza. Prevede i seguenti passi:

1. Mischiare in maniera casuale i dati di partenza;
2. Dividere i dati in k gruppi;
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.

(N.B.: tutti i processi di normalizzazione (es. feature selection, data balancing) vanno effettuati ad ogni addestramento!)

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 (leaky validation strategy). Questo può portare a problemi di **data leakage**.

La convalida incrociata può però avere alcuni problemi:

1. Essendo casuale, il primo passo della validazione potrebbe inavvertitamente portare un **vantaggio** al classificatore, ad esempio una divisione dei gruppi irrealistica.

Una soluzione può essere quella di ripetere N volte la validazione, in modo da limitare l'influenza della casualità del primo passo.
Un'altra soluzione può essere quella di 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;
2. La convalida incrociata non può essere usata facilmente nel caso in cui i dati seguano un ordine temporale perché li mischierebbe portando ad un caso irrealistico.

Validare un modello di machine learning: Metriche di valutazione

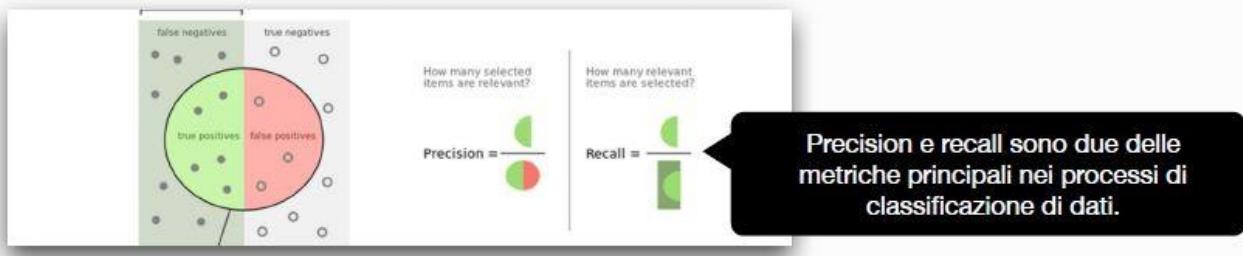
A prescindere dalla procedura di validazione che utilizzeremo, avremo bisogno di strumenti adatti per valutare la bontà delle predizioni.

Parliamo quindi della *matrice di confusione*, anche detta tabella di errata classificazione, la quale restituisce una rappresentazione dell'accuratezza di un classificatore.

	Istanze realmente positive	Istanze realmente negative
Istanze predette come positive	Veri positivi	Falsi positivi
Istanze predette come negative	Falsi negativi	Veri negativi

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.



Definendo come TP il numero di veri positivi, come FP il numero di falsi positivi, come TN il numero di veri negativi, e come FN il numero di falsi negativi:

$$Precision = \frac{TP}{(TP + FP)}$$

$$Recall = \frac{TP}{(TP + FN)}$$

$$Specificity = \frac{TP}{(TN + FP)}$$

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN + FN)}}$$

Ogni metrica fornisce un'indicazione complementare per la valutazione del modello di classificazione. 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. E' chiaramente 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. E' chiaramente una metrica che vogliamo massimizzare.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

Un'attenzione particolare richiede però l'accuracy. Per definizione, questa indica il numero totale di predizioni corrette (sia della classe positiva che negativa).

Al numeratore abbiamo anche i veri negativi, ovvero il numero di istanze correttamente classificate come negative. Questo potrebbe creare un problema di interpretazione nel caso di dataset sbilanciati dove il numero di casi positivi è molto basso...

In questi casi, il classificatore sarà sicuramente più abile a riconoscere i veri negativi, poiché sono molti di più. E quindi, che significa avere una accuracy del 99%?

Immaginate un problema di classificazione dei melanomi, dove (per fortuna) il numero di veri positivi è molto basso: un'accuracy altissima potrebbe far pensare che il classificatore sia efficacissimo, ma è solo un'illusione: il classificatore riconoscerà bene chi non ha il melanoma... ma a noi interessa scoprire chi, invece, ne ha affetto!



Le metriche forniscono indicazioni, ma vanno interpretate correttamente sulla base dei problemi che si analizzano. Mai fidarsi ciecamente delle metriche e, soprattutto, attenzione all'uso che fate di queste!

Problemi di regressione (slide)

Problemi di clustering

Un problema di clustering ha l'obiettivo di classificare i dati, ma senza assegnare loro un'etichetta. Infatti, non abbiamo classi predefinite ma ogni cluster può essere interpretato come una classe di *oggetti simili*, ovvero aventi simili caratteristiche.

L'obiettivo è raggruppare oggetti in gruppi che abbiano un certo grado di omogeneità ma che, al tempo stesso, abbiano un certo grado di eterogeneità rispetto agli altri gruppi.

(Da un lato, vogliamo minimizzare la distanza tra gli elementi di un cluster, massimizzando quindi la similarità tra di loro. Dall'altro vogliamo massimizzare la distanza tra più cluster, minimizzando quindi la similarità tra di loro.)

La qualità di un algoritmo di clustering dipende dalla misura di similarità utilizzata: cioè la sua abilità di scoprire alcuni o tutti i pattern nascosti, ovvero le caratteristiche che legano elementi simili; questo però è un qualcosa che non potremmo calcolare in maniera automatica poiché non abbiamo le etichette di partenza.

Altre proprietà che rendono un algoritmo di clustering *buono* possono essere la scalabilità, la robustezza agli outlier (in un insieme di osservazioni, è un valore anomalo chiaramente distante dalle altre osservazioni disponibili).

Misure di similarità

La più ovvia misura di similarità (o di diversità) tra due pattern è la distanza fra essi. **Ma non sempre è così:** non sempre la distanza tra due pattern è significativa per indicare la diversità!

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

Così facendo, fare cluster sarebbe semplicissimo, poiché potremmo definire una soglia sulla distanza e raggruppare i pattern al di sotto di questa soglia.

Il problema però diventa rispondere alla seguente domanda: **Qual è la giusta soglia da utilizzare?**

Per rispondere a questa domanda, 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. *Disegualanza triangolare:* $\forall x, y, z \in S, d(x, z) \leq d(x, y) + d(y, z)$.

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

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

A seconda delle esigenze, potremmo dover ricorrere ad altre tipologie di similarità. Ad esempio, nel caso di testi, non potremmo neanche calcolare la distanza euclidea.

Altre metriche di similarità sono le seguenti:

- **Distanza di Manhattan;**
- **Distanza di Mahalanobis;**
- **Distanza di Jaccard;**
- **Distanza di Hamming;**
- **Distanza di Levenshtein.**

Sebbene ci siamo molte altre metriche è impossibile dire quale sia meglio di un'altra; questo dipende dal problema.

Per identificare la “giusta” soglia, si utilizza il criterio della *somma degli errori quadrati*.

Problemi di raggruppamento: Criteri da adottare

Per identificare la “giusta” soglia, dobbiamo poi scegliere il criterio da utilizzare in fase di ottimizzazione dei cluster.

Più formalmente, supponiamo di avere un insieme $D = \{x_1, x_2, \dots, x_n\}$ composto da n campioni, che vogliamo partizionare in esattamente k insiemi disgiunti D_1, D_2, \dots, D_k .

Ogni sottoinsieme rappresenta un cluster, con i campioni nello stesso cluster che sono per qualche motivo più simili l'uno l'altro rispetto ai campioni negli altri cluster.

Come vedremo anche in seguito, il criterio più semplice e usato è quello della *somma degli errori quadrati* (square sum estimate).

Supponiamo che l'insieme dato di n campioni sia stato partizionato in qualche modo in k cluster D_1, D_2, \dots, D_k .

Supponiamo inoltre che n_i sia il numero di campioni in D_i e che m_i sia la media aritmetica dei campioni, ovvero:

$$m_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

Allora, la somma degli errori quadrati sarà uguale a:

$$j_e = \sum_{i=1}^k \sum_{x \in D_i} |x - m_i|^2$$

Per un dato cluster D_i , il vettore delle medie m_i (detti **centroidi**) è la migliore rappresentazione dei campioni nel dataset.

Algoritmi

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 partizione 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 e punta ad unirli 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.

La maggior parte degli algoritmi si basa su partizionamento iterativo ad errore quadratico e clustering gerarchico agglomerativo. L'algoritmo delle **k-medie** è di gran lunga quello a partizionamento iterativo ad errore quadratico più famoso. Funziona secondo i seguenti step:

1. Seleziona 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 inoltre un algoritmo di partizionamento poiché fornisce un'unica partizione degli elementi. È ad errore quadratico poiché mira a minimizzare l'errore rispetto ai centroidi.

L'algoritmo ha una buona efficienza. Fornisce buoni risultati se i cluster sono compatti, ipersferici e ben separati nelle caratteristiche.

Dovremmo però impostare un valore k di cluster a priori: 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 potremmo affidarci agli algoritmi di ricerca.

I **problemi** più comuni di *k-means* sono:

- Una scelta errata del valore k potrebbe portarci a creare più o meno cluster rispetto al numero ideale. Parliamo quindi di risultato sub-ottimale;
- Una scelta errata dei centroidi potrebbe influenzare negativamente il risultato. O potrebbe influenzare negativamente le performance, richiedendo a *k-means* numerose iterazioni per arrivare ad un buon risultato;
- Alcune "forme" non si prestano bene all'utilizzo di questo algoritmo.

Per migliorare le prestazioni di *k-means* in questi casi possiamo aumentare il k consentendo all'algoritmo di adattarsi meglio ai dati.

Inoltre, alcune attività di pre 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. Ma non basta. 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 di cluster "vicini" sono operazioni che aiutano ad ottenere un migliore risultato. Questi passi possono anche essere implementati durante l'esecuzione.

Clustering gerarchico

Clustering gerarchico

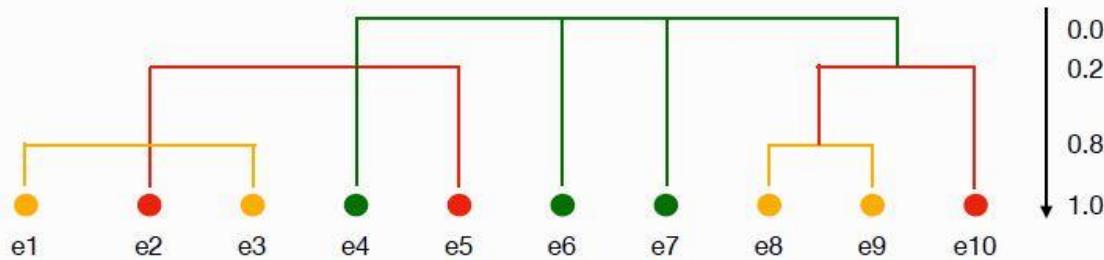
Il clustering gerarchico cerca di considerare raggruppamenti multi-livello (ovvero, a diversi livelli di similarità).

Mentre l'algoritmo k-means restituisce delle partizioni disgiunte, alcuni gruppi di pattern potrebbero avere caratteristiche simili quando osservati ad un certo livello.

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.



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.

La scelta principale nel clustering gerarchico consiste nella misura di distanza tra cluster, che sarà utilizzata per dividere o agglomerare cluster (N.B.: qui si parla di distanza tra cluster e non di distanze metriche tra due elementi). A prescindere dalla misura metrica utilizzata, possiamo determinare la distanza tra cluster in vari modi. Le più note sono le seguenti:

$$d_{min}(D_i, D_j) = \min_{x \in D_i, x' \in D_j} |x - x'| \quad \text{Minima distanza tra due punti nei cluster } D_i \text{ e } D_j.$$

$$d_{max}(D_i, D_j) = \max_{x \in D_i, x' \in D_j} |x - x'| \quad \text{Massima distanza tra due punti nei cluster } D_i \text{ e } D_j.$$

Il clustering gerarchico è un altro metodo molto utilizzato e, rispetto a quanto visto per il *k-means*, non richiede di definire a priori il numero di cluster da ricercare. Ne esistono due versioni:

- **Agglomerativo:** si parte assegnando un cluster diverso per ogni singolo dato in ingresso. Fatto questo, si procede iterativamente ad agglomerare più cluster insieme fino a quando si arriva ad avere un unico cluster che contiene tutti i dati;
- **Divisivo:** è l'esatto contrario dell'agglomerativo. Si parte con un unico cluster che contiene tutti i dati e poi, iterativamente, si procede a suddividere i cluster esistenti in più sottocluster.

In entrambi i casi, quindi, si esplorano tutte le possibili combinazioni di cluster, da un estremo (un unico cluster) all'altro (un cluster per dato) e viceversa.

Una volta svolte tutte queste operazioni, per comprendere al meglio il risultato ottenuto, andiamo a costruire il **dendogramma**. Si tratta di un grafico ad albero dove sull'asse delle ordinate è riportata la “distanza” tra i cluster e sull'asse delle ascisse vengono riportati i vari dati in ingresso.

In questo diagramma, inoltre, le righe verticali corrispondono ad un cluster, quelle orizzontali ad operazioni di unione o di divisione (a seconda se si utilizza la versione agglomerativa o divisiva).

Quindi il dendogramma rappresenta una sorta di “memoria” dei cluster e visualizza come questi cambiano in funzione della distanza massima che vogliamo applicare ai nostri dati, distanza che funzionerà come una soglia.

Density-based clustering

Density-based clustering

Il clustering basato sulla densità raggruppa pattern considerando la loro densità nella distribuzione.

DBSCAN



<https://github.com/NSHipster/DBSCAN>

k-means



Questo tipo di clustering andrà a risolvere il problema delle “forme” che abbiamo visto con k-means

Non entreremo troppo nel dettaglio di questi algoritmi, ma basta sapere che si basano su due parametri per stabilire il criterio di raggruppamento dei cluster.

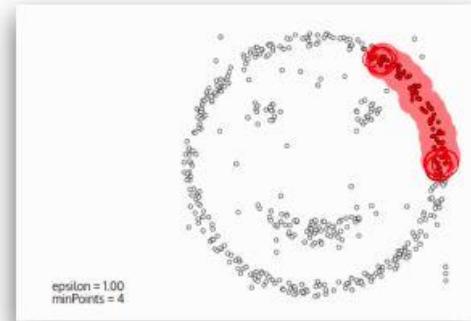
Il parametro *minPts*, che stabilisce il numero minimo di punti per considerare un intorno di un punto denso.

Il parametro ϵ , che definisce un intorno circolare di ciascun punto dai suoi vicini.

Aumentando o riducendo il valore di questi parametri,
potremo migliorare la qualità del clustering.

La cosa importante da capire è che questo tipo di clustering è in grado di scoprire forme arbitrarie.

L'algoritmo più noto basato sulla densità è chiamato DBSCAN.



Valutazione dei risultati

Il problema degli algoritmi per la risoluzione di problemi di clustering è che sono 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 le tante a disposizione, vengono generalmente utilizzate le seguenti tre metriche:

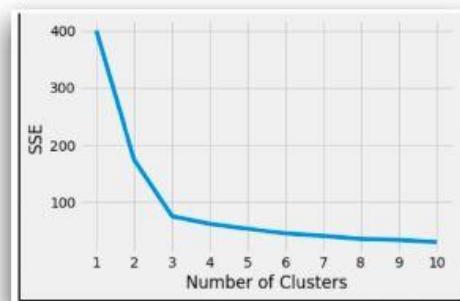
- **Elbow point**, punto di gomito: spesso utilizzato per valutare il numero migliore di cluster da generare negli algoritmi di clustering partizionale, ad esempio *k-means*;
- **Silhouette coefficient**, coefficiente di forma: misura la consistenza dei cluster, andando a misurare quanto simili sono gli elementi che compongono un singolo cluster;
- **MoJo distance**: può essere calcolata solo se abbiamo a disposizione delle etichette per poter valutare la bontà del cluster. In alcuni casi, le etichette vengono ignorate in fase di costruzione del modello per poi essere usate in fase di validazione.

Problemi di raggruppamento: Valutazione dei risultati - Elbow 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 ché 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 $k=4$.

Problemi di raggruppamento: Valutazione dei risultati - Silhouette coefficient

Il silhouette coefficient è una misura della coesione e separazione tra i dati. Più in particolare, quantifica quanto i dati siano ben disposti nei cluster generati.

Il coefficiente si basa su due parametri: (1) Quanto bene i dati sono *ammassati* nel cluster di riferimento; (2) 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:

$$s(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))}$$

con:

- $a(i)$ che rappresenta la distanza media dell' i -esimo punto rispetto a tutti gli altri punti appartenenti allo stesso cluster;
- $b(i)$ che rappresenta la distanza media dell' i -esimo punto rispetto a tutti gli altri punti appartenenti al cluster più vicino del cluster a cui è stato assegnato;
- $s(i)$ che rappresenta il coefficiente di silhouette dell' i -esimo punto.

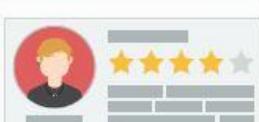
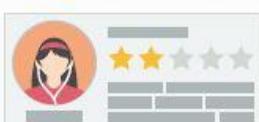
Il valore finale di silhouette è dato dalla media dei coefficienti di silhouette calcolati per ogni elemento del problema.

Problemi di raggruppamento: Valutazione dei risultati - MoJo distance

La Move-Join (MoJo) distance è una metrica che calcola il numero minimo di operazioni di spostamento e raggruppamento di elementi sono necessari per passare dal clustering identificato da un algoritmo al clustering ideale degli elementi.

Qualcuno potrebbe chiedersi: *Ma se ho a disposizione un oracolo che riporta il raggruppamento ideale, perché dovrei fare clustering con un algoritmo?*

La risposta è semplice: per valutare un algoritmo di clustering prima di utilizzarlo su nuovi dati sconosciuti! Facciamo un esempio.



Conoscete tutti il meccanismo delle user review, che consente agli utenti di applicazioni mobile di esprimere opinioni su un'app, far notare agli sviluppatori eventuali problemi e/o funzionalità mancanti che dovrebbero essere implementate.

Applicazioni popolari, come Instagram, Whatsapp o TikTok, ricevono migliaia di review al giorno e tenere traccia di quello che succede è difficile, se non impossibile.

Tuttavia, sebbene il numero di review sia enorme, è probabile che gli utenti facciano notare cose simili! Ad esempio, un bug può essere notato da più utenti.

Quindi, possiamo pensare di usare una tecnica di clustering!

Ma, esattamente, clustering di cosa? E come?

Dovremmo sicuramente estrarre tutte le user review dell'app di interesse (o una parte, magari le più recenti o quelle relative all'ultima versione rilasciata) e provare a raggrupparle in base ad una misura di somiglianza testuale.

Quindi, come metrica di somiglianza potremmo usare, ad esempio, la distanza di Jaccard o quella di Hamming.

A quel punto, dovremmo ottimizzare la generazione dei cluster. Per farlo, potremmo usare diversi algoritmi di clustering (k-medoids o altri algoritmi più specializzati nel raggruppamento di stringhe e testi scritti in linguaggio naturale).

Non sappiamo però quale di questi algoritmi è più efficace a risolvere un problema di questo tipo - consideriamo che, parlando di user review, parliamo di testi rumorosi che sono naturalmente difficili da raggruppare.

Per confrontare i vari algoritmi potremmo quindi provare la seguente idea. Facciamo un sacrificio ed estraiamo le user review della release R_i , raggruppiamole manualmente formando dei cluster semanticamente validi e valutiamo quanto i vari algoritmi di clustering sperimentati si avvicinano al nostro operato.

Una volta stabilito il miglior algoritmo potremo poi assumere che, per nuove release dell'app, possiamo affidarci al risultato per prendere le appropriate decisioni.

In questo senso, la MoJo distance ci fornisce una misura molto più precisa della bontà dei cluster, in quanto si basa su dati reali (l'oracolo).

Più in generale, una valutazione fatta tramite il calcolo della MoJo distance ci dà maggiore confidenza nell'utilizzare i risultati del clustering. Inoltre, può più facilmente abilitare l'utilizzo del clustering come mezzo per altre operazioni.

Ad esempio, ChangeAdvisor.

