Indice

Capitolo 1
Introduzione

1.1 Agente intelligente

Si definisce **agente intelligente**, o semplicemente **agente**, qualsiasi entitá in grado di percepire l'ambiente in cui si trova mediante sensori e modificando tale ambiente compiendo delle azioni, mappando percezioni ad azioni. Con **ambiente** si intende la parte di universo a disposizione delle percezioni dell'agente e da questa influenzabile. L'intelligenza artificiale é definibile come lo studio degli agenti.

> Un essere umano puó essere modellato come un agente, potendo percepire l'ambiente tramite occhi, orecchie e altri organi e agendo su di esso per mezzo dei suoi arti. Allo stesso modo, un robot puó essere modellato come un agente, percependo l'ambiente attraverso telecamere o sensori infrarossi e agendo su di esso mediante appendici e/o motori elettrici. Infine, anche un programma per computer puó essere modellato come un agente, se si considera l'input umano (tramite tastiera, mouse, touchscreen o voce) come percezione ed il suo output (scrivere su un file, mostrare un contenuto a schermo, generare un suono, eccetera) come azione compiuta sull'ambiente.

La sequenza di percezioni di un agente é la storia completa di tutto ció che l'agente ha percepito. In generale, la scelta dell'azione compiuta da un agente in un certo istante dipende dalla sua conoscenza a priori e/o dall'intera sequenza di percezioni precedente. Formalmente, il comportamento di un agente é descritto da una funzione agente che mappa sequenze di percezioni in azioni: $f : \mathrm{Pow}(P) \to A$. Tale funzione é un concetto astratto, una caratterizzazione *esterna* di un agente: *internamente*, la funzione agente di un agente intelligente é implementata da un **programma agente**; tale funzione viene eseguita da un dispositivo elettronico dotato di sensori di sorta, chiamato **architettura**.

Un **agente razionale** é un agente che "fa la scelta giusta". La nozione di "scelta giusta" comunemente adottata nel campo dell'intelligenza artificiale é il **consequenzialismo**: il comportamento dell'agente é valutato sulla base delle conseguenze delle sue azioni. Se un agente, in relazione ad un certa percezione, compie una azione desiderabile dal punto di vista dell'utilizzatore, allora tale agente ha compiuto la "scelta giusta", ed é definibile agente razionale. La nozione di desiderabilitá viene descritta da una **misura di prestazione** che valuta ogni sequenza di stati in cui l'ambiente si trova. In genere, é preferibile definire una misura di prestazione rispetto a ció che si vuole accada all'ambiente piuttosto che rispetto al modo in cui ci si aspetta che funzioni.

É allora possibile fornire una definizione operativa di agente razionale: per ogni possibile sequenza di percezioni, un agente razionale sceglierá di compiere l'azione che, sulla base delle percezioni precedenti e sulla base della conoscenza che possiede a priori, restituisce il massimo valore possibile in termini di misura di prestazione. Si noti come "razionale" non significhi "onniscente", ovvero in grado di prevedere con assoluta certezza ció che accadrá in futuro, dato che questo é realisticamente impossibile; un agente razionale deve limitarsi a compiere azioni che massimizzano la prestazione *attesa*.

La definizione di agente razionale sopra presentata prevede che questo possieda anche una qualche nozione di **apprendimento**: per quanto la sua configurazione iniziale possa essere fissata, questa puó venire modificata e potenziata con l'esperienza. Nel caso in cui l'ambiente sia interamente conosciuto a priori, l'agente non ha alcuna forma di apprendimento, limitandosi a compiere le azioni preimpostate.

Un agente che compie azioni esclusivamente sulla base della sua conoscenza a priori e non fa uso di apprendimento si dice che non é **autonomo**. Un agente razionale dovrebbe invece essere autonomo, ovvero partire sí da una base di conoscenza pregressa ma, attraverso l'apprendimento, colmarne le lacune. Dopo abbastanza esperienza, ci si aspetta che un agente razionale diventi di fatto indipendente dalla sua conoscenza a priori.

É possibile classificare gli ambienti rispetto a cinque metriche informali, utili a ragionare sulla difficoltá del problema e sulla modalitá risolutiva da adottare:

- **Accessibile** o **inaccessibile**. Un ambiente é tanto accessibile quanto un agente é in grado di ottenere le informazioni sul suo stato di cui necessita con completa accuratezza. Un ambiente puó essere inaccessibile perché i sensori dell'agente non sono precisi oppure perché parte dell'ambiente é del tutto preclusa ai sensori dell'agente. Gli ambienti nel mondo reale hanno necessariamente un certo grado di inaccessibilitá;
- **Deterministico** o **non deterministico**. Un ambiente é deterministico (in riferimento alle azioni dell'agente) se la sua evoluzione é completamente determinata dal suo stato attuale e dalle azioni dell'agente. Un ambiente é non deterministico se la sua evoluzione é anche influenzata da forze al di lá dell'agente. Il mondo fisico da modellare ha sempre un certo grado di non determinismo;
- **Episodico** o **sequenziale**. In un ambiente episodico l'esperienza di un agente puó essere divisa in step atomici dove la scelta di un azione dipende esclusivamente dalla percezione attuale. In un ambiente sequenziale le azioni che un agente compie possono dipendere del tutto o in parte da quali azioni sono state prese in precedenza;
- **Statico** o **dinamico**. Un ambiente é statico se non subisce modifiche mentre l'agente sta deliberando, altrimenti é dinamico;
- **Discreto** o **continuo**. Un ambiente é discreto se il numero di stati in cui questo puó trovarsi é finito, ovvero se é possibile (almeno in linea teorica) enumerare tutti i suoi possibili stati, altrimenti é continuo. Essendo i computer discreti per definizione, modellare un ambiente continuo attraverso un sistema automatico richiederá sempre un certo grado di approssimazione.

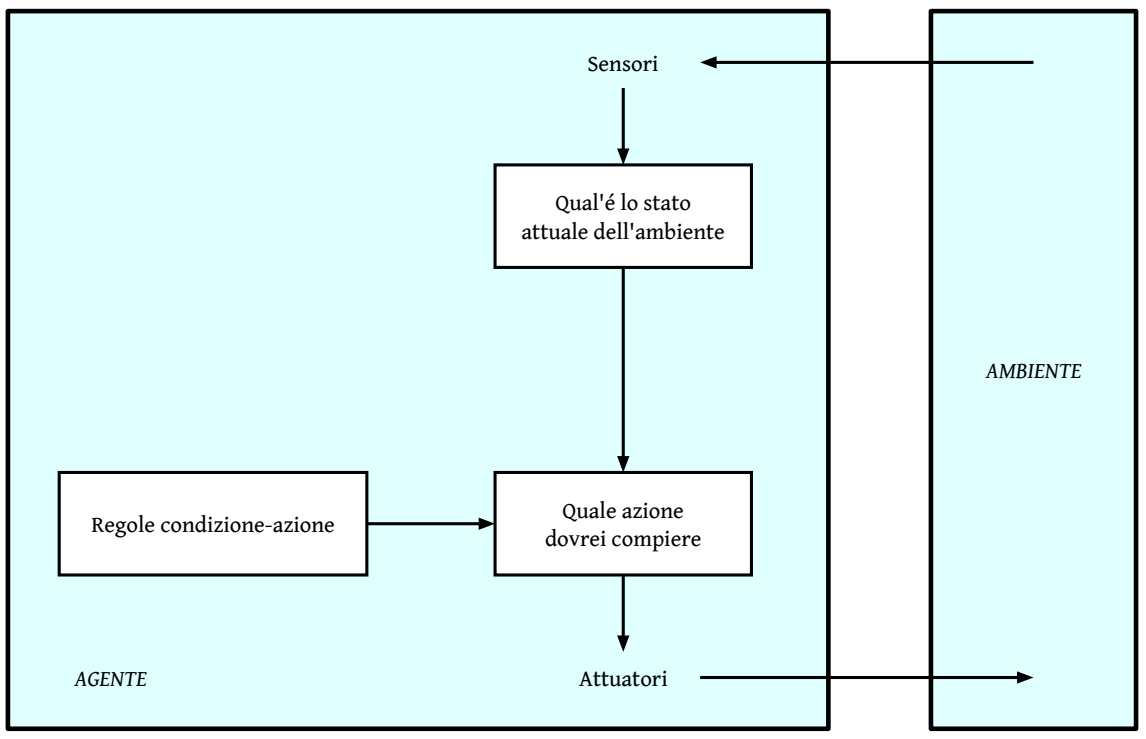
- Si consideri come ambiente il gioco degli scacchi e come agenti i giocatori umani (si assuma che le mosse non abbiano alcun limite di tempo). Tale ambiente é:
  1. Accessibile, perché ciascun giocatore ha completa conoscenza dello stato della partita;
  2. Deterministico, perché l'evoluzione degli stati dipende esclusivamente da quali mosse scelgono di compiere i giocatori;
  3. Sequenziale, perché le mosse di un giocatore possono anche dipendere da quali mosse ha compiuto in precedenza;
  4. Statico, perché durante l'esecuzione di una mossa e durante la scelta della stessa lo stato della partita rimane invariato;
  5. Discreto, perché il numero di possibili stati in cui la partita puó trovarsi é finito.
- Si consideri come ambiente le strade di una cittá e come agente un sistema di guida automatico per automobili. Tale ambiente é:
  1. Inaccessibile, perché non é possibile conoscere l'intero stato del traffico di tutta la cittá in ciascun istante;
  2. Non deterministico, perché l'evoluzione del traffico non dipende esclusivamente dalle scelte dell'agente;
  3. Sequenziale, perché la scelta di quale strada percorrere puó dipendere anche da quali strade ha percorso in precedenza;
  4. Dinamico, perché lo stato della cittá e del traffico cambiano anche mentre l'agente é in movimento;
  5. Continuo, perché lo stato della cittá e del traffico si modificano costantemente.

Gli agenti intelligenti possono essere informalmente classificati in quattro categorie, di crescente ordine di complessitá.

1.1.1 Agenti con riflessi semplici

Gli agenti piú facili da realizzare sono gli **agenti con riflessi semplici.** Questi agenti non hanno alcun modello dell'ambiente: scelgono che azione compiere esclusivamente sulla base della percezione attuale e non hanno cognizione delle percezioni precedenti.
Agenti di questo tipo scelgono che azioni compiere seguendo **regole condizione-azione**: *se* si verifica una certa condizione, *allora* viene compiuta l'azione associata a tale condizione.
Una rappresentazione schematica di un agente con riflessi semplici é presentata in basso. La funzione `INTERPRET-INPUT` genera una descrizione astratta della percezione ricevuta dall'agente, mentre la funzione `RULE-MATCH` restituisce la prima azione associata a tale rappresentazione di percezione nel set di regole `rules` .

```
rules <= set of condition-action rules

function SIMPLE-REFLEX-AGENT(percept)
 state <= INTERPRETER-INPUT(percept)
 rule <= RULE-MATCH(state, rules)
 action <= rule.action
 return action
```



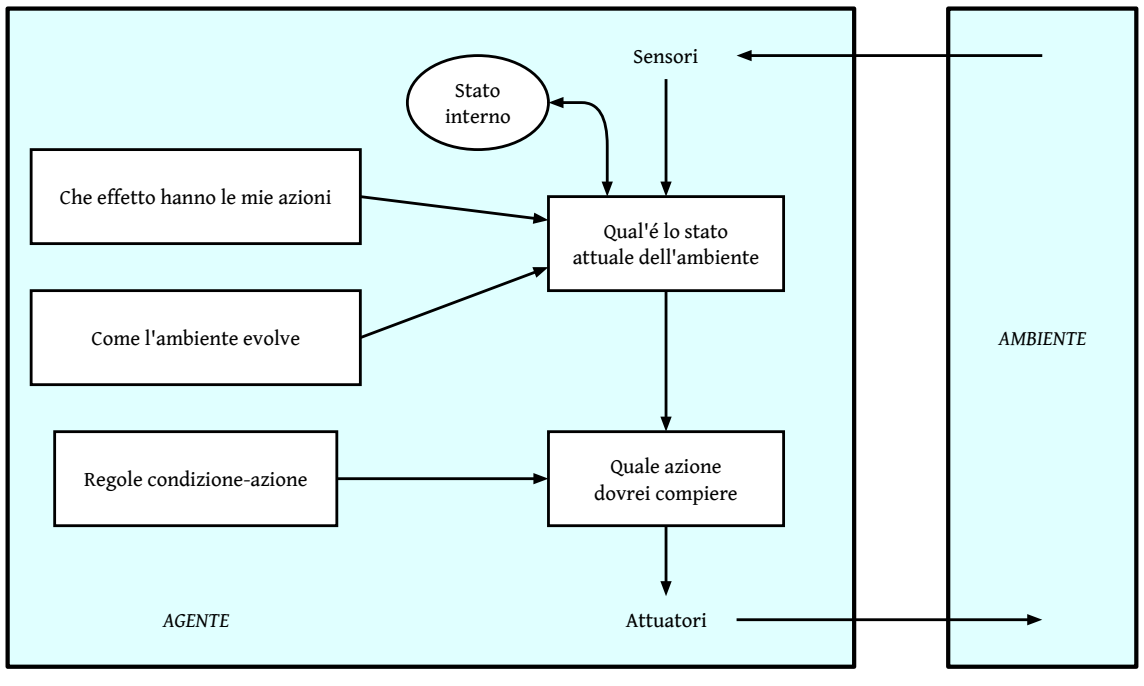
Gli agenti con riflessi semplici hanno una intelligenza limitata. Infatti, agenti di questo tipo operano correttamente solamente se l'azione da compiere che massimizza la funzione di prestazione puó essere determinata solo sulla base delle proprie percezioni, ovvero se l'ambiente é completamente accessibile. Se nella propria conoscenza a priori sono presenti errori o se l'ambiente é accessibile solo in parte, l'agente sará destinato ad operare in maniera non razionale.
Ancora piú problematica é la situazione in cui agenti con riflessi semplici entrano in loop infiniti, dato che non sono in grado di determinarli. L'unica contromisura che possono adottare é randomizzare le proprie azioni, dato che in questo modo si riduce la probabilitá che l'agente compia

le stesse azioni piú volte di fila. Tuttavia, sebbene questo approccio possa mettere una pezza al problema del loop infinito in maniera semplice, in genere comporta uno spreco di risorse, e pertanto risulta difficilmente in un comportamento razionale da parte dell'agente.

1.1.2 Agenti con riflessi, ma basati su un modello

Il modo piú efficiente per risolvere il problema dell'avere a che fare con un agente parzialmente accessibile é tenere traccia della parte di ambiente di cui questo non ha conoscenza. Ovvero, l'agente dovrebbe avere una qualche sorta di **stato interno** che dipende dalle percezioni che questo ha captato in precedenza, di modo da avere informazioni su alcuni degli stati diversi da quello corrente. Agenti di questo tipo sono detti **agenti con riflessi ma basati su un modello**.
Aggiornare periodicamente tale stato interno richiede che l'agente possieda due forme di conoscenza. Innanzitutto, é necessario avere informazioni relative al modo in cui l'ambiente si evolve nel tempo, sia in termini di come le azioni dell'agente influenzano l'ambiente che in termini di come l'ambiente si evolve in maniera indipendente dall'agente. Questo corpo di informazioni prende il nome di **modello di transizione**. Inoltre, é necessario avere informazioni relative a come l'evoluzione dell'ambiente si riflette sulle percezioni dell'agente, nel complesso chiamate **modello sensoriale**.
Una rappresentazione schematica di un agente con riflessi ma basati su un modello é presentata in basso, dove la funzione UPDATE-STATE aggiorna lo stato interno dell'agente prima di restituire l'azione da compiere.

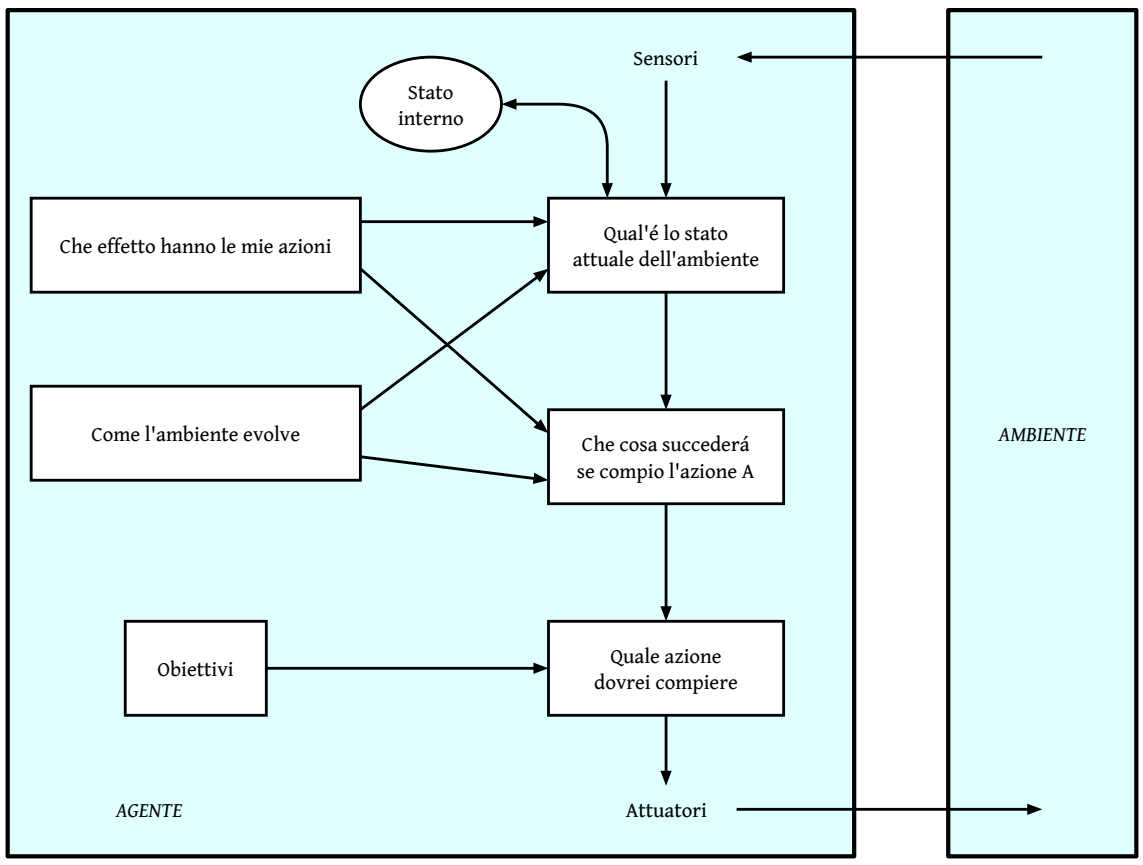
```
state <= the agent's current conception of the environment state
transition_model <= a description on how the next state depends on the current state and action
sensor_model <= a description on how the current world state is reflected in the agent's percepts
rules <= set of condition-action rules
action <= the most recent action (starts NULL)


function MODEL-BASED-REFLEX-AGENT(percept)
 state <= UPDATE-STATE(state, action, percept, transition_model, sensor_model)
 rule <= RULE-MATCH(state, rules)
 action <= rule.action
 return action
```



Si noti come difficilmente un agente con riflesso basato su un modello puó determinare con certezza lo stato attuale dell'ambiente. In genere, un agente puó limitarsi ad averne una descrizione parziale.

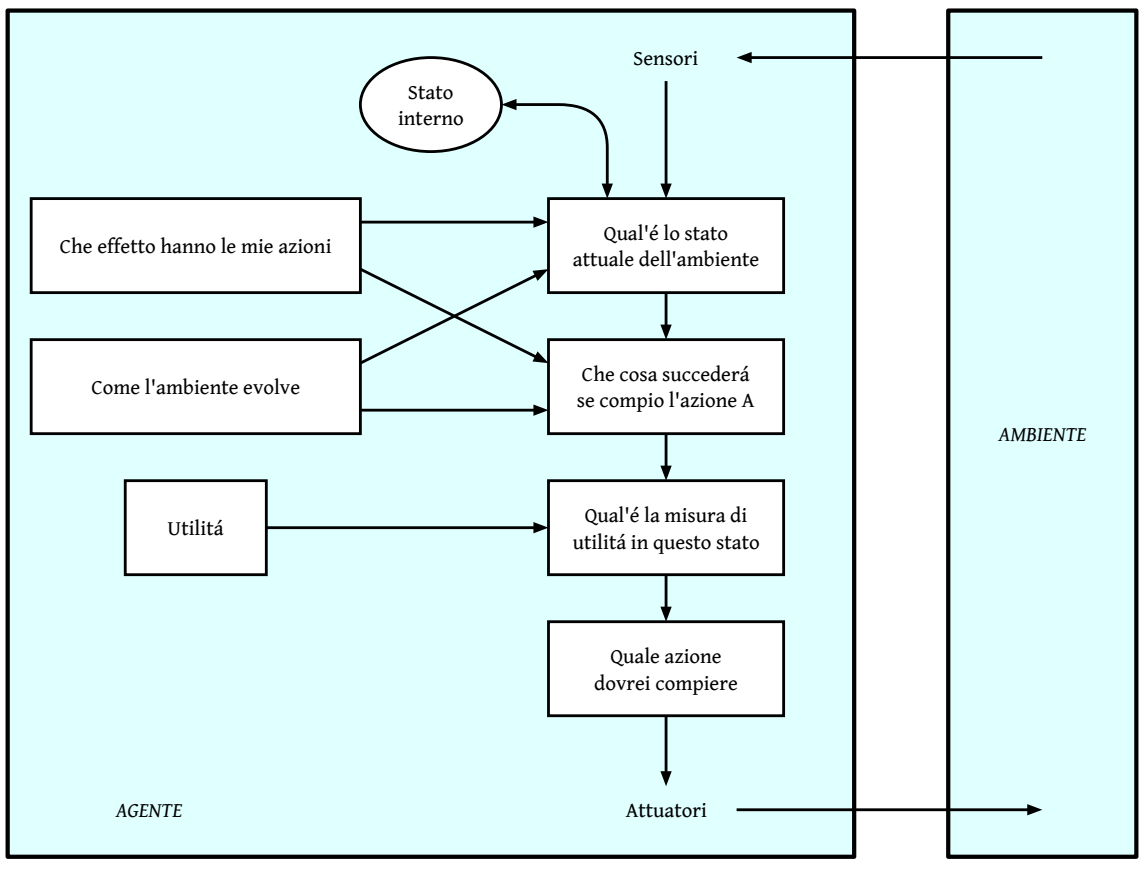
1.1.3 Agenti basati su un modello, ma basati su obiettivi

Vi sono situazioni in cui la scelta di quale sia l'azione migliore da compiere da parte di un agente dipenda anche da un qualche tipo di obiettivo a lungo termine. Non sempre questo obiettivo viene raggiunto nell'operare una sola azione, ma puó richiedere diverse azioni intermedie. In agenti di questo tipo, la medesima azione ed il medesimo stato interno possono risultare in azioni diverse se é diverso l'obiettivo.



1.1.4 Agenti basati su un modello e guidati da utilitá

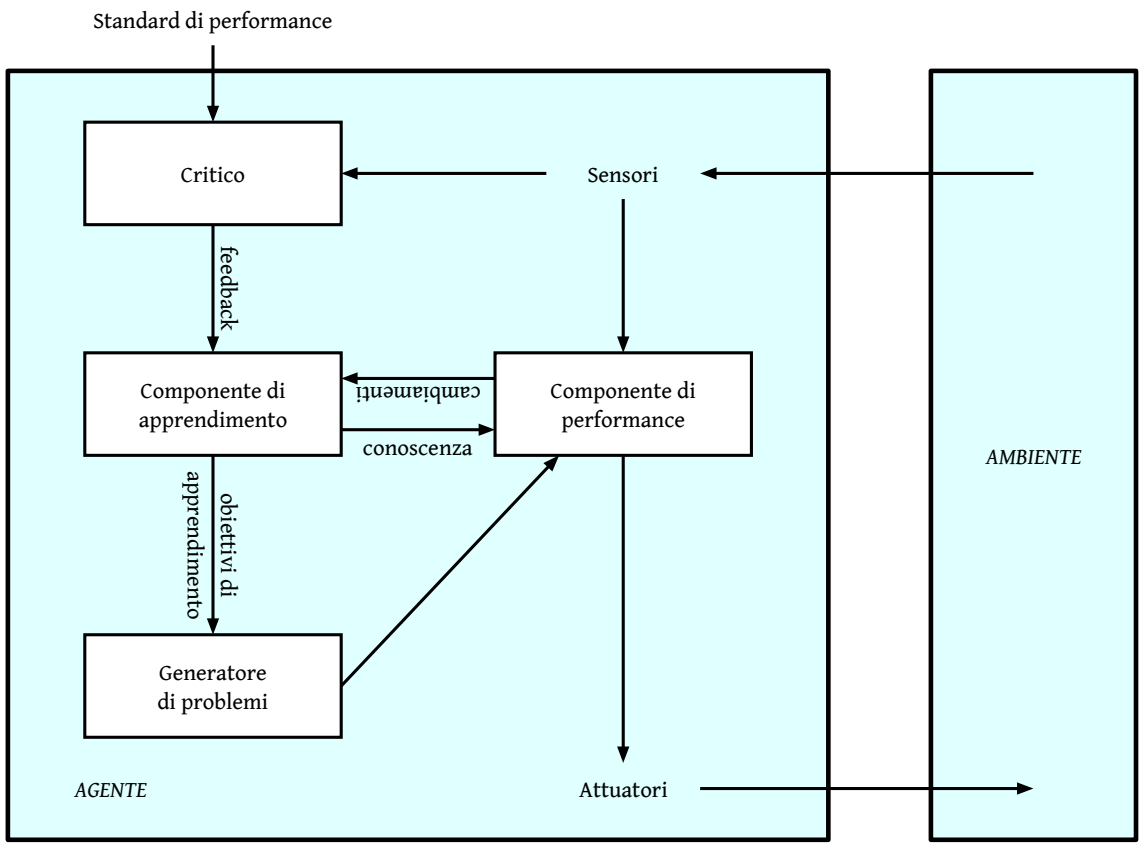
Non sempre é possibile costruire un agente razionale semplicemente spingendolo a raggiungere un obiettivo. Infatti, se tale obiettivo puó essere raggiunto tramite diverse sequenze di azioni, una potrebbe essere preferibile ad un'altra. Inoltre, un agente potrebbe dover perseguire piú obiettivi contemporaneamente fra di loro incompatibili, ovvero compiere azioni che lo "avvicinano" ad un obiettivo ma al contempo "allontanarlo" da un altro.
Un obiettivo permette di discriminare gli stati dell'ambiente esclusivamente come "favorevoli" e "sfavorevoli", senza alcuna sfumatura nel mezzo. Un migliore approccio prevede invece di introdurre una misura di **utilitá**, che influenza la scelta dell'agente nello scegliere quale azione compiere (insieme alla misura di prestazione, all'obiettivo da seguire e dal proprio stato interno).
La misura di utilitá permette all'agente di, nel dover perseguire piú obiettivi fra di loro incompatibili, scegliere l'azione che comporta il miglior compromesso nell'avanzamento di tutti loro. Inoltre, non sempre la struttura dell'ambiente garantisce che sia possibile raggiungere con assoluta certezza un obiettivo semplicemente eseguendo le azioni appropriate; anche in questo caso, la misura di utilitá permette di valutare quanto sia "conveniente" per l'agente compiere una certa azione in vista di un determinato obiettivo sulla base di quanto sia ragionevole che tale obiettivo venga effettivamente raggiunto.



1.1.5 Agenti che apprendono

Gli agenti piú interessanti sono indubbiamente quelli in grado di **apprendere**; tutti i tipi di agenti presentati finora possono essere costruiti come agenti che apprendono. Il notevole vantaggio che presentano é che possono operare in un ambiente del tutto sconosciuto apprendendo da questo, di modo da compiere le azioni migliori anche in situazioni dove lo stesso designer non ha modo di poter prevedere quali queste possano essere.
Un agente in grado di apprendere puó essere concettualmente suddiviso in quattro componenti:

- La **componente di apprendimento**, che si occupa di migliorare la performance dell'agente;
- La **componente di performance**, che sceglie quale azione compiere sulla base delle percezioni e dello stato di conoscenza interno. Di fatto, questa componente costituiva l'intero agente dei modelli precedenti;
- Il **critico**, che informa la componente di apprendimento di quanto l'agente si sta comportando in maniera ottimale (razionale) sulla base di uno standard di performance prestabilito. Questa componente é necessaria perché le percezioni, di per loro, non sono in grado di informare l'agente sull'ottimalitá del proprio comportamento;
- Il **generatore di problemi**, che suggerisce azioni all'agente che possono comportare nuove ed informative esperienze. Questa componente é necessaria perché se l'agente si affidasse esclusivamente alla componente di performance sceglierebbe sempre le azioni migliori sulla base della sua conoscenza attuale, che non sono necessariamente complete. Il generatore di problemi puó portare l'agente a compiere azioni che possono potenzialmente essere localmente subottimali ma che sul lungo termine possono portare a compiere azioni ancora migliori.



Si dice che un agente compie un **apprendimento** se migliora le proprie prestazioni dopo aver compiuto delle osservazioni sull'ambiente. Quando l'agente in questione é un computer, si parla di **machine learning**: il computer ricava dei dati, costruisce un modello sulla base di questi ultimi ed utilizza tale modello sia come ipotesi sul mondo che come software in grado di risolvere problemi.
La programmazione tradizionale prevede essenzialmente di descrivere delle regole che, fornite ad un computer, risolvono un problema. Questo presuppone che il programmatore sappia *giá*, in una qualche misura, come risolverlo. Nel machine learning, il programmatore stabilisce un modo per produrre dati che addestri un algoritmo di apprendimento a descrivere tali regole in maniera automatica al suo posto.
I motivi per investire su un agente in grado di apprendere sono fondamentalmente due. Il primo é che il designer non é in grado di anticipare ogni possibile situazione futura, ed é quindi necessario che sia l'agente stesso (eventualmente guidato) a prendersene carico. Il secondo é che alcuni problemi sono cosí complessi che nemmeno il designer é in grado di determinare come risolverli, eppure sufficientemente approcciabili da poter fornire gli strumenti all'agente per poterlo fare.
Si noti come, nella maggior parte dei casi, un algoritmo di machine learning non prevede che questo debba costantemente apprendere, cosí come non debba necessariamente apprendere piú di una volta. Nonostante esistano alcuni algoritmi di machine learning basati sull'apprendimento continuo, in genere questi apprendono a partire da uno o piú dataset e mantengono indefinitamente la conoscenza acquisita. Similmente, le prestazioni di un algoritmo di machine learning non necessariamente migliorano aggiungendo nuovi dati alla sua conoscenza. Tali dati potrebbero infatti non fare altro che "confondere" l'immagine del mondo che l'algoritmo si é fatto. Cio non toglie che possano esserci situazioni in cui il modello che l'agente ha appreso debba venire aggiornato, ovvero sostituito con un nuovo modello che meglio predice lo scopo per cui l'agente sta apprendendo.

Nello specifico, esistono due metodi per far apprendere ad algoritmo di machine learning: **batch learning** e **online learning**. Nel primo, l'algoritmo viene addestrato a partire dall'intero dataset, mentre nel secondo l'algoritmo viene addestrato fornendogli i dati sequenzialmente, in maniera individuale o in piccoli gruppi detti mini-batch. L'online learning permette di addestrare un algoritmo con un ammontare di dati che, se considerati tutti in una sola volta, richiederebbero risorse computazionali proibitive, ma richiede maggiore attenzione perché le informazioni apprese in ciascun istante possono influenzare ció che viene appreso negli istanti successivi. A tale scopo é necessario definire un **learning rate**, ovvero quanto rapidamente debba l'algoritmo adattarsi alle nuove informazioni che vengono introdotte.

Capitolo 2
Rappresentazione della conoscenza

2.1 Knowledge representation and reasoning

Gli esseri umani sono in grado di compiere azioni anche sulla base del fatto che possiedono delle **conoscenze** utilizzate per operare dei **ragionamenti** su una **rappresentazione** interna della conoscenza. Nel campo della AI questo si traduce nella costruzione di **agenti basati sulla conoscenza**.
Il componente principale di un agente basato sulla conoscenza é la **base di conoscenza**, o KB. Una KB é composta da un insieme di **fatti**, che rappresentano delle asserzioni sul mondo. Un agente basato sulla conoscenza deve essere in grado di fare **inferenze**, ovvero essere in grado di aggiungere dei nuovi fatti alla KB sulla base di quelli presenti applicando delle **regole**. Affinché questo sia possibile, é necessario che alcuni fatti siano presenti nella KB fin da subito. Questi vengono detti **assiomi**; l'unione di tutti gli assiomi prende il nome di **conoscenza pregressa** (**background knowledge**).
Sia i fatti (le asserzioni sul mondo) che le regole (le trasformazioni che aggiungono nuovi fatti alla KB sulla base di quelli presenti) vengono espressi in genere espressi in linguaggi specifici. Tali linguaggi sono detti **linguaggi di Knowledge Representation and Reasoning**, o **linguaggi KRR** (**linguaggi di rappresentazione della conoscenza**). Un linguaggio KRR deve necessariamente basarsi su una qualche formalizzazione della logica, e ci si chiede allora quale formalizzazione della logica potrebbe ben adattarsi ad essere quella utilizzata dagli agenti basati sulla conoscenza.
La logica proposizionale (logica di ordine zero) puó venire scartata subito: nonostante abbia il pregio di essere decidibile, é troppo semplicistica, dato che non supporta i quantificatori universali "per ogni" e "esiste". Un miglior candidato potrebbe allora essere la logica proposizionale (logica del primo ordine), ma anche questa presenta dei problemi:

- *Decidibilitá*. Come mostrato dai Teoremi di Incompletezza di Godel, la logica proposizionale é **indecidibile**, ovvero non tutte le formule possono essere provate vere o false all'interno della logica stessa [1]. Questo significa che un sistema di deduzione automatico, essendo limitato dall'Halting Problem, potrebbe rimanere eternamente bloccato nel computare se una data proposizione segua dalle premesse senza essere in grado di fornire una risposta;
- *Complessitá*. La logica proposizionale é estremamente espressiva, pertanto alcune inferenze possono richiedere molto tempo computazionale (per quanto finito) per essere completate;
- *Approssimazione*. Per lo stesso motivo, non tutte le proprietá della logica proposizionale sono strettamente necessarie nel campo della IA. Cercare di implementarle tutte risulterebbe in uno spreco di risorse e nella costruzione di un sistema di deduzione inefficiente.

La scelta di un formalismo logico adatto al campo delle IA sembrerebbe allora ricadere in una logica che si trovi "nel mezzo" fra la logica proposizionale e la logica predicativa.

2.2 Knowledge Graphs

Un **Knowledge Graph** (**KG**) é un grafo diretto ed etichettato il cui scopo é riportare e trasmettere conoscenze sul mondo reale. I nodi del grafo rappresentano delle **entitá**, ovvero degli oggetti che appartengono al mondo di interesse, mentre gli archi del grafo rappresentano delle **relazioni** che intercorrono fra queste entitá.
Con "conoscenza" si intende genericamente qualsiasi cosa sia *nota*: tale conoscenza puó essere ricavata da dal mondo che il grafo vuole modellare oppure estratta dal grafo stesso. La conoscenza puó essere composta sia da semplici asserzioni che coinvolgono due entitá ("A possiede/fa uso di/fa parte di/... B") oppure asserzioni che coinvolgono gruppi di entitá ("tutti i membri di A possiedono/fanno uso/fanno parte di/... B"). Le asserzioni semplici sono riportate come etichette degli archi del grafo: se esiste un arco fra i nodi A e B, significa che A e B sono legati dalla relazione che etichetta l'arco che li unisce.
Formalmente, un Knowledge Graph é definito a partire dalla quintupla $\langle E, L, T, P, A \rangle$:

- Un insieme E di simboli, che rappresentano gli identificativi associati alle entitá;
- Un insieme L di **letterali**, che rappresentano tutti i dati "grezzi" che il modello necessita di rappresentare (stringhe, numeri, eccettera);
- Un insieme T di tipi;
- Un insieme P di simboli di relazione;
- Un insieme A di assiomi.

A loro volta, gli assiomi vengono distinti in due sottogruppi:

- I fatti, ovvero assiomi che riguardano le singole entita. Indicano:
  - Se una certa entitá appartiene ad un certo tipo, ovvero $t(e) \mid t(l)$ con $e \in E$ e $l \in L$;
  - Se due entitá sono legate da una certa relazione, ovvero $r(e_1, e_2) \mid r(e, l)$ con $e_i \in E$ e $l \in L$.

1. Piú correttamente, si dice che la logica proposizionale é **semidecidibile**, in quanto é sempre possibile dimostrare se una proposizione é vera sulla base delle premesse ma non é sempre possibile dimostrare se sia falsa.

- Gli assiomi generali, ovvero assiomi che non riguardano singole entitá ma riguardano classi. La loro espressivitá dipende dal linguaggio logico a cui il KG fa riferimento, ma in genere sono nella forma $\forall x(t_1(x) \rightarrow t_2(x))$, ovvero che specificano una relazione di ordine parziale rispetto ai tipi.

Nei modelli di database relazionale, i dati sono rigidamente strutturati; la struttura é data dallo schema del database (che definisce le relazioni, le entitá, gli attributi, ecc ...). I dati e lo schema sono *fortemente accoppiati*, dato che lo schema deve necessariamente venire definito prima di poter inserire i dati. Inoltre, lo schema é prescrittivo, dato che i dati non conformi allo schema non possono venire inseriti nel database.
Nei modelli di database a grafo, i dati sono parzialmente strutturati, dato che lo schema "emerge" in maniera implicita dal modo in cui sono scritte le triple. I dati e lo schema sono *debolmente accoppiati*, dato che i dati possono venire inseriti prima ancora di definire lo schema [2]. Inoltre, lo schema non é prescrittivo, dato che i dati non conformi alla forma attuale dello schema possono venire inseriti comunque (e modificano lo schema).
Lo schema di un grafo RDF puó essere visto sotto due aspetti. Il primo aspetto é lo schema come "patto sociale", dove i costruttori di grafi si impegnano a seguire degli standard (non obbligatori) per fare in modo che diversi grafi siano fra loro compatibili. Il secondo aspetto é lo schema é uno schema deduttivo, dato che fornisce solamente il significato dei termini e permette di fare inferenze (anche false).
Un primo approccio al fare in modo che i grafi siano compatibili é quello di costruire dei vocabolari standard che vengono impiegati per modellare domini diversi. Questo approccio funziona se esistono degli enti autorevoli che forniscono tali vocabolari; fra questi figurano **FOAF** (**friend of a friend**) e schema.org .
Modellare i dati sotto forma di grafo offre maggior flessibilità per integrare nuovi dataset rispetto ai modelli relazionali standard, dove uno schema deve essere definito prima che i dati possano essere inseriti. Nonostante anche modelli di dato ad albero (XML, JSON, ecc ...) offrano questa flessibilità, i modelli a grafo non necessitano di dover organizzare i dati in una gerarchia. Inoltre, i modelli a grafo permettono facilmente di rappresentare relazioni cicliche.
Essendo un KG un grafo, é possibile studiarne le proprietá tipiche dei grafi (simmetria, antisimmetria, transitivitá, eccetera) e metterle in relazione con il significato che hanno nel modello che questi rappresentano. É inoltre possibile *visitare* il grafo per ricavare informazioni piú elaborate di quelle riportate nei soli archi.

2.3 Resource Description Framework

Resource Description Framework (**RDF**) é un esempio di modello di dati a grafo; sebbene inizialmente concepito per il web (é infatti parte di un insieme di protocolli piú grande noto come **Semantic Web Stack**), trova uso anche come formato per la rappresentazione della conoscenza.

2.4 Termini

RDF é un modello di dati pensato per descrivere risorse. Con **risorsa** si intende qualsiasi entitá a cui sia possibile associare un'identitá, che siano entitá virtuali (pagine web, siti web, file, ...), entitá concrete (libri, persone, luoghi, ...) o entitá astratte (specie animali, categorie, ere geologiche, ...). Ad una risorsa RDF viene fatto riferimento attraverso un **termine**; RDF ammette l'esistenza di tre tipi di termini: **IRI**, **letterali** e **nodi blank**.
Un IRI (**International Resource Identifier**) é una stringa di caratteri Unicode che identifica univocamente una qualsiasi risorsa; se due risorse hanno lo stesso IRI, allora sono in realtá la stessa risorsa. Gli IRI sono un superset degli **URI** (**Unique Resource Identifier**), che hanno la medesima funzione ma sono limitati ai soli caratteri ASCII.
Gli URI costituiscono a loro volta un soprainsieme sia degli **URL** (**Universal Resource Locator**) sia degli **URN** (**Uniform Resource Name**). Il primo serve ad indicare la locazione di una risorsa (sul web), mentre il secondo il nome proprio della risorsa, scritto con una sintassi specifica. Pertanto, ad una risorsa é possibile riferirsi indifferentemente per locazione (URL) o per nome (URN). [3].

Le seguenti stringhe alfanumeriche sono degli IRI validi:

```
https://www.example.org/alice   https://en.wikipedia.org/wiki/Ice_cream   https://www.nyc.org
```

I letterali forniscono informazioni relative a descrizioni, date, valori numerici, ecc ... . In RDF, un letterale é costituito dalle seguenti tre componenti:

- Una **forma lessicale**, ovvero una stringa di caratteri Unicode;
- Un **datatype IRI** che indica il tipo di dato del letterale, definendo un dominio di possibili valori che questo puó assumere. Viene preceduto da "^^";
- Un **language tag** che indica la lingua in cui il termine viene espresso. Viene preceduto da "@"

2. Questa non é comunque una buona pratica, dato che é comunque preferibile definire lo schema prima dei dati.
3. Si noti come gli IRI risolvono il problema di avere a che fare con risorse diverse aventi lo stesso nome, ma non risolvono il problema inverso, ovvero dove IRI distinti si riferiscono alla stessa risorsa. RDF permette che una situazione di questo tipo si verifichi, ma in genere é preferibile risolvere questo tipo di conflitti adottando uno degli IRI che si riferiscono alla stessa risorsa a discapito degli altri.

I letterali piú semplici sono quelli composti dalla sola forma lessicale; il datatype ed il language tag sono opzionali, ma spesso utili a dare l'interpretazione corretta del letterale a cui si riferiscono. I tipi di dato definiti da RDF sono un sottoinsieme dallo standard XSD, a cui si aggiungono i tipi di dato `rdf:XML` e `rdf:XMLLiteral` propri di RDF. Questi possono essere raggruppati in quattro categorie:

- **Booleani**, (`xsd:boolean`);
- **Numerici**, sia interi (`xsd:decimal` , `xsd:byte` , `xsd:unsignedInt` , ecc ...) che razionali (`xsd:float` e `xsd:double`);
- **Temporali**, che siano istanti di tempo (`xsd:time` , ...), lassi di tempo (`xsd:duration` , ...) o una data specifica (`xsd:gDay` , `xsd:gMonth` , `xsd:gYear` , ...);
- **Testuali**, sequenze di caratteri generiche (`xsd:string`) oppure conformi rispetto ad una certa sintassi (`rdf:XML` , `rdf:XMLLiteral` , `xsd:anyURI` , ecc ...).

Alcuni tipi di dato sono derivati da altri tipi di dato, ovvero restringono i valori ammissibili dal dato da cui derivano ad un sottoinsieme piú piccolo (e piú specifico); i tipi di dato che non derivano da altri sono detti **primitivi**. Inoltre, mentre alcuni tipi di dato (come `xsd:decimal`) hanno una cardinalitá infinita numerabile, altri (come `xsd:unsignedLong`) hanno un numero finito di valori ammissibili.
Se ad un letterale non é associato un tipo di dato, si assume che sia di tipo `xsd:string` ; l'unica eccezione sono i letterali che presentano un language tag, a cui viene implicitamente assegnato il tipo `rdf:langString` . Sebbene RDF ammetta la possibilitá di definire dei tipi di dato custom, non fornisce un meccanismo standard per riportare esplicitamente che tale tipo di dato derivi da un altro, o per definire un dominio di valori ammissibili.
Vi sono situazioni in cui é preferibile che una certa risorsa non venga identificata per mezzo di un IRI, ad esempio perché un'informazione é mancante oppure perché non é rilevante. RDF gestisce tali casistiche per mezzo dei **blank nodes**, che per convenzione hanno come prefisso il carattere "_". Se una risorsa é identificata da un blank node, significa che tale risorsa esiste, ma non si ha modo o interesse di assegnarle un nome. I blank node operano come variabili esistenziali locali al loro dataset; due blank node di due dataset distinti si riferiscono a due risorse distinte.

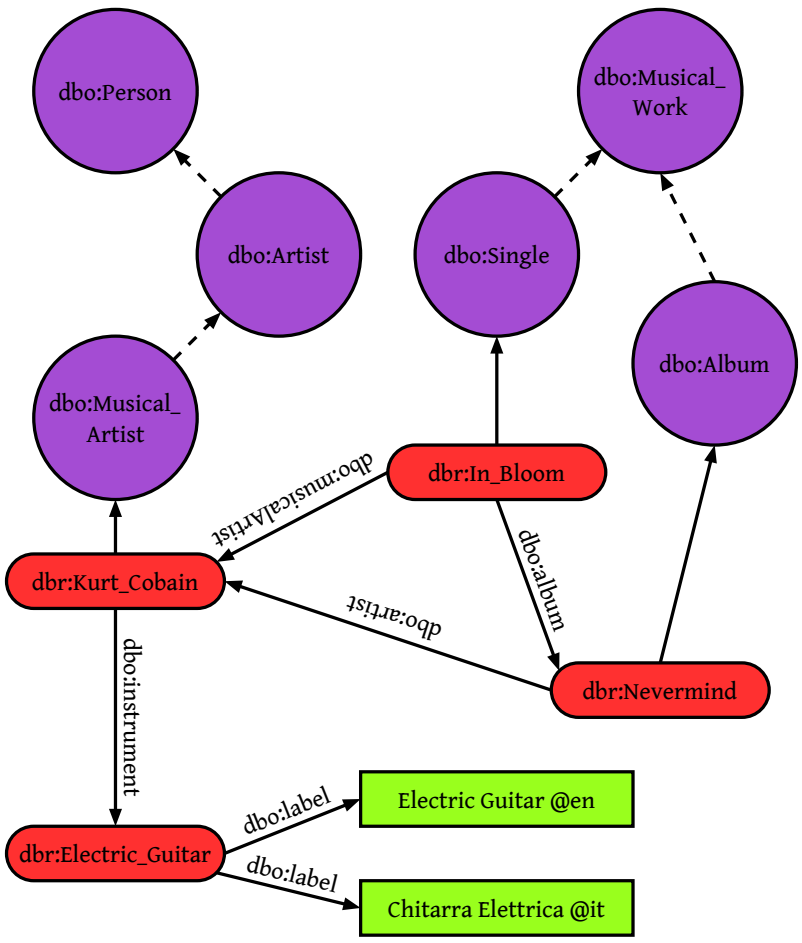
2.4.1 Triple

I dati in formato RDF non possono riportare risorse singole, ma solo ed esclusivamente **triple**. Una tripla RDF é nella forma soggetto-predicato-oggetto [4], dove tutti e tre gli elementi sono termini RDF. Nello specifico, il soggetto deve essere un IRI o un blank node, il predicato deve essere un IRI e l'oggetto puó essere di qualsiasi tipo di termine.

```
ex:Boston ex:hasPopulation "646000"^^xsd:integer                ex:VoynichManuscript ex:hasAuthor _:b
```

Queste restrizioni sono in linea con lo scopo che RDF si prefissa. Ai predicati deve necessariamente venire fornito un nome, dato che l'informazione "un soggetto ed un oggetto sono legati da un predicato ignoto" non é particolarmente rilevante. Inoltre, tale nome deve essere unico, perché i predicati devono poter essere univocamente identificati in qualsiasi dataset. Infine, per RDF, i letterali sono risorse di minore importanza rispetto agli IRI, pertanto sarebbe poco sensato averli come soggetto di una tripla.
Sebbene le triple RDF non abbiano di per loro una semantica, le restrizioni sui tipi di termini che possono comparire in ciascuna tripla porta portano a due tipi di interpretazioni. Se il primo elemento é un IRI o un blank node ed terzo elemento é un letterale, la tripla é da interpretarsi come una descrizione: la tripla (A, B, C) é da intendersi come "All'entitá A é associata la proprietá C". Se il primo elemento é un IRI o un blank node ed terzo elemento é un IRI, la tripla é da interpretarsi come una relazione: la tripla (A, B, C) é da intendersi come "L'entitá A é legata per mezzo di B all'entitá C" [5].
Un insieme di triple RDF costituisce un **grafo RDF**. Il nome grafo deriva dall'osservazione che ciascuna tripla RDF puó essere rappresentata in maniera equivalente come una coppia di nodi di un grafo uniti da un arco: l'etichetta di tale arco é il predicato della tripla, il soggetto é il nodo di partenza dell'arco e l'oggetto é il nodo di arrivo. Piú triple RDF danno allora vita ad un grafo diretto ed etichettato. Tale grafo é un esempio di knowledge graph.
Il fatto che RDF sia un modello di dati strutturato a grafo lo rende molto flessibile. Infatti, per introdurre nuovi predicati in un grafo RDF é sufficiente aggiungere un arco che ha tale predicato come etichetta, cosí come per introdurre nuovi soggetti o oggetti é sufficiente aggiungere dei nodi. Similmente, due grafi diversi (che corrispondono a due dataset diversi) possono essere unificati in maniera diretta mediante l'operazione di unione sui due insiemi di triple; l'unica eccezione sono i grafi che contengono dei blank node, perché il loro significato dipende dal grafo in cui si trovano, ed é quindi necessario prendere misure aggiuntive.

4. La struttura segue quella delle lingue anglosassoni.
5. Sebbene, per convenzione, il soggetto di una tripla sia la "risorsa primaria" che viene descritta dalla tripla stessa, la distinzione é del tutto arbitraria, in quanto é possibile invertire l'ordine del soggetto e dell'oggetto di una tripla per ottenerne una che descrive la stessa cosa.



2.5 Sintassi: N-triples e Turtle

Le uniche forme di sintassi specificate da RDF sono il vincolo di tripla ed i tipi di termine che possono comparire nelle tre posizioni delle triple. A parte queste restrizioni, RDF non fornisce alcun formalismo su come, ad esempio, riportare gli IRI ed i letterali. A tal scopo, sono stati definiti diversi formalismi per le triple RDF.
Una rappresentazione testuale estremamente semplice é **N-triples**; questa prevede di riportare per intero ciascun elemento di ogni tripla, una tripla per riga, terminandole con un punto. Le tre componenti di ciascuna tripla ed il punto alla fine della tripla sono separate da uno o piú caratteri di spaziatura (spazi, tab, a capo, ecc ...). Se un elemento é un IRI, viene riportato fra parentesi angolate, mentre se é un letterale viene riportato fra doppi apici. I blank node, i language tag ed i datatype IRI vengono riportati come di consueto. Una riga che inizia con il carattere "#" viene interpretata come un commento.

Le tre triple riportate di seguito sono sintatticamente valide per N-triples.

```
<http://www.example.org/alice>                  <http://schema.org/knows>          <http://www.example.org/bob>    .
_:dave                                          <http://xmlns.com/foaf/0.1/name>   "Dave Beckett"^^xsd:string     .
<http://www.w3.org/2001/sw/RDFCore/ntriples/>   <http://purl.org/dc/terms/title>   "N-Triples"@en-US               .
```

N-triples é tanto intuitivo quanto poco leggibile, perché gli IRI sono sempre riportati per intero, e gli IRI tendono ad essere molto lunghi. Una rappresentazione testuale leggermente piú complessa é **Turtle**, che eredita la sintassi di N-triples estendendola ed aggiungendovi delle abbreviazioni per migliorarne la leggibilitá.
Ai prefissi puó essere associata una parola chiave mediante la direttiva `@prefix:` . Se due triple consecutive hanno in comune il soggetto, é possibile terminare la prima con un punto e virgola e non riportare il soggetto nella seconda. Se due triple consecutive hanno in comune sia il soggetto che il predicato, é possibile terminare la prima con una virgola e non riportare soggetto e predicato nella seconda.

Turtle permette di definire triple RDF molto piú facilmente rispetto a N-triples.

```
@prefix dbr:   <http://dbpedia.org/resource/>   .
@prefix dbo:   <http://dbpedia.org/ontology/>   .


dbr:Kurt_Cobain   dbo:instrument      dbr:Electric_guitar .
dbr:In_Bloom      dbo:musicalArtist   dbr:Kurt_Cobain     ;
                  dbo:album           dbr:Nevermind       .
dbr:Nevermind     dbo:artist          dbr:Kurt_Cobain     .
```

2.6 SPARQL Protocol And RDF Query Language

Avendo a disposizione un grafo RDF, ci si chiede come sia possibile formulare domande sullo stesso, ad esempio determinare se esiste una tripla in cui figura un certo IRI. Dato che porre questo tipo di domande in linguaggio naturale è di difficile interpretazione per una macchina, queste vanno riformulate in un **linguaggio di query**. In particolare, un linguaggio di query appositamente pensato per estrarre informazioni da grafi RDF è **SPARQL (SPARQL Protocol And RDF Query Language)** [6].
La nozione più importante nel linguaggio SPARQL è il **pattern di tripla RDF**. Questa è di fatto analoga ad una tripla RDF, ma oltre ad ammettere IRI, letterali e nodi blank può contenere anche **variabili di query**, che ha il carattere "?" come prefisso. Tale pattern viene riportato nel quarto campo di una query SPARQL dopo la direttiva `WHERE .`
Un pattern di tripla viene valutato mappando le variabili/costanti del pattern alle costanti del grafo, di modo che l'immagine del pattern rispetto alla mappa (dove le variabili del pattern sono sostituite con le rispettive costanti del grafo) sia un sottografo del grafo. Nello specifico, gli IRI ed i letterali hanno un match solamente con, rispettivamente, un IRI ed un letterale a loro identico, mentre i blank node e le variabili di query hanno un match con qualsiasi termine. La differenza fra i due sta nel fatto che i termini che hanno un match con una variabile di query possono venire restituiti come parte della soluzione, mentre quelli che hanno un match con un blank node non possono.
Sia Con un insieme infinito numerabile di costanti, e sia invece Var un insieme infinito numerabile di variabili: i due insiemi sono disgiunti. L'insieme dei termini $Term$ é formulato come $Term = Con \cup Var$. Un grafo diretto ed etichettato é definito come una tupla $G = (V, E, L)$, dove $V \subseteq Con$ é un insieme di nodi, $L \subseteq Con$ é un insieme di etichette e $E \subseteq V \times L \times V$ é un insieme di archi.
Un pattern di tripla é formalmente definito come una tupla $Q = (V, E, L)$, dove $V \subseteq Term$ é un insieme di termini assegnabili ai nodi (IRI e blank nodes), $L \subseteq Term$ é un insieme di termini assegnabili agli archi (IRI) e e $E \subseteq V \times L \times V$ é un insieme di archi (triple pattern).
Sia $\mu : Var \mapsto Con$ una mappa, il cui dominio é indicato con Dom(μ). Dato un pattern di tripla Q, sia $Var(Q)$ l'insieme di tutte le variabili che compaiono in Q. Sia poi μ(Q) l'immagine di Q rispetto ad μ, ovvero il sottografo indotto da Q dove tutte le variabili $v \in \text{Var}(Q) \cap \text{Dom}(\mu)$ vengono sostituite con $\mu(v)$.
Dati due grafi diretti ed etichettati $G_1 = (V_1, E_1, L_1)$ e $G_2 = (V_2, E_2, L_2)$, si dice che G_1 é sottografo di G_2 se $V_1 \subseteq V_2, E_1 \subseteq E_2, L_1 \subseteq L_2$.
Formalmente, sia Q un pattern di tripla e sia G un grafo diretto ed etichettato. La valutazione del pattern Q sul grafo G, indicato con $Q(G)$, viene definito dall'insieme $Q(G) = \{\mu \mid \mu(Q) \subseteq G \wedge \text{Dom}(\mu) = \text{Var}(Q)\}$.
Un pattern di tripla restituisce una tabella. Per questo motivo, un pattern di tripla puó venire poi esteso con gli operatori propri dell'algebra relazionale per creare **pattern complessi**. Gli operatori elementari dell'algebra relazionale sono i seguenti:

- π, che restituisce la tabella con una o piú colonne rimosse;
- σ, che restituisce solo le righe della tabella che rispettano una determinata condizione;
- ρ, che restituisce la tabella con una o piú colonne cambiate di nome;
- ∪, che unisce le righe di due tabelle in un'unica tabella;
- −, che rimuove le righe della prima tabella che compaiono nella seconda;
- ⋈, che estendono le righe della prima tabella con le righe della seconda tabella che rispettano una determinata condizione;

I pattern complessi sono definiti in maniera ricorsiva come segue:

- Se Q é un pattern semplice, allora Q é un pattern complesso;
- Se Q é un pattern complesso e $V \subseteq \text{Var}(Q)$, allora $\pi_v(Q)$ é un pattern complesso;
- Se Q é un pattern complesso e R é una condizione di selezione espressa per mezzo di operatori booleani ($\wedge, \vee, \neg, =$), allora $\sigma_R(Q)$ é un pattern complesso;
- Se Q_1 e Q_2 sono due pattern complessi, allora $Q_1 \bowtie Q_2$, $Q_1 \cup Q_2$ e $Q_1 - Q_2$ sono pattern complessi.

Data una mappa μ, per un insieme di variabili $V \subseteq \text{Var}$ sia $\mu[V]$ la proiezione delle variabili V da μ, ovvero la mappatura μ' tale per cui $\text{Dom}(\mu') = \text{Dom}(\mu) \cap V$ e $\mu'(v) = \mu(v)$ per ogni $v \in \text{Dom}(\mu')$. Data la condizione di selezione R ed una mappa μ, si indica con $\mu \vdash R$ che la mappa μ soddisfa R. Infine, due mappe μ_1 e μ_2 vengono dette *compatibili* se $\mu_1(v) = \mu_2(v) perogniv \in \text{Dom}(\mu_1) \cap \text{Dom}(\mu_2)$, ovvero se mappano le variabili che hanno in comune alle medesime costanti. Due mappe compatibili μ_1 e μ_2 si indicano con $\mu_1 \sim \mu_2$.
Le operazioni sui pattern semplici, che restituiscono pattern complessi, si indicano allora come segue:

6. Sia il nome che la struttura delle query di SPARQL hanno molto in comune con **SQL**, che è invece un linguaggio di query per database relazionali.

- $\pi_V(Q)(G) = \{\mu \mid \mu \in Q(G)\}$
- $\sigma_R(Q)(G) = \{\mu \mid \mu \in Q(G) \wedge \mu \vdash R\}$
- $Q_1 \bowtie Q_2(G) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in Q_2(G) \wedge \mu_2 \in Q_1(G) \wedge \mu_1 \sim \mu_2\}$
- $Q_1 \cup Q_2(G) = \{\mu \mid \mu \in Q_1(G) \vee \mu \in Q_2(G)\}$
- $Q_1 - Q_2(G) = \{\mu \mid \mu \in Q_1(G) \wedge \mu \notin Q_2(G)\}$

Una funzionalitá che distingue i linguaggi di query é la possibilitá di includere le **path expression** nelle query. Una path expression é una espressione regolare che permette di avere un match su percorsi di lunghezza variabile fra due nodi mediante una **path query** (x, r, y), dove x e y possono essere sia variabili che costanti. Le path expression *semplici* sono quelle dove r é una costante, ovvero l'etichetta di un arco; si noti come le path expression siano sempre invertibili. É poi possibile costruire path expression *complesse* mediante i noti operatori delle espressioni regolari oppure mediante inversione:

- Se r é una path expression (l'etichetta di un arco), allora r^* é una path expression (un certo numero di archi etichettati r o anche nessuno);
- Se r é una path expression, allora r^- é una path expression (l'etichetta r letta a rovescio);
- Se r_1 e r_2 sono due path expression, allora $r_1 \mid r_2$ é una path expression (é presente l'etichetta r_1 di un arco oppure é presente l'etichetta r_2 di un arco);
- Se r_1 e r_2 sono due path expression, allora $r_1 \cdot r_2$ é una path expression (é presente l'etichetta r_1 di un arco seguita dall'etichetta r_2 di un arco).

Dato un grafo diretto ed etichetato $G = (V, E, L)$ ed una path expression r, si definisce l'applicazione di r su G, ovvero $r[G]$, come segue:

- $r[G] = \{(u, v) \mid (u, r, v) \in E\}(r \in Con)$
- $r^-[G] = \{(u, v) \mid (v, u) \in r[G]\}$
- $r_1 \mid r_2[G] = r_1[G] \cup r_2[G]$
- $r_1 \cdot r_2[G] = \{(u, v) \mid \exists w \in V : (u, w) \in r_1[G] \wedge (w, v) \in r_2[G]\}$
- $r^*[G] = \{(u, u) \mid u \in V\} \bigcup_{n \in \mathbb{N}^+} r^n[G]$

Dato un grafo diretto ed etichettato G, delle costanti $c_i \in Con$ e delle variabili $z_i \in Var$, una **path query** semplice é una tripla (x, y, z) dove $x, y \in Con \cup Var$ e r é una path expression. La valutazione di una path query é definita come segue:

- $(c_1, r, c_2)(G) = \{\mu_\emptyset \mid (c_1, c_2) \in r[G]\}$
- $(c, r, z)(G) = \{\mu \mid \mathrm{Dom}(\mu) = \{z\} \wedge (c, \mu(z)) \in r[G]\}$
- $(z, r, c)(G) = \{\mu \mid \mathrm{Dom}(\mu) = \{z\} \wedge (\mu(z), c) \in r[G]\}$
- $(z_1, r, z_2)(G) = \{\mu \mid \mathrm{Dom}(\mu) = \{z_1, z_2\} \wedge (\mu(z_1), \mu(z_2)) \in r[G]\}$

Dove $\mu_\emptyset$ indica la mappatura vuota, ovvero $\mathrm{Dom}(\mu_\emptyset) = \emptyset$.
Path query semplici possono essere usate come pattern di tripla per ottenere **graph pattern di navigazione**. Se Q é un pattern di tripla, allora é anche un graph pattern di navigazione. Se Q é un graph pattern di navigazione e (x, r, y) é una path query, allora $Q \bowtie (x, r, y)$ é un graph pattern di navigazione.
Una query SPARQL è costituita dalle seguenti sei componenti, non tutte strettamente obbligatorie:

1. *Dichiarazione dei prefissi*. Similmente a Turtle, è possibile dichiarare dei prefissi mediante la direttiva `PREFIX`, seguita dal nome scelto per il prefisso e dall'URI a cui il prefisso è associato;
2. *Tipo di query*. SPARQL supporta quattro tipi di query:
   - `SELECT`, che restituisce il risultato della query sotto forma di tabella. Questa supporta l'eliminazione delle soluzioni duplicate per mezzo delle direttive `REDUCED` (possono essere rimosse) e `DISTINCT` (devono essere rimosse). É possibile restituire l'intera tabella con tutte le colonne con "*" oppure specificando solo parte delle colonne mediante proiezione;
   - `ASK`, che restituisce true se la query ha un risultato non nullo e false altrimenti;
   - `CONSTRUCT`, che restituisce il risultato della query sotto forma di (sotto) grafo;
   - `DESCRIBE`, che restituisce il risultato della query sotto forma di grafo che descrive termini e soluzioni.
3. *Costruzione del dataset*. mediante la direttiva `FROM` è possibile specificare su quale/i grafo/i si vuole operare la query. Se vengono specificati più grafi, la query verrà operata sulla loro unione;
4. *Pattern*. La direttiva `WHERE` specifica il pattern che discrimina un elemento del grafo che è parte della soluzione da uno che non lo è. Le condizioni sono riportate in un blocco di parentesi graffe seguendo la sintassi Turtle;
5. *Aggregazione*. Le direttive `GROUP BY` e `HAVING`, analoghe alle direttive omonime di SQL permettono di raggruppare o di filtrare gli elementi della soluzione secondo specifiche regole. I valori possono venire aggregati sulla base di diverse direttive quali `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`;
6. *Modificatori della soluzione*. Alcune direttive permettono di modificare gli elementi della soluzione disponendoli secondo un certo ordine (`ORDER BY`) oppure restituendone solo una parte.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX mo: <http://purl.org/ontology/mo/>


SELECT ?album_name ?track_title
WHERE
    dbpedia:The_Beatles foaf:made ?album       .
    ?album              dc:title  ?album_name  ;
                        mo:track  ?track       .
    ?track              dc:title  ?track_title .
```

I modificatori di soluzione sono diversi, fra cui figurano:

- OPTIONAL quando una parte del grafo non é obbligatoria;
- UNION quando si vuole ricavare l'unione di due o piú sottografi risultanti;
- MINUS quando si vuole eliminare i risultati che hanno una corrispondenza con un pattern;
- VALUES quando parte del match é predefinito;
- BIND quando parte del match é precalcolato;
- FILTER quando occorre rimuovere i risultati che rispecchiano un certo pattern espresso sottoforma di espressione booleana;

Le espressioni booleane ammesse in SPARQL possono contenere i seguenti elementi:

- Variabili e costanti;
- Operatori di uguaglianza e disuguaglianza: = , <= , >= , != ;
- Connettori: && , || , (,) , ! ;
- Espressioni regolari: regex(?x, "A.*") ;
- Test sulla natura delle variabili: isURI(?x) , isBlank(?x) , isLiteral(?x) , bound(?x) .

- Inclusione di una stringa
  CONTAINS(literal1, literal2), STRSTARTS(literal1, literal2), STRENDS(literal1, literal2)
- Crea un letterale con un tipo di dato associato
  STRDT(value, type)
- Crea un letterale con un language tag associato
  STRLANG(value, lang)
- Concatena piú stringhe
  CONCAT(literal1, literal2, ..., literaln)
- Estrai una sottostringa
  SUBSTR(literal, start [, length])

2.7 RDFS

Come giá detto, il terzo membro di una tripla RDF puó essere un IRI o un letterale. Nel primo caso, é possibile vedere tale tripla come la descrizione di una relazione fra l'entitá primo membro della tripla e l'entitá terzo membro della tripla, mentre nel secondo caso la tripla riporta che il primo membro della tripla ha come attributo il terzo membro. Si noti peró come RDF non fornisca esplicitamente un'interpretazione di questo tipo, ma é piú una assunzione implicita.
La semantica definita da RDF si limita soltanto al vincolo di tripla (tutte le risorse devono essere nella forma soggetto-predicato-oggetto) ed il tipo di ciascun termine (il predicato non puó essere un blank node, il soggetto non puó essere un letterale, ecc ...). Al di lá di questo, RDF non permette la costruzione di una vera e propria **ontologia**.
L'ontologia é una branca della filosofia che si occupa di comprendere la natura delle cose e come categorizzarle. Nel contesto dell'informatica, con ontologia si intende una rappresentazione formale della conoscenza rispetto ad un determinato dominio; si occupa quindi di determinare quali sono le entitá che appartengono a tale dominio, come possono essere categorizzate, quali sono le loro proprietá, quali di queste proprietá sono rilevanti e quali no, ecc ...
L'obiettivo di una ontologia informatica non é quello di trovare la modellazione "corretta" (qualunque cosa questo significhi) per un determinato dominio, quanto piú trovare una rappresentazione che sia funzionale per tutte le parti interessate. Nel contesto di un sistema distribuito, costruire ontologie dettagliate le cui definizioni sono state prese di comune accordo da tutti i nodi fornisce loro una concettualizzazione comune, sulla base della quale potersi scambiare informazioni.
Resource Description Framework Schema (RDFS) é un semplice linguaggio che permette di associare uno schema ad un insieme di dati scritti in formato RDF. Questo permette di descrivere le risorse RDF in termini di classi e di proprietá. Queste hanno rdfs: come prefisso.
RDFS si compone di due elementi concettuali ad alto livello: le **proprietá** e le **classi**. Le proprietá sono le relazioni che sussistono fra coppie di risorse: sono i termini in genere presenti come predicati nelle triple. Le classi sono gruppi di risorse che hanno caratteristiche in comune. Una

risorsa puó essere membro di piú classi. Un membro di una classe é detto **istanza** di tale classe. La classe di una risorsa viene anche chiamata il suo **tipo**. Per convenzione, le classi hanno un nome con la prima lettera maiuscola, mentre le proprietá hanno un nome con la prima lettera minuscola.
RDFS permette inoltre di fare **inferenze** a partire dalle informazioni a disposizione. Nello specifico, a partire da una certa semantica, é possibile definire una nozione di **entailment** tra due grafi RDF di modo che se il primo grafo contiene triple vere, allora anche il secondo conterrá triple vere (rispetto alla medesima semantica). In questo caso, il secondo grafo non aggiunge alcuna informazione che non sia giá presente, eventualmente implicitamente, nel primo grafo. RDFS mette a disposizione 13 regole di inferenza:

Regola	**Se vale ...**	**... allora si deduce**
Regola 1	xxx aaa yyy .	aaa rdf:type rdfs:Property .
Regola 2	aaa rdfs:domain xxx . yyy aaa zzz .	yyy rdf:type xxx .
Regola 3	aaa rdfs:range xxx . yyy aaa zzz .	zzz rdf:type xxx .
Regola 4a	xxx aaa yyy .	xxx rdf:type rdfs:Resource .
Regola 4b	xxx aaa yyy .	yyy rdf:type rdfs:Resource .
Regola 5	xxx rdfs:subPropertyOf yyy . yyy rdfs:subPropertyOf zzz .	xxx rdfs:subPropertyOf zzz .
Regola 6	xxx rdf:type rdf:Property .	xxx rdfs:subPropertyOf xxx .
Regola 7	aaa rdfs:subPropertyOf bbb . xxx aaa yyy .	xxx bbb yyy .
Regola 8	xxx rdf:type rdfs:Class .	xxx rdfs:subClassOf rdfs:Resource .
Regola 9	xxx rdfs:subClassOf yyy . zzz rdf:type xxx .	zzz rdf:type yyy .
Regola 10	xxx rdf:type rdfs:Class .	xxx rdfs:subClassOf xxx .
Regola 11	xxx rdfs:subClassOf yyy . yyy rdfs:subClassOf zzz .	xxx rdfs:subClassOf zzz .
Regola 12	xxx rdf:type rdfs:ContainerMembershipProperty .	xxx rdfs:subPropertyOf rdfs:member .
Regola 13	xxx rdf:type rdfs:DataType .	xxx rdfs:subClassOf rdfs:Literal .

La proprietá `rdf:type` permette di istanziare una classe. La tripla `A rdf:type B` indica che l'entitá `A` é una istanza della classe `B`. Spesso `rdf:type` viene abbreviato con `a`. Diverse entitá in RDFS sono istanze di metaclassi predefinite:

- Ogni risorsa (classi, entitá, proprietá, letterali, ecc ...) é implicitamente istanza della metaclasse `rdfs:Resource` ;
- Tutte le proprietá sono istanza di `rdf:Property` ;
- Le classi sono istanza di `rdfs:Class` ;
- I letterali sono istanza di `rdfs:Literal` ;
- I tipi di dato (`xsd:string` , `xsd:integer` , ecc ...) sono istanza di `rdfs:Datatype` .

```
ex:LemonCheesecake   ex:contains   ex:Lemon
ex:LemonCheesecake   ex:contains   ex:Cheese
ex:LemonCheesecake   rdf:type      ex:DessertRecipe
ex:Lemon             rdf:type      ex:Ingredient
ex:Lemon             rdf:type      ex:Fruit
ex:Cheese            rdf:type      ex:Ingredient
ex:Cheese            rdf:type      ex:Dairy
```

`rdfs:subClassOf` mette due classi nella relazione di sottoclasse. La tripla `C rdfs:subClassOf D` indica che la classe `C` é una sottoclasse della classe `D` , ovvero che tutte le istanze di `C` sono automaticamente anche istanze di `D` . Questa relazione é sia riflessiva (ogni classe é sottoclasse di sé stessa) che transitiva (se `C` é sottoclasse di `D` e `D` é sottoclasse di `E` , allora `C` é sottoclasse di `E`).

Si consideri il seguente insieme di triple:

```
ex:DessertRecipe      rdfs:subClassOf   ex:Recipe
ex:VeganRecipe        rdfs:subClassOf   ex:VegetarianRecipe
ex:VegetarianRecipe   rdfs:subClassOf   ex:Recipe
ex:LemonPie           rdfs:subClassOf   ex:DessertRecipe
ex:LemonPie           rdfs:subClassOf   ex:VeganRecipe
```

Per simmetricitá, sono automaticamente vere anche le seguenti triple:

```
ex:DessertRecipe      rdfs:subClassOf   ex:Recipe
ex:Recipe             rdfs:subClassOf   ex:Recipe
ex:VeganRecipe        rdfs:subClassOf   ex:VeganRecipe
ex:VegetarianRecipe   rdfs:subClassOf   ex:VegetarianRecipe
ex:LemonPie           rdfs:subClassOf   ex:LemonPie
```

Inoltre, per transitivitá, vale:

```
ex:VeganRecipe        rdfs:subClassOf   ex:Recipe
ex:LemonPie           rdfs:subClassOf   ex:VegetarianRecipe
ex:LemonPie           rdfs:subClassOf   ex:Recipe
```

rdfs:subPropertyOf mette due proprietá nella relazione di sottoproprietá. La tripla `C rdfs:subPropertyOf D` indica che la proprietá `P` é una sottoproprietá della proprietá `Q`, ovvero che tutte le coppie di entitá legate da `P` sono automaticamente legate anche da `Q`. Cosí come la relazione di sottoclasse, la relazione di sottoproprietá é é sia riflessiva che transitiva.

A partire dalle triple:

```
ex:hasTopping      rdfs:subPropertyOf   ex:hasIngredient
ex:hasIngredient   rdfs:subPropertyOf   ex:contains
```

É possibile inferire:

```
ex:hasTopping      rdfs:subPropertyOf   ex:hasTopping
ex:hasIngredient   rdfs:subPropertyOf   ex:hasIngredient
ex:contains        rdfs:subPropertyOf   ex:contains
ex:hasTopping      rdfs:subPropertyOf   ex:contains
```

rdfs:domain mette in relazione una proprietá `P` ed una classe `C`. La tripla `P rdfs:domain C` indica che se due elementi `x` e `y` sono messi in relazione dalla proprietá `P`, allora `x` é una istanza di `C`.

A partire dalle triple:

```
ex:hasIngredient   rdfs:domain        ex:Recipe
ex:LemonPie        ex:hasIngredient   ex:Lemon
```

É possibile inferire:

```
ex:LemonPie   rdf:type   ex:Recipe
```

rdfs:range mette in relazione una proprietá `P` ed una classe `C`. La tripla `P rdfs:range C` indica che se due elementi `x` e `y` sono messi in relazione dalla proprietá `P`, allora `y` é una istanza di `C`.

A partire dalle triple:

```
ex:hasIngredient   rdfs:range         ex:Ingredient
ex:LemonPie        ex:hasIngredient   ex:Lemon
```

É possibile inferire:

```
ex:Lemon   rdf:type   ex:Ingredient
```

Le classi e le proprietá forniscono un **vocabolario**, ovvero un insieme di termini RDF per descrizioni generali. Una singola proprietá o una classe puó essere usata per descrivere un numero arbitrario di istanze. É facile riutilizzare uno stesso vocabolario in diversi grafi RDF.
RDFS permette di fare query SPARQL su grafi RDF ed ottenere informazioni che non sono esplicitamente contenute nel grafo, applicando le regole di inferenza.

Si consideri il seguente grafo RDF (espresso in notazione Turtle):

```
@prefix dbr:  https://dbpedia.org/resource/
@prefix dbo:  https://dbpedia.org/ontology/
@prefix rdfs: https://www.w3.org/2000/01/rdfs-schema#

dbo:Singer                            rdfs:subClassOf     dbo:MusicalArtist     .
dbr:Come_As_You_Are_(Nirvana_Song)    dbo:Singer          dbr:Kurt_Cobain       .
dbr:Come_As_You_Are_(Nirvana_Song)    dbo:MusicalArtist   dbr:Dave_Grohl       .
dbr:Come_As_You_Are_(Nirvana_Song)    dbo:MusicalArtist   dbr:Krist_Novoselic   .
```

É possibile derivare che le entitá dbr:Kurt_Cobain e dbr:Come_As_You_Are_(Nirvana_Song) sono legate da dbo:MusicalArtist applicando le regole di entailment RDFS. Questo perché le due entitá sono legate da dbo:Singer e tale classe é una sottoclasse di dbo:MusicalArtist . Infatti, tale tripla é presente nel risultato dalla seguente query SPARQL nonostante nel grafo non sia riportata esplicitamente:

```
SELECT ?name
WHERE {
   dbr:Come_As_You_Are_(Nirvana_Song)   dbo:MusicalArtist   ?name   .
}
```

?name
dbr:Kurt_Cobain
dbr:Dave_Grohl
dbr:Krist_Novoselic

Esistono due approcci in merito al combinare le inferenze e le query. Il primo prevede di applicare le regole di inferenza su tutte le triple del grafo prima che questo venga pubblicato e salvare tutte le triple inferite all'interno dello stesso. In questo modo, quando viene effettuata una query, tutte le triple sono giá presenti nel grafo ed é sufficiente restituirle. Questo comporta peró che ogni volta che il grafo viene modificato, ad esempio perché viene introdotta o rimossa una tripla, occorre riapplicare le regole di inferenza per aggiornarlo.
Il secondo approccio prevede di applicare le regole di inferenza quando viene effettuata una query che le richiede. In questo modo non é necessario aggiornare il grafo ogni volta che questo viene modificato, ma d'altra parte ogni query sará piú lenta perché é necessario spendere ulteriore tempo per il calcolo delle inferenze.

2.8 OWL

Le ontologie che RDFS permette di costruire non sono particolarmente espressive. Ad esempio, RDFS presenta le seguenti limitazioni:

- Non é possibile modellare le **classi disgiunte**, ovvero non é possibile definire delle classi a cui sia impedito avere istanze in comune;
- Non é possibile specificare che una proprietá sia transitiva, inversa e/o simmetrica;
- Non é possibile specificare un vincolo di **cardinalitá**, ad esempio che l'istanza di una classe possa essere in relazione con al massimo *n* istanze di un'altra classe;
- Non é possibile costruire classi applicando gli operatori dell'insiemistica (unione, intersezione, complemento) sulle classi esistenti;
- Non é possibile definire un range/dominio che vari in base a quale entitá si riferisce.

Si consideri il grafo RDF presentato di seguito, che contiene informazioni relative a cittá, regioni e paesi:

```
@prefix rdf:  https://www.w3.org/1999/02/22-rdf-syntax-ns#
@prefix rdfs: https://www.w3.org/2000/01/rdfs-schema#
@prefix dbr:  https://dbpedia.org/resource/
@prefix dbo:  https://dbpedia.org/ontology/

capitalOf      rdfs:domain   dbo:Capital    .
capitalOf      rdfs:range    dbo:Country    .
cityOf         rdfs:range    dbo:Country    .
cityOf         rdfs:range    dbo:Region     .

dbr:Milan      cityOf        dbr:Lombardy   .
dbr:Milan      cityOf        dbr:Italy      .
dbr:Rome       capitalOf     dbr:Italy      .
dbr:Italy      rdf:type      dbo:Country    .
dbr:Lombardy   rdf:type      dbo:Region     .
```

Nonostante le triple siano tutte logicamente valide, é comunque possibile applicare le regole di inferenza di RDFS per derivare delle triple che non lo sono.
Ad esempio, `dbr:Milan` é legato sia a `dbr:Italy` che a `dbr:Lombardy` per mezzo del predicato `cityOf`. Tuttavia, l'esistenza della tripla `cityOf rdfs:range dbo:Country .` permette di applicare la regola di inferenza 3, a partire dalla quale si deriva che `dbr:Lombardy` é una istanza della classe `dbo:Country`.

Per sopperirvi é necessario utilizzare un linguaggio piú ricco. A tal scopo é stato definito **Ontology Web Language** (**OWL**), che opera come RDFS su triple conformi allo standard RDF ma permettendo una modellazione piú fine.
A differenza di RDFS, che si compone di "sole" 13 regole di inferenza, OWL si prefigge di modellare una ontologia molto complessa, ed un insieme di regole di inferenza dedicate non sarebbe sufficiente. Per questo motivo, OWL utilizza un linguaggio logico vero e proprio, ispirato ad una famiglia di linguaggi chiamati **Description Logic** (**DL**). Tali linguaggi non sono altro che restrizioni della logica del primo ordine. Nel caso specifico di OWL2, la versione attuale[7], la DL di riferimento é chiamata **SROIQ**: sebbene SROIQ e OWL siano intimamente collegati, i due hanno terminologie distinte, ma mappabili uno-ad-uno.
A partire dalla specifica completa di OWL sono stati definiti tre **profili**. Questi sono dei "dialetti" di OWL 2, ovvero delle restrizioni al linguaggio pensati per distinti casi d'uso. Ogni profilo ha una propria ricchezza espressiva ed una propria capacitá computazionale. [8]. I profili sono tre:

- **OWL 2 EL** permette di modellare classificazioni semplici (comunque piú sofisticate di quanto possa fare RDFS), ma viene garantito il calcolo delle inferenze in tempo polinomiale;
- **OWL 2 QL** é costruito di modo che le inferenze siano automaticamente traducibili come query su database relazionali;
- **OWL 2 RL** é pensato per essere implementato in maniera efficiente in sistemi a regole.

Se RDFS metteva a disposizione la relazione di sottoclasse, OWL fornisce il predicato `owl:equivalentClass`, che indica che le due classi che mette in relazione hanno gli stessi membri. Tale predicato é piú ricco di `rdfs:subClassOf`, perché oltre allo specificare che una classe é sottoclasse di un'altra é anche possibile introdurre dei vincoli aggiuntivi. Inoltre, il predicato `owl:disjointWith` indica che due classi non possono avere una istanza in comune.

L'inconsistenza nell'esempio precedente viene risolta introducendo la tripla `dbo:Region owl:disjointWith dbo:Country .`, perché in questo modo si impedisce che `dbr:Lombardy` possa essere istanza di `dbo:Country`.

L'istanza di una classe in OWL prende il nome di **individuo**. Un individuo é legato alla propria classe per mezzo di `rdfs:type`. OWL permette di specificare che due individui sono in realtá lo stesso individuo (nonostante abbiano due IRI distinti) per mezzo del predicato `owl:sameAs`. Inoltre, é possibile specificare che due individui sono distinti per mezzo del predicato `owl:differentFrom`. OWL, infatti, non adotta la politica *UniqueNameAssumption* (**UNA**), ovvero l'idea che due entitá a cui sono stati assegnati due nomi diversi (due URI diversi, in questo caso) siano necessariamente distinte esse stesse.

7. Per comoditá, da ora in poi con "OWL" si intenderá la specifica completa della seconda versione del linguaggio (se non diversamente specificato).
8. In genere, i reasoner commerciali utilizzano una intersezione di questi dialetti di modo da bilanciare efficienza ed espressivitá.

```
@prefix dbr:  https://dbpedia.org/resource/
@prefix msb:  https://musicbrainz.org/artist/
@prefix owl:  https://www.w3.org/2002/07/owl#

msb:5b11f4ce-a62d-471e-81fc-a69a8278c7da   owl:sameAs          dbr:Nirvana_(band)    .
dbr:Nirvana                                owl:differentFrom   dbr:Nirvana_(band)    .
```

Sebbene le proprietá in RDF(S) siano in genere modellate come attributi di una entitá o come relazione fra due entitá, non esiste un costrutto che permetta di fare esplicitamente questa distinzione. In OWL si distingue invece fra `owl:ObjectProperty`, ovvero proprietá i cui valori sono risorse, e `owl:DatatypeProperty`, ovvero proprietá i cui valori sono letterali.
Cosí come per le classi, OWL permette di stabilire che due proprietá (con diverso IRI) si riferiscono alla medesima proprietá per mezzo del predicato `owl:equivalentProperty`, e stabilire che due proprietá sono distinte per mezzo di `owl:propertyDisjointWith`.
OWL permette di assegnare delle caratteristiche alle proprietá che permettono di inferire nuovi fatti sulla base delle stesse:

- Se una proprietá `p` appartiene alla classe `owl:SymmetricProperty`, allora tale proprietá é simmetrica. Ovvero:

  Se in un grafo sono presenti le triple `p rdf:type owl:SymmetricProperty .` `x p y .` Allora é possibile inferire `y p x .`

- Se due proprietá `p` e `q` sono messe in relazione dal predicato `owl:inverseOf`, allora tali proprietá sono l'una l'inversa dell'altra. Ovvero:

  Se in un grafo sono presenti le triple `p owl:inverseOf q .` `x p y .` Allora é possibile inferire `y q x .`

- Se una proprietá `p` appartiene alla classe `owl:TransitiveProperty`, allora tale proprietá é transitiva. Ovvero:

  Se in un grafo sono presenti le triple `p rdf:type owl:TransitiveProperty .` `x p y .` `y p z .` Allora é possibile inferire `x p z .`

- Se una proprietá `p` appartiene alla classe `owl:FunctionalProperty`, allora tale proprietá é una relazione funzionale, ovvero una relazione il cui argomento é associato ad al piú un valore. Ovvero:

  Se in un grafo sono presenti le triple `p rdf:type owl:FunctionalProperty .` `x p y .` `x p z .` Allora é possibile inferire `y owl:sameAs z .`

- Se una proprietá `p` appartiene alla classe `owl:InverseFunctionalProperty`, allora l'inverso di tale proprietá é una relazione funzionale. Ovvero:

  Se in un grafo sono presenti le triple `p rdf:type owl:InverseFunctionalProperty .` `x p z .` `y p z .` Allora é possibile inferire `x owl:sameAs y .`

Si consideri il grafo RDF presentato di seguito:

```
@prefix rdf:  https://www.w3.org/1999/02/22-rdf-syntax-ns#
@prefix rdfs: https://www.w3.org/2000/01/rdfs-schema#
@prefix dbr:  https://dbpedia.org/resource/
@prefix dc:   https://purl.org/dc/elements/1.1/
@prefix foaf: https://xmlns.org/foaf/0.1/

foaf:knows   rdfs:domain   foaf:Person   ;
             rdfs:range    foaf:Person   .
foaf:made    rdfs:domain   foaf:Agent    .

dbr:Kurt_Cobain   foaf:made     dbr:Heart-Shaped_Box   ;
                  foaf:knows    dbr:Dave_Grohl         .
```

Sia in RDFS che in OWL é possibile inferire:

```
dbr:Kurt_Cobain   a   foaf:Agent    ;
                  a   foaf:Person   .
dbr:Dave_Grohl    a   foaf:Person   .
```

OWL permette peró di derivare molte piú informazioni. Aggiungendo al grafo le triple:

```
dc:creator   owl:inverseOf   foaf:Made                .
foaf:knows   a               owl:SymmetricProperty   .
```

É possibile inferire anche:


```
dbr:Heart-Shaped_Box   dc:creator   dbr:Kurt_Cobain   .
dbr:Dave_Grohl         foaf:knows   dbr:Kurt_Cobain   .
```

Per poter costruire classi mediante operatori booleani é necessario che queste siano organizzate in una struttura a **reticolo**, con un top ed un bottom. A tal scopo, ogni entitá in OWL (classi, proprietá, letterali, ecc ...) é implicitamente istanza della classe owl:Thing , mentre la classe owl:Nothing é la classe che non ha istanze. Similmente, ogni proprietá é implicitamente istanza della classe owl:TopObjectProperty , mentre nessuna proprietá é istanza della classe owl:BottomObjectProperty . Le classi possono avere piú superclassi dirette; le proprietá possono avere piú superproprietá dirette.

La sintassi di SROIQ é composta da tre elementi: **concetti**, **ruoli** e **asserzioni**. Un concetto SROIQ corrisponde ad una classe OWL, un ruolo SROIQ ad una proprietá OWL ed una asserzione SROIQ ad un individuo. Si distinguono poi le **definizioni** dagli **assiomi**: le definizioni permettono di fare riferimento ad un concetto/ruolo/asserzione o di definirne di nuovi, mentre gli assiomi specificano una proprietá di un certo concetto/ruolo/asserzione.

Le asserzioni che si riferiscono ai concetti e ai ruoli costituiscono la **Terminological Box** (**T-box**), mentre le asserzioni che si riferiscono agli assiomi costituiscono la **Assertional Box** (**A-box**). La A-box riporta le informazioni relative agli individui OWL; a tutti gli individui é necessario associare un nome univoco (non sono ammessi blank node come in RDF(S)). La T-box definisce la semantica relativa alle classi OWL.

La semantica di una Description Logic, e quindi anche di SROIQ, é definita a partire da una **teoria dei modelli**. Una interpretazione I di una Description Logic é tipicamente definita come una coppia $(\Delta^I, \cdot^I)$, dove Δ^I é il **dominio di interpretazione** e $\cdot^I$ é la **funzione di interpretazione**. Il dominio di interpretazione contiene un insieme di individui. La funzione di interpretazione mappa la definizione di un individuo, di un concetto o di un ruolo e li mappa, rispettivamente, ad un elemento del dominio, ad un sottoinsieme del dominio o ad un insieme di coppie ordinate estratte dal dominio. D'altra parte, gli assiomi sono interpretati come condizioni semantiche. Dalla semantica di una Description Logic discende una "classica" nozione di entailment, ovvero dove per due ontologie O_1 e O_2 vale $O_1 \models O_2$ se e solo se ogni interpretazione che soddisfa O_1 soddisfa anche O_2.

Vi sono diverse possibili tecniche per costruire inferenze sulla base di una DL. Fra queste, figura la **tecnica a tableau**, una tecnica generale utilizzata in diverse logiche per testare la soddisfacibilitá di una o piú formule. L'idea alla base della tecnica consiste nell'esplorare lo spazio delle possibilitá che possono soddisfare tali formule: le possibilitá che conducono ad una contraddizione vengono scartate, e se tutte le possibilitá vengono scartate la formula é considerata una contraddizione. Nel caso specifico delle Description Logic, ad esempio, la tecnica a tableau prevede di esplorare tutte le possibilitá che possono condurre ad un modello per l'ontologia in esame; questa é allora soddisfacibile se (almeno) un modello esiste ed una contraddizione in caso contrario.

Si noti come non sia sempre possibile *chiudere* un tableau, ovvero esaurire tutte le possibilitá ed ottenere una risposta. Questo perché, essendo SROIQ una logica **indecidibile**, possono presentarsi dei cicli infiniti in cui vengono continuamente eseguite le stesse sostituzioni senza poter proseguire oltre. Un ciclo di questo tipo puó essere facilmente individuabile da un umano, ma un risolutore automatico fatica a distinguere una computazione molto onerosa (ma che giungerá a termine) da un ciclo infinito.

	Nome	Espr.	Semantica	Equivalente in OWL
Simboli di base	Individuo	a	$a^I \in \Delta^I$	
	Concetto	C	$C^I \subseteq \Delta^I$	Classe
	Ruolo	R	$R^I \subseteq \Delta^I \times \Delta^I$	Proprietá
Assiomi della Abox	Asserzione di concetto	$C(a)$	$a^I \in C^I$	:a :rdfType :C
	Asserzione di ruolo	$R(a, b)$	$(a^I, b^I) \in R^I$	:a :R :b
Assiomi della Tbox	Inclusione di concetto	$C \sqsubset D$	$C^I \subseteq D^I$	:C :rdfsSubclassOf :D
	Equivalenza di concetto	$C = D$	$C^I = D^I$	:C owl:EquivalentClass :D
Costruttori di ruolo	Inversione di ruolo	R^-	$\left(R^-\right)^I = \left\{\left(y, x\right) \mid \left(x, y\right) \in R^I\right\}$	
Costruttori di concetto	Top	$\top$	Δ^I	owl:Thing
	Bottom	$\bot$	$\emptyset$	owl:Nothing
	Negazione	$\neg C$	$\Delta^I - C^I$	[rdf:type owl:Class ; owl:complementOf :C]
	Intersezione	$C \sqcap D$	$C^I \cap D^I$	[rdf:type owl:Class ; owl:intersectionOf (:C :D)]
	Unione	$C \sqcup D$	$C^I \cup D^I$	[rdf:type owl:Class ; owl:unionOf (:C :D)]
	Nominale	$\{a\}$	$\{a^I\}$	[a owl:Class ; owl:oneOf (:a)]
	Restrizione esistenziale	$\exists R.C$	$\{x \in \Delta^I \mid R^I(x) \cap C^I \neq \emptyset\}$	[rdf:type owl:Restriction ; owl:onProperty :R ; owl:someValuesFrom :C]
	Restrizione universale	$\forall R.C$	$\{x \in \Delta^I \mid R^I(x) \subseteq C^I\}$	[rdf:type owl:Restriction ; owl:onProperty :R ; owl:allValuesFrom :C]
	Restrizione 'al piú'	$\leq nR.C$	$\{x \in \Delta^I \mid\mid R^I(x) \cap C^I \mid \leq n\}$	[rdf:type owl:Restriction ; owl:minQualifiedCardinality "n"^^xsd:nonNegativeInteger ; owl:onProperty :R ; owl:onClass :C]
	Restrizione 'almeno'	$\geq nR.C$	$\{x \in \Delta^I \mid\mid R^I(x) \cap C^I \mid \geq n\}$	[rdf:type owl:Restriction ; owl:maxQualifiedCardinality "n"^^xsd:nonNegativeInteger ; owl:onProperty :R ; owl:onClass :C]
	Restrizione esatta	$= nR.C$	$\{x \in \Delta^I \mid\mid R^I(x) \cap C^I \mid = n\}$	[rdf:type owl:Restriction ; owl:qualifiedCardinality "n"^^xsd:nonNegativeInteger ; owl:onProperty :R ; owl:onClass :C]
	Riflessivitá locale	$\exists R.\text{Self}$	$\{x \in \Delta^I \mid (x, x) \in R^I\}$	[rdf:type owl:Restriction ; owl:onProperty :R ; owl:hasSelf "true"^^xsd:boolean]

A-box	T-box
GenitoreEquinoMaschio(Zia, Marty) GenitoreEquinoMaschio(Zach, Marty) GenitoreEquinoFemmina(Zia, Lea) GenitoreEquinoFemmina(Zach, Lea) Zebroide(Zach)	GenitoreEquinoMaschio ⊑ Genitore GenitoreEquinoFemmina ⊑ Genitore CavalloMaschio ⊑ EquinoMaschio CavalloFemmina ⊑ EquinoFemmina Equino ≡ EquinoMaschio ⊔ EquinoFemmina EquinoMaschio ⊓ EquinoFemmina ⊑ ⊥ ⊤ ⊑ ∀GenitoreEquinoMaschio⁻.Equino ⊤ ⊑ ∀GenitoreEquinoFemmina⁻.Equino ⊤ ⊑ ∀GenitoreEquinoMaschio.CavalloMaschio ⊤ ⊑ ∀GenitoreEquinoFemmina.CavalloFemmina Equino ⊑ =2Genitore NonZebraEquino ≡ Equino ⊓ ¬Zebra Zebroide ≡ ∃Genitore.Zebra ⊓ ∃Genitore.NonZebraEquino

Si noti come OWL e le Description Logic adottino la politica **Open World Semantic**, ovvero tutto ció che non é esplicitamente contenuto nella Knowledge Base e non é deducibile dagli assiomi (ovvero, tutto ció su cui non si ha informazione) viene assunto come vero.

Capitolo 3
Problemi di ricerca

3.1 Risolvere problemi con la ricerca

Non é sempre scontato quale debba essere l'azione che permette ad un agente razionale di massimizzare la sua funzione di prestazione. In questo caso, l'agente deve essere in grado di *programmare*: individuare una sequenza di azioni che, intraprese, permettono di raggiungere uno stato obiettivo. Un agente con queste caratteristiche viene chiamato **problem-solver** e la computazione che sottostá all'individuare tale sequenza prende il nome di **ricerca**.
La ricerca puó essere descritta sotto forma di algoritmo. É possibile classificare gli algoritmi in due classi: **informati**, ovvero che operano in un ambiente del quale hanno tutte le informazioni in qualsiasi momento, e **non informati**, dove una (piú o meno) grande parte di queste informazioni non é ottenibile in ogni momento. Un ambiente in cui opera un algoritmo informato é, di norma: accessibile, deterministico, episodico, statico e discreto.
Un problem-solver con a disposizione questo livello di conoscenza sull'ambiente puó allora organizzare il processo di risoluzione del problema in quattro fasi:

1. **Formulazione dell'obiettivo**. L'agente determina quale sia l'obiettivo da perseguire e, di conseguenza, guida il suo operato e le azioni che andrá a compiere in una certa direzione;
2. **Formulazione del problema**. L'agente formula una descrizione degli stati e le azioni necessarie a poter raggiungere tale obiettivo, ovvero un *modello* della parte di ambiente di interesse;
3. **Formulazione della soluzione**. Prima di compiere una qualsiasi azione nel mondo reale, l'agente simula una sequenza di azioni sul modello, fino a trovarne una che gli permette di raggiungere l'obiettivo. Una sequenza con queste caratteristiche viene chiamata **soluzione**. Si noti come l'agente possa dover formulare diverse sequenze che non sono soluzioni prima di riuscire a trovarne una, oppure potrebbe determinare che una soluzione non esiste;
4. **Esecuzione**. Una volta individuata una soluzione (se esiste), l'agente compie, uno alla volta, i passi di cui questa é costituita.

In un ambiente accessibile, deterministico e discreto la soluzione ad ogni problema é una sequenza fissata [1]. Ovvero, una volta che tale soluzione é stata individuata, l'agente puó percorrerne i passi con la consapevolezza che, dall'uno all'altro, non é necessario ricavare percezioni aggiuntive dall'ambiente per rivalutare la soluzione presa. Questo tipo di approccio é chiamato **closed loop**, ed é possibile solamente se l'ambiente possiede le caratteristiche sopra citate. Se l'ambiente fosse inaccessibile, non sarebbe possibile ottenere subito la soluzione per intero. Se l'ambiente fosse sequenziale o non deterministico, l'agente dovrebbe ricalcolare la soluzione ad ogni passo, perché le caratteristiche dell'ambiente sarebbero mutevoli.
Formalmente, é possibile formulare un **problema di ricerca** come segue:

- Un insieme di **stati**, ovvero di *configurazioni* in cui l'ambiente puó trovarsi. Tale insieme viene chiamato **spazio degli stati**, e puó essere sia finito che infinito;
- Uno **stato iniziale**, ovvero lo stato in cui l'agente inizia il suo operato;
- Uno o piú **stati obiettivo**, ovvero stati in cui il problema é risolto una volta che l'ambiente si trova in uno di questi. Se gli stati obiettivo sono piú di uno, allora si assume che il problema sia risolto a prescindere da quale di questi si raggiunge;
- Le **azioni** che l'agente puó compiere. Queste possono dipendere dallo stato in cui l'agente si trova oppure possono essere eseguite a prescindere. Dato uno stato s, la funzione $\text{ACTIONS}(s)$ restituisce l'insieme di azioni che l'agente puó compiere se si trova in s. Ciascuna di queste azioni si dice **applicabile** in s;
- Una **funzione di transizione**, che descrive l'effetto che l'eseguire ciascuna azione comporta. Il cambiamento di stato, da uno stato di partenza ad uno stato di arrivo, per mezzo di una certa azione, prende il nome di **transizione**. Dato uno stato s ed una azione a, la funzione $\text{RESULT}(s, a)$ restituisce lo stato che viene raggiunto se viene eseguita a mentre ci si trova in s;
- Opzionalmente, una **funzione di costo**, che associa un valore numerico a ciascuna transizione. Dati due stati s e s' ed una azione a, la funzione $\text{ACTION-COST}(s, a, s')$ restituisce il *costo* che comporta il passare da s a s' applicando a. La funzione di costo rappresenta il "disincentivo" che l'agente ha nel compiere una determinata azione, per quanto possa essere necessaria a raggiungere lo stato obiettivo. Un problema di ricerca in cui la funzione di costo non é specificata puó essere pensato come avente una funzione di costo che assegna il medesimo costo a tutte le azioni [2].

1. Nonostante questa situazione sembri irrealistica, diversi ambienti reali possono essere modellati in questo modo.
2. Da questo momento in poi, se non riportato esplicitamente, si assume di stare trattando problemi dove la funzione di ricerca non é definita.

Una qualsiasi sequenza di azioni forma un **percorso**; una soluzione non é altro che un percorso avente uno degli stati obiettivo come ultimo stato. A partire da uno stesso ambiente e da un agente capace di compiere le stesse azioni, cambiando gli stati obiettivo si ottiene un problema completamente diverso.
Talvolta si ha interesse non a ricavare una soluzione qualsiasi, ma bensí una soluzione che rispecchia determinate caratteristiche, detta **soluzione ottimale**. Se la funzione di costo é assente, la soluzione ottimale é in genere quella composta dal minimo numero di azioni. Se invece é presente una funzione di costo, la soluzione ottimale é quella che ha il costo complessivo piú piccolo: tale costo é dato dalla somma, eventualmente pesata, dei costi che comporta ciascuna transizione che avviene nel percorso. Anche le soluzioni ottimali possono non essere univoche.

Si consideri una versione semplificata del gioco del Pacman, dove Pacman si trova in una griglia 4x4. Ciascuna cella della griglia é identificata da una coppia di coordinate, numerate da 1 a 4 lungo gli assi. Inizialmente, Pacman si trova nella cella in basso a sinistra, ovvero la cella (1, 1). A Pacman é permesso muoversi di una cella alla volta lungo i quattro assi cardinali, fintanto che non si muove in celle che si trovano all'interno della griglia. A partire da questa situazione, é possibile costruire diversi problemi di ricerca.
Si consideri un primo problema in cui l'obiettivo é raggiungere la cella in alto a destra, ovvero la cella (4, 4). Informalmente, il problema é cosí definito:

- Spazio degli stati: (1, 1), (1, 2), (1, 3), (1, 4), … , (4, 3), (4, 4)
- Stato iniziale: (1, 1)
- Stato obiettivo: (4, 4)
- La funzione ACTIONS ha in input una coppia di coordinate e restituisce le direzioni in cui Pacman puó muoversi senza uscire dalla griglia. Ad esempio, ACTIONS(2, 4) restituisce {E, W, S}
- La funzione RESULT ha in input una coppia di coordinate ed una azione e restituisce una nuova coppia di coordinate. Ad esempio RESULT(2, 4, W) restituisce (1, 4)

Una possibile soluzione potrebbe essere il percorso:

$$(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (2, 2) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 4) \rightarrow (4, 4)$$

Tuttavia, tale soluzione non é ottimale. Infatti, una possibile soluzione ottimale é:

$$(1, 1) \rightarrow (2, 1) \rightarrow (2, 2) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 4) \rightarrow (4, 4)$$

Si consideri un secondo problema, in cui l'obiettivo é quello di raccogliere tutte le pillole. In questo caso, ciascuna cella della griglia é identificata sia da una coppia di coordinate che da un valore booleano, che indica se la cella contiene o non contiene una pillola.
Lo spazio degli stati di questa formulazione del problema é nettamente piú complessa, perché é necessario sia tenere traccia della posizione di Pacman in ogni istante sia delle pillole raccolte. Inoltre, non é piú rilevante la cella in cui Pacman si trova dopo aver raccolto tutte le pillole.

- Spazio degli stati:

```
((1, 1, F), (1, 2, T), (1, 3, T), …, (4, 3, T), (4, 4, T)), (1, 1)
((1, 1, F), (1, 2, F), (1, 3, T), …, (4, 3, T), (4, 4, T)), (1, 2)
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, T), (4, 4, T)), (1, 2)
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, T), (4, 4, T)), (1, 3)
[…]
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, T)), (4, 3)
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, F)), (4, 4)
[…]
```

- Stato iniziale:

```
(1, 1, F), (1, 2, T), (1, 3, T), …, (4, 3, T), (4, 4, T)
```

- Stati obiettivo:

```
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, F)), (1, 1)
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, F)), (1, 2)
[…]
((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, F)), (4, 4)
```

- La funzione ACTIONS é analoga alla precedente
- La funzione RESULT ha in input uno stato ed una azione e restituisce un nuovo stato in cui sia le coordinate sia il valore booleano delle celle viene aggiornato. Se la cella raggiunta aveva valore T , viene cambiato in F ; se aveva valore F , rimane invariato. Ad esempio:

```
RESULT(((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, T)), (4, 3), E) =
       ((1, 1, F), (1, 2, F), (1, 3, F), …, (4, 3, F), (4, 4, F)), (4, 4)
```

Si noti come una formulazione di questo tipo, in cui l'intero stato viene passato come argomento di RESULT , sarebbe estremamente inefficiente se implementata sotto forma di codice. Tuttavia, da un punto di vista strettamente matematico, la rappresentazione é corretta.

Lo spazio degli stati puó venire rappresentato sotto forma di grafo, detto **state space graph**: i nodi del grafo corrispondono agli stati, gli archi corrispondono alle azioni che permettono di passare da uno stato all'altro ed il costo delle azioni é, se il problema ha associato una funzione di costo, l'etichetta dell'arco.
Se lo spazio degli stati é un insieme finito, allora lo state space graph conterrá esattamente tanti nodi quanti sono gli stati; se lo spazio degli stati é un insieme infinito, non vi é modo di costruire il grafo per intero. Si noti peró come, in genere, uno space state graph é troppo grande per essere rappresentato per intero, anche se il numero di stati é finito; é invece preferibile costruire il grafo on-the-fly, sulla base di quali nodi é necessario rappresentare.

3.2 Algoritmi di ricerca

Prende il nome di **algoritmo di ricerca** un algoritmo che, avendo in input un problema di ricerca con una data istanza, restituisce in output una soluzione a tale problema se (almeno) una soluzione esiste, oppure un errore se non esiste alcuna soluzione.
In termini molto generali, é possibile descrivere un algoritmo di ricerca in questo modo. Viene innanzitutto creato l'insieme `PartialSolutionSet`, che inizialmente contiene il solo stato iniziale. Fintanto che l'insieme `PartialSolutionSet` non é vuoto, l'algoritmo cerca di trovare una soluzione; se l'insieme si svuota, allora una soluzione non esiste.
Il corpo del ciclo principale inizia estraendo uno dei percorsi di `PartialSolutionSet` ed analizzando l'ultimo stato di tale percorso. Per ciascuna azione applicabile in tale stato, viene creato un insieme `Successors` che contiene tutti gli stati raggiungibili a partire da tale stato compiendo tali azioni. Per ciascuno degli stati cosí raggiunti, si costruisce una `potentialSolution` accodando tale stato al percorso corrente e si valuta se lo stato in questione é uno stato obiettivo: se lo é, allora `potentialSolution` é effettivamente una soluzione e l'algoritmo la restituisce, altrimenti `potentialSolution` viene aggiunta a `PartialSolutionSet`.

```
procedure GENERIC-SEARCH(initialState, strategy)
    PartialSolutionSet = [initialState]
    while (PartialSolutionSet ≠ []) do
        path ← CHOOSE-PATH(PartialSolutionSet, strategy)
        lastState ← path[-1]
        Successors ← []
        foreach action in ACTIONS(lastState) do
            Successors ← Successors ∪ {RESULT(lastState, action)}
        foreach state in Successors do
            potentialSolution ← path + state
            if (state.type = "goal") then
                return potentialSolution
            else
                PartialSolutionSet ← INSERT-PATH(PartialSolutionSet, potentialSolution, strategy)
    return "No solution found"
```

La funzione `CHOOSE-PATH` determina quale dei percorsi finora costruiti debba essere quella da analizzare nell'iterazione corrente, mentre la funzione `INSERT-PATH` inserisce il nuovo percorso appena costruito in una determinata posizione di `PartialSolutionSet`. Entrambe dipendono da una certa **strategia**, ovvero da un set di regole usate per determinare la scelta. L'adottare una strategia piuttosto che un'altra influisce notevolmente sulle prestazioni dell'algoritmo, perché prima un percorso che si rivelerá poi essere una soluzione viene analizzato e prima l'algoritmo termina (naturalmente, l'algoritmo non puó sapere in anticipo se il percorso in analisi é oppure non é una soluzione, altrimenti il problema non si porrebbe proprio).
L'algoritmo ha due cicli for innestati, pertanto il tempo di esecuzione é approssimativamente quadratico nel numero degli stati. Ogni volta che all'algoritmo viene passato in input uno stato, questo ricalcola (se esiste) una soluzione che abbia quello stato come stato iniziale. Un algoritmo di questo tipo rientra nella categoria degli agenti guidati da modello e basati su obiettivi.
Si noti come non esiste garanzia che l'algoritmo termini, perché é del tutto ammissibile poter ritornare ad uno stato giá visitato in precedenza, e l'algoritmo potrebbe bloccarsi in un loop infinito.
Ci si chiede se é possibile costruire un algoritmo che rientri nella categoria degli agenti semplici. Tecnicamente, sarebbe possibile utilizzare l'algoritmo per costruire una sorta di lookup table, dove a ciascuno stato iniziale é associata la relativa soluzione; in questo modo, sarebbe necessario richiamare l'algoritmo una sola volta per ciascuno stato, dopodiché la soluzione per uno stato giá passato in input verrebbe restituita immediatamente. Difficilmente questo approccio potrebbe funzionare, perché un problema reale ha un numero di stati troppo grande; una singola esecuzione dell'algoritmo ripetuta per ogni possibile stato iniziale sarebbe comunque insostenibile. Inoltre, l'algoritmo presuppone che sia possibile ottenere tutte le informazioni dall'ambiente, ma questo é vero solamente se l'ambiente é accessibile.

Nella prima versione del problema di Pacman dell'esempio precedente, un approccio di questo tipo é effettivamente possibile. Essendo il numero totale di stati 16, a ciascuno di questi é possibile associare uno dei possibili percorsi ottimali in una tabella di questo tipo:

Stato iniziale	**Soluzione**
(1, 1)	(1, 1) → (2, 1) → (2, 2) → (2, 3) → (3, 3) → (3, 4) → (4, 4)
(2, 1)	(2, 1) → (2, 2) → (2, 3) → (3, 3) → (3, 4) → (4, 4)
(3, 1)	(3, 1) → (3, 2) → (3, 3) → (3, 4) → (4, 4)
…	…
(3, 4)	(3, 4) → (4, 4)
(4, 4)	(4, 4)

Applicare un approccio simile alla seconda versione del problema diviene invece del tutto irrealistico, dato che il numero totale di stati é troppo grande.

Lo space state graph non si adatta bene a rappresentare le soluzioni fornite da GENERIC-SEARCH , perché i percorsi e le soluzioni sono difficili da evidenziare esplicitamente. Una rappresentazione migliore é quella offerta da una struttura ad albero detta **albero di ricerca**; ciascun nodo corrisponde ad uno degli stati nello spazio degli stati, mentre gli archi corrispondono alle azioni che costituiscono le transizioni. La radice dell'albero corrisponde allo stato iniziale del problema.
A differenza dello state space graph, dove ogni nodo é univocamente uno stato, nell'albero di ricerca diversi nodi possono riferirsi allo stesso stato. Questo rende gli alberi di ricerca meno efficienti in termini di spazio occupato, perché questi avranno un numero di nodi pari o superiore allo state space graph equivalente. Il vantaggio degli alberi di ricerca é che per identificare una soluzione é sufficiente seguire un percorso che va dal nodo iniziale ad un nodo foglia che ha associato uno stato obiettivo.
É possibile tradurre l'algoritmo di ricerca generico in termini di alberi. L'albero, `Tree` , inizialmente contiene il solo nodo radice, cosí come l'insieme `Leaves` , detto **frontiera**, che riporta tutti i nodi foglia. Fintanto che l'insieme `Leaves` non é vuoto, l'algoritmo cerca di trovare una soluzione; se l'insieme si svuota, allora una soluzione non esiste. Ciascun nodo puó essere pensato come una struttura a due campi: un campo `parent` che indica il rispettivo nodo genitore ed un campo `state` che indica lo stato a cui il nodo é associato. `Leaves` é, in genere, implementato con una coda, mentre `Tree` puó essere una qualsiasi rappresentazione usata per gli alberi.
Il corpo del ciclo principale inizia estraendo `nodeToExpand` , uno dei nodi in `Leaves` , attraverso una determinata strategia; il nodo viene rimosso da `Leaves` . Viene poi analizzato lo stato associato a tale nodo, ovveor `nodeToExpand.state` : se questo é uno stato obiettivo, allora si ricostruisce una soluzione salendo di nodo in nodo attraverso il campo `parent` fino a trovare la radice, che é identificabile univocamente essendo l'unico nodo ad avere `NULL` come valore per questo campo.
Se invece `nodeToExpand.state` non é uno stato obiettivo, viene applicata a `nodeToExpand` l'operazione di **espansione**. Questa consiste nel costruire tanti nodi `newNode` per ciascuna azione applicabile in `nodeToExpand.state` , ciascuno avente tale azione come campo `action` e avente `nodeToExpand` come campo `parent` . Tale nodo viene poi aggiunto a `Leaves` , mentre a `Tree` viene aggiunta una tupla che ha `nodeToExpand` e `newNode` come elementi, che rappresenta la transizione compiuta dagli stati riferiti ai due nodi.

```
procedure GENERIC-TREE-SEARCH(initialState, strategy)
    root.parent ← NULL
    root.state ← initialState
    Leaves ← [root]
    Tree ← [root]


    while (Leaves ≠ []) do
     nodeToExpand ← EXTRACT-LEAF(Leaves, strategy)
     if (nodeToExpand.state.type = "goal") then
         thisNode = nodeToExpand
         Solution = []
         do
            Solution ← Solution + thisNode
          thisNode ← thisNode.parent
        while (thisNode.parent ≠ NULL)
        return Solution
 else
        foreach action in ACTIONS(nodeToExpand.state) do
            newLeaf.state ← RESULT(nodeToExpand.state, action)
            newLeaf.parent ← nodeToExpand
            Leaves ← ADD-TO-QUEUE(Leaves, newLeaf, strategy)
            Tree ← Tree ∪ (nodeToExpand, newLeaf)


return "No solution found"
```

GENERIC-SEARCH e GENERIC-TREE-SEARCH sono di fatto equivalenti. Il vantaggio di GENERIC-TREE-SEARCH é che utilizza una rappresentazione "standardizzata", quella ad albero, che permette di utilizzare strutture dati note per tenere traccia delle computazioni (GENERIC-SEARCH non specifica che strutture dati utilizzare). Assumendo che la frontiera sia implementata per mezzo di una coda, `strategy` determina solamente il tipo di coda in questione e qual'é il nodo scelto di volta in volta per essere espanso.
L'algoritmo verrebbe ulteriormente ottimizzato osservando come l'espansione di uno stesso nodo potrebbe ripresentarsi piú volte. A tale scopo, é possibile tenere traccia di tutti i nodi giá espansi salvando il risultato dell'espansione in una struttura dati a parte, `expanded` , ed utilizzarla per

velocizzare le computazioni successive. In questo modo, quando viene trovato un nodo da espandere, prima si controlla se il risultato dell'espansione é giá presente in `expanded` ; se lo é, si estrae da questa il risultato, altrimenti il nodo viene espanso normalmente e si salva in `expanded` il risultato.
Le diverse incarnazioni di `GENERIC-TREE-SEARCH` vengono valutate sulla base di quattro metriche:

- **Completezza**. Un algoritmo di ricerca si dice **completo** se garantisce, per qualsiasi istanza, di fornire una risposta, a prescindere che questa sia affermativa (soluzione trovata) o negativa (soluzione non trovata). Inoltre, se almeno una soluzione esiste, l'algoritmo deve essere in grado di trovarla;
- **Ottimalitá**. Un algoritmo di ricerca si dice **ottimale** se garantisce di trovare una soluzione ottima;
- **Complessitá in tempo**, nel caso peggiore;
- **Complessitá in spazio**, nel caso peggiore.

La completezza di un algoritmo di ricerca é influenzata (anche) dallo spazio di stati su cui questo opera. Se lo spazio di stati in questione é finito, allora l'algoritmo di ricerca sará certamente completo, perché prima o poi tutti i nodi verranno raggiunti ed espansi. Se lo spazio di stati é infinito ma é noto che almeno una soluzione esista, un algoritmo di ricerca *potrebbe* comunque essere completo a seconda di come é stato implementato. Se invece lo spazio di stati é infinito e non esiste nessuna soluzione, allora qualsiasi algoritmo di ricerca non sará mai completo, perché su qualsiasi istanza del problema incorrerrá in un loop infinito.
In genere, le prestazioni di un algoritmo che opera su un grafo sono espresse in termini della cardinalitá del suo insieme di archi e della cardinalitá del suo insieme di vertici. Questa é la scelta migliore nel caso in cui il grafo sia **esplicito**, ovvero dove é effettivamente rappresentato sotto forma di stati e di archi.
Gli algoritmi di ricerca operano invece su un grafo "indotto", **implicito**, dove la loro dimensione non è nota a priori. Per questo motivo, si preferisce valutare le prestazioni dell'algoritmo in termini di **profonditá**, indicato con m, e **branching factor**, indicato con b: la prima indica massimo numero di nodi che puó andare a costituire una soluzione, mentre la seconda indica il massimo numero di successori che ciascun nodo puó avere.
Idealmente é possibile visualizzare questi parametri "inscrivendo" l'albero in un triangolo avente altezza m e base b. Se ogni nodo puó avere al piú b nodi successori, al primo livello dell'albero vi saranno b^0 nodi (la sola radice), al secondo livello dell'albero vi saranno b^1 nodi, al terzo livello dell'albero b^2 nodi, ecc ... Questo significa che il numero di nodi dell'intero albero di ricerca é certamente non superiore a $O(b^m)$.

3.3 Ricerca non informata

Un algoritmo di ricerca **non informato** non possiede informazioni in merito a "quanto vicino" sia uno stato rispetto agli obiettivi.

3.3.1 Backtracking Search

Backtracking Search é un algoritmo di ricerca dove la strategia usata consiste essenzialmente nell'espandere sempre il nodo piú profondo.
Nello specifico, Backtracking Search espande prima la radice, poi sceglie il primo nodo cosí generato e lo espande, dopodiché sceglie il primo nodo da questo generato e lo espande, ecc ... Se viene raggiunto un nodo il cui relativo stato é uno stato obiettivo, l'algoritmo termina restituendo il percorso costruito. Se viene invece raggiunto un nodo il cui stato non é uno stato obiettivo ma che non é piú possibile espandere, l'algoritmo opera un **backtracking**, ovvero "ritorna" al primo nodo lungo il percorso che non é stato ancora interamente espanso.

```
// Recursive, high level
procedure BACKTRACKING-SEARCH(s, path)
    if (IS-END(s)) then
        update bestPath
    foreach a in ACTIONS(s) do
        Extend path with SUCCESSOR(s, a)
        BACKTRACKING-SEARCH(SUCCESSOR(s, a), path)
    return bestPath
```

```
// Iterative, based on GENERIC-TREE-SEARCH
procedure BACKTRACKING-SEARCH(initialState)
    root.parent ← NULL
    root.state ← initialState
    Leaves ← [root]
    Tree ← [root]
    Solutions ← []


    while (Leaves ≠ []) do
        nodeToExpand ← POP(Leaves)
        if (nodeToExpand.state.type = "goal") then
            thisNode = nodeToExpand
            thisSolution = []
            do
                thisSolution ← thisSolution + thisNode
                thisNode ← thisNode.parent
            while (thisNode.parent ≠ NULL)
            Solutions ← Solutions ∪ {thisSolution}
        else
            foreach action in ACTIONS(nodeToExpand.state) do
                newLeaf.state ← RESULT(nodeToExpand.state, action)
                newLeaf.parent ← nodeToExpand
                PUSH(Leaves, newLeaf)
                Tree ← Tree ∪ {(nodeToExpand, newLeaf)}


if (Solutions = []) then
    return "No solution found"
else
    bestSolution ← Solutions[0]
    foreach potentialSolution in Solutions do
        if (|potentialSolution| < |bestSolution|) then
            bestSolution ← potentialSolution
    return bestSolution
```

Per quanto riguarda il tempo di esecuzione dell'algoritmo, si osservi come, nel caso peggiore, la soluzione ottimale venga trovata nell'ultimo nodo espanso dell'albero. Infatti, non c'é modo di sapere, una volta trovata una soluzione, se esista una soluzione migliore. Questo significa che

Backtracking Search necessita di espandere l'albero per intero, e quindi il suo tempo di esecuzione sia $O(b^m)$, assumendo che non si verifichino cicli.
Per quanto riguarda lo spazio occupato, Backtracking Search necessita di memorizzare esclusivamente la frontiera dell'albero e tutte le soluzioni parziali. Dato che la frontiera dell'albero non puó essere superiore al branching factor, e dato che é necessario memorizzare al piú m frontiere distinte, la complessitá in termini di spazio di Backtracking Search é $O(bm)$.
Se lo spazio degli stati é finito, Backtracking Search é completo fintanto che non esistono cicli; sebbene possa esplorare gli stessi stati piú volte negli stessi percorsi, prima o poi tutti gli stati vengono raggiunti. Se sono presenti dei cicli, l'algoritmo potrebbe rimanere bloccato in un loop infiniti. Se lo spazio degli stati é infinito, l'algoritmo potrebbe rimanere bloccato nell'espandere lo stesso percorso indefinitamente, anche se non sono presenti cicli.
Un possibile modo per garantire la completezza di Backtracking Search consiste nel fissare una profonditá massima D, oltre la quale l'algoritmo compie backtracking a prescindere dal nodo in esame. Tuttavia, cosí facendo, l'algoritmo non é piú in grado di garantire la correttezza, perché tutte le soluzioni che si trovano ad una profonditá maggiore di D sono perdute. Un approccio di questo tipo é preferibile solamente se la natura del problema permette di conoscere in anticipo che non puó esistere una soluzione piú in profonditá di D, e nella maggior parte dei casi reali la profonditá massima é sconosciuta fino alla risoluzione del problema.

```
procedure BOUNDED-BACKTRACKING-SEARCH(s, path, D)
    if (|path| > D) then
        return NULL
    if (IS-END(s)) then
        update bestPath
    foreach a in ACTIONS(s) do
        Extend path with SUCCESSOR(s, a)
        BOUNDED-BACKTRACKING-SEARCH(SUCCESSOR(s, a), path, D)
    return bestPath
```

In questo caso, essendo D necessariamente inferiore a m, é ragionevole esprimere la complessitá in termini di tempo e di spazio in funzione di D, rispettivamente $O(b^D)$ e $O(bD)$.

3.3.2 Depth-First Search

Una variante di Backtracking Search é **Depth-First Search**, o **DFS**, che opera con la stessa strategia ma si interrompe immediatamente appena viene trovata una soluzione. Come struttura dati atta a contenere i nodi della frontiera é bene scegliere una coda LIFO. Questo perché i nuovi nodi che vengono aggiunti, che si trovano necessariamente dopo i nodi che li hanno generati, vengono posti prima di questi ultimi.

```
// Recursive, high level
procedure DEPTH-FIRST-SEARCH(s, path, D)
    if (|path| > D) then
        return NULL
    if (IS-END(s)) then
        return path
    foreach a in ACTIONS(s) do
        Extend path with SUCCESSOR(s, a)
        DEPTH-FIRST-SEARCH(SUCCESSOR(s, a), path, D)
```

```
// Iterative, based on GENERIC-TREE-SEARCH
procedure DEPTH-FIRST-SEARCH(initialState)
    root.parent ← NULL
    root.state ← initialState
    Leaves ← [root]
    Tree ← [root]

    while (Leaves ≠ []) do
        nodeToExpand ← POP(Leaves)
        if (nodeToExpand.state.type = "goal") then
            thisNode = nodeToExpand
            Solution = []
            do
                Solution ← Solution + thisNode
                thisNode ← thisNode.parent
            while (thisNode.parent ≠ NULL)
            return Solution
        else
            foreach action in ACTIONS(nodeToExpand.state) do
                newLeaf.state ← RESULT(nodeToExpand.state, action)
                newLeaf.parent ← nodeToExpand
                PUSH(Leaves, newLeaf)
                Tree ← Tree ∪ (nodeToExpand, newLeaf)

    return "No solution found"
```

La soluzione restituita da DFS é la prima che viene trovata, ma non vi é alcuna garanzia che questa sia una soluzione ottimale. Infatti, potrebbe esserci una soluzione migliore di quella trovata lungo i nodi lasciati inesplorati, ma questi non verranno mai raggiunti. Fintanto che lo spazio degli stati é finito e fintanto che le soluzioni si trovano a meno profonditá di D, DFS é comunque completo, perché una soluzione (per quanto non necessariamente ottimale) verrá sempre trovata.
DFS ha peró il vantaggio di dover tenere traccia solamente del percorso in esame e dei vari punti di scelta, non di tutti i percorsi finora trovati. Inoltre, sebbene il tempo di esecuzione teorico nel caso peggiore sia comunque $O(b^D)$, nella pratica questo tende ad essere molto inferiore, perché una soluzione viene in genere trovata molto prima di esplorare l'albero per intero.
Ci si chiede se sia possibile estendere DFS per renderlo immune alla presenza dei cicli. Si osservi come, nell'espandere un certo nodo, si conoscano giá tutti i nodi progenitori del nodo in esame e di conseguenza tutti gli stati a cui questi si riferiscono. Pertanto, se si tenta di espandere un nodo che si riferisce allo stesso stato di un nodo suo progenitore, allora si ha la certezza che quell'espansione condurrá ad un ciclo.
Pertanto, un possibile approccio consisterebbe nell'inserire un controllo all'interno di `DEPTH-FIRST-SEARCH` che, prima di espandere un nodo, controlla ricorsivamente in tutti i nodi precedenti per verificare che lo stato associato a ciascuno di questi sia distinto dallo stato associato al nodo in esame. Se esiste almeno un nodo con queste caratteristiche, allora viene eseguito backtracking immediatamente, tornando a prima che venisse fatta la scelta di tale nodo.
Questo approccio é certamente corretto, ma richiede un numero di controlli che cresce con la profonditá: al primo livello serve controllare solo il nodo precedente, al secondo livello i due nodi precedenti, ecc ... Per prevenire questa penalitá in termini di tempo di esecuzione, é possibile

salvare il percorso finora costruito non nel solo albero ma anche in una struttura dati di supporto che permetta un accesso ai suoi membri in tempo costante, come ad esempio una hash table.

3.3.3 Breadth-First Search

Breadth-First Search, o **BFS**, é un algoritmo di ricerca dove la strategia usata consiste essenzialmente nell'espandere sempre il nodo meno profondo.
Nello specifico, BFS espande prima la radice, poi sceglie il primo nodo cosí generato e lo espande, dopodiché espande il secondo nodo cosí generato, fino ad espandere tutti i successori della radice. A questo punto, opera backtracking ed espande i nodi del livello successivo. L'algoritmo termina quando si tenta di espandere un nodo con associato uno stato obiettivo o quando non é piú possibile espandere alcun nodo.
Come struttura dati atta a contenere i nodi della frontiera é bene scegliere una coda FIFO. Questo perché i nuovi nodi che vengono aggiunti, che si trovano necessariamente dopo i nodi che li hanno generati, vengono posti in fondo alla coda, mentre i nodi giá nella coda, che sono stati quindi aggiunti prima, vengono espansi prima.

```
procedure BREADTH-FIRST-SEARCH(initialState)
    root.parent ← NULL
    root.state ← initialState
    Leaves ← [root]
    Tree ← [root]

    while (Leaves ≠ []) do
        nodeToExpand ← HEAD(Leaves)
    if (nodeToExpand.state.type = "goal") then
        thisNode = nodeToExpand
        Solution = []
        do
            Solution ← Solution + thisNode
            thisNode ← thisNode.parent
        while (thisNode.parent ≠ NULL)
        return Solution
    else
        foreach action in ACTIONS(nodeToExpand.state) do
            newLeaf.state ← RESULT(nodeToExpand.state, action)
            newLeaf.parent ← nodeToExpand
            APPEND(Leaves, newLeaf)
            Tree ← Tree ∪ (nodeToExpand, newLeaf)

return "No solution found"
```

BFS, nonostante restituisca immediatamente la prima soluzione che trova, é comunque un algoritmo ottimale, perché la soluzione ottenuta é necessariamente quella costituita dal minimo numero di azioni. Infatti, quando viene esplorato un nodo alla profonditá d, tutti i nodi a profonditá $d-1$, $d-2$, ecc ... sono giá stati espansi; se uno di questi nodi avesse contenuto uno stato obiettivo, sarebbe giá stato trovato. Si noti peró come questo sia vero solamente se il problema di ricerca non ha associata una funzione di costo, perché altrimenti la soluzione costituita dal minimo numero di azioni potrebbe non essere quella che minimizza il costo totale.
Se lo spazio di stati é finito ed almeno una soluzione esiste, BFS é un algoritmo completo, perché prima o poi tutti i nodi verranno raggiunti ed é garantito che non possa verificarsi un ciclo. Se lo spazio di stati é infinito ma esiste comunque almeno una soluzione, BFS mantiene comunque la proprietá di completezza.
Per quanto riguarda il tempo di esecuzione dell'algoritmo, si osservi come, nel caso peggiore, la soluzione ottimale (che é anche la prima soluzione trovata) venga trovata nell'ultimo nodo espanso dell'albero. Tuttavia, nella pratica é possibile assumere che la soluzione ottimale venga trovata ad una profonditá s, prima di raggiungere il fondo. Dato che BFS necessita di espandere interamente l'albero fino ad s, il suo tempo di esecuzione sia $O(b^s)$.
Per quanto riguarda lo spazio occupato, BFS necessita di memorizzare tutti i nodi espansi fino al nodo attuale. Assumendo nuovamente che la profonditá dell'ultimo nodo sia s, dato che ogni nodo genera a sua volta al piú b nodi la complessitá in termini di spazio di BFS é ancora $O(b^s)$.
BFS si rivela migliore di DFS nel caso in cui le soluzioni sono poche e si trovano a bassa profonditá. DFS si rivela migliore di BFS quando le soluzioni sono molte e si trovano molto in profonditá, perché aumenta la probabilitá di trovarne una. Inoltre, DFS é preferibile nelle situazioni in cui non si ha interesse a trovare la soluzione ottimale, ma é sufficiente trovarne una qualsiasi.

3.3.4 Iterative-Deepening Search

Iterative-Deepening Search, o **IDS**, é una variante di DFS che ne combina i vantaggi con la completezza di BFS. IDS opera come DFS, ma la profonditá massima aumenta ad ogni iterazione. Nello specifico, IDS opera DFS con profonditá massima 0 (sulla sola radice), se trova una soluzione la restituisce, altrimenti ricomincia da capo con profonditá massima 1, e continua in questo modo fino a che una soluzione viene trovata.
IDS é un algoritmo ottimale fintanto che il problema di ricerca su cui é applicato non ha associata una funzione di costo, perché come in BFS se una soluzione fosse esistita ad una profonditá maggiore di quella in esame in un qualsiasi momento allora sarebbe giá stata trovata. É inoltre, come DFS, un algoritmo completo fintanto che non si presentano cicli (che é comunque possibile individuare e prevenire).
Il costo in termini di tempo é il medesimo di DFS, ovvero $O(b^d)$, perché di fatto consiste nell'applicare tale algoritmo per d volte. Inoltre, il costo in termini di spazio é comunque $O(bd)$, perché in ogni iterazione dell'algoritmo é necessario espandere solamente il sottoalbero in esame, e tutte le computazioni fatte nelle iterazioni precedenti possono essere scartate. Infatti, dato che la dimensione dell'albero cresce esponenzialmente di livello in livello, l'unico costo effettivamente rilevante é quello relativo all'ultimo albero, ovvero $O(bd)$, perché sará infinitamente maggiore del costo necessario a costruire gli alberi fino ai livelli precedenti.

3.3.5 Uniform Cost Search

Uniform Cost Search [3], o **UCS**, é un algoritmo di ricerca pensato per problemi con associata una funzione di costo. Questo consiste essenzialmente nell'espandere sempre il nodo che ha il minimizza il costo complessivo nel percorso dalla radice al nodo stesso.
Nello specifico, UCS calcola per ogni nodo ad ogni iterazione il valore di $g(n)$, ovvero il costo complessivo che si avrebbe nel raggiungere n a partire dalla radice, e sceglie di espandere il nodo che minimizza $g(n)$. La frontiera viene salvata in un coda di prioritá ordinata in ordine decrescente rispetto a $g(n)$, di modo che il nodo estratto sia sempre quello che minimizza tale funzione.
La complessità in tempo di UCS è espressa in termini di C^*, il costo di una soluzione ottimale, e di ε, il limite inferiore al costo di una singola azione. Si osservi infatti come UCS espanda tutti i nodi che hanno come costo complessivo inferiore alla soluzione ottimale: approssimativamente, la profondità di tale soluzione sarà C^* / ε. Dato che UCS adotta una strategia simile a BFS, il tempo di esecuzione sarà $O\left(b^{C^*/\varepsilon}\right)$. Si noti infatti che, se tutte le azioni hanno il medesimo costo, $(C^* / \varepsilon) \approx d$, e UCS diventa indistinguibile da BFS. Inoltre, l'algoritmo deve memorizzare l'intera frontiera, la cui dimensione è approssimativamente C^* / ε; per questo motivo, la complessità in termini di spazio è $O\left(b^{C^*/\varepsilon}\right)$.
É facile verificare come UCS sia un algoritmo completo. Inoltre, se i costi delle azioni sono tutti positivi, é un algoritmo ottimale, perché la soluzione trovata é necessariamente quella che minimizza la funzione di costo.

3.4 Ricerca informata

Un algoritmo di ricerca **informato** possiede informazioni in merito a "quanto vicino" sia uno stato rispetto agli obiettivi. A differenza degli algoritmi di ricerca non informati, che procedono espandensosi in ogni direzione, gli algoritmi di ricerca informati possono orientare la loro computazione verso una determinata direzione.
Viene detta **funzione di euristica**, o semplicemente **euristica**, una funzione che fornisce informazioni piú o meno precise su quanto lo stato generico di un problema sia vicino ad uno stato obiettivo del medesimo problema. Nello specifico: tale funzione, indicata con $h(n)$, restituisce una stima numerica del percorso a costo minimo che ha inizio nel nodo n e ha fine nel nodo con stato obiettivo piú vicino. Si noti come una euristica indichi semplicemente una ipotetica direzione "privilegiata" da seguire, ma non garantisca in alcun modo che seguendo tale direzione si raggiunga la soluzione con il minimo costo.

> Si estenda il problema dell'esempio precedente introducendo degli ostacoli lungo il percorso. In questo modo, da ciascuna cella della griglia è possibile muoversi solamente in specifiche direzioni. Questo significa che il problema di raggiungere la cella (4,4) non ha più per soluzione quella presentata, perchè potrebbe non essere più valida. Tuttavia, è garantito che qualsiasi soluzione non possa essere migliore, in termini di costo, di quella ipotetica presentata. Per questo motivo, Il numero di celle fra quella attuale e quella obiettivo è una possibile euristica per il problema di ricerca.

I modi per costruire una euristica sono molteplici. Il piu semplice consiste nel costruire una versione "semplificata" del problema in esame, detta **problema rilassato**, analogo in tutto al problema originale meno che nella funzione di transizione. Questa viene infatti ampliata aggiungendo azioni che permettono di raggiungere in meno passi lo stato obiettivo da qualsiasi stato. In questo modo, si ottiene un limite inferiore al numero di azioni necessarie a raggiungere uno stato obiettivo nel problema originario; tale limite inferiore diviene la funzione di euristica.

> Cio che e stato fatto nel precedente esempio e un esempio di costruzione di un problema rilassato, introducendo azioni aggiuntive che permettono a Pacman di muoversi ignorando gli ostacoli ed individuando un limite inferiore al numero di azioni necessarie.

Una euristica si dice **ammissibile** se approssima sempre i costi per difetto. Ovvero, sia $h^*(n)$ l'effettivo costo minimo necessario a raggiungere il piu vicino stato obiettivo per un nodo n; una euristica $h(n)$ e ammissibile se vale $h(n) \leq h^*(n)$ per qualsiasi n.
Una euristica $h(n)$ si dice **consistente** se, per ogni nodo n e per ogni successore n' di n generato dall'azione a, vale:

$$h(n) \leq \text{COST}(n, a, n') + h(n')$$

Che non é altro che la disuguaglianza triangolare rispetto all'euristica.
La proprietá di consistenza é piú forte dell'ammissibilitá, perché ogni euristica consistente é anche ammissibile, ma non tutte le euristiche ammissibili sono consistenti. Inoltre, se l'euristica é consistente, quando viene raggiunto un nodo per la prima volta si ha la certezza che questo si trovi su uno dei percorsi ottimali, e non verrá mai aggiunto alla frontiera piú di una volta.
Date piú euristiche ammissibili per il medesimo problema, é possibile compararle per valutare quale sia la migliore. Se date due euristiche h_1 e h_2 vale $h_1(n) \geq h_2(n)$ per ogni valore di n, si dice che h_1 **domina** h_2. Se l'euristica h_1 domina h_2, allora h_1 é sempre una euristica migliore di h_2, perché il bound restituito da h_1 sará sempre maggiore di quello restituito da h_2, e sará quindi piú vicino all'effettivo valore della soluzione ottimale.
Se vale $h_1(n) \geq h_2(n)$ solo per alcuni valori di n, una euristica che certamente domina entrambe é $h(n) = \max(h_1(n), h_2(n))$, perché per tutti i possibili n varrá sempre $h(n) = h_1(n)$ e $h(n) \geq h_2(n)$ oppure $h(n) = h_2(n)$ e $h(n) \geq h_1(n)$. Inoltre, il massimo di piú euristiche ammissibili é sempre un'euristica ammissibile a sua volta.

3. Di fatto, Uniform Cost Search non é altro che l'**Algoritmo di Dijkstra** applicato a grafi senza cicli.

3.4.1 Greedy Search

L'algoritmo **Greedy Search** adotta la strategia di scegliere di espandere sempre il nodo il cui valore di euristica è il minore possibile, ovvero il nodo n che minimizza la funzione $h(n)$. Il nodo viene scelto a prescindere da quale questo sia e da che risultato abbia la relativa espansione. Nella migliore delle ipotesi, questo permette di raggiungere una soluzione velocemente; nella peggiore delle ipotesi, Greedy Search di fatto è assimilabile ad una versione "guidata" di Depth-First Search. Greedy Search è classificabile come algoritmo greedy in quanto la scelta localmente ottima (il nodo che minimizza la funzione di euristica) viene assunta come scelta globalmente ottima, ovvero assumendo che tale nodo sia effettivamente parte di una soluzione ottimale.
Per quanto l'algoritmo sia completo (fintanto che lo spazio degli stati è finito), in genere non è ottimale, perchè nulla garantisce che i nodi aventi minimo valore di euristica siano tutti parte di una soluzione ottima. Comportandosi approssimativamente come DFS, Greedy Search ha un costo in termini di spazio pari a $O(bD)$. Le performance in termini di tempo sono fortemente influenzate dalla scelta dell'euristica: sebbene il tempo di esecuzione teorico sia $O(b^d)$, una buona euristica permette di abbassare considerevolmente tale limite, in certi casi anche fino a $O(bm)$.
Esistono problemi dove è noto a priori che l'euristica sia "infallibile", ovvero che se un nodo ha un valore di euristica minimo allora è certamente parte di una soluzione ottimale. In tali casi, è sostanzialmente garantito che Greedy Search sia l'algoritmo di ricerca con le migliori prestazioni.

3.4.2 A* Search

Il piú comune algoritmo di ricerca informato é **A* Search** (pronuncia: "A-star search"), che combina gli approcci di Uniform Cost search e di Greedy Search. La strategia dell'algoritmo prevede di espandere il nodo che è sia quello "suggerito" dall'euristica sia quello che richiede il minimo costo per raggiungerlo.
Nello specifico, A* Search prevede di espandere il nodo n che minimizza la funzione $f(n) = h(n) + g(n)$, dove $h(n)$ è la funzione di euristica e $g(n)$ è il costo totale associato al raggiungere il nodo. Le due funzioni vengono anche chiamate rispettivamente **forward cost** e **backward cost**. La frontiera viene salvata in una coda di priorità ordinata in ordine decrescente rispetto a $f(n)$, di modo che il nodo estratto sia sempre quello che minimizza tale funzione.
Le prestazioni in termini di tempo impiegato e spazio occupato da A* search sono le medesime di UCS. Fintanto che lo spazio degli stati e finito, A* search é un algoritmo completo. L'ottimalita di A* search dipende dalle caratteristiche della funzione di euristica.

Se A* search su alberi utilizza una euristica ammissibile, A* search e ottimale.

Dimostrazione. Si consideri un problema di ricerca su cui viene applicato A* search avente almeno due nodi obiettivo: A e B, dove il percorso ottimale dalla radice ad A e la soluzione meno costosa. Sia poi $h(n)$ una funzione di euristica ammissibile applicata alla risoluzione del problema. Se A* search e ottimale, il nodo A deve venire espanso prima di B.

Si assuma che la frontiera contenga il nodo B ed almeno un altro nodo non obiettivo n, il quale si trova sul percorso ottimale dalla radice ad A. Il nodo n potrebbe coincidere con A stesso. Se A* search e ottimale, n deve venire espanso prima di B.
Per definizione, si ha $f(n) = g(n) + h(n)$. Essendo l'euristica ammissibile, deve aversi $f(n) \leq g(n) + h^*(n)$. Tuttavia, la somma fra il backward cost (il costo minimo dalla radice ad n) ed il forward cost (il costo minimo da n ad A) coincide esattamente con il costo minimo dalla radice ad A, pertanto $g(n) + h^*(n) = g(A)$. Sostituendo nella relazione precedente, si ha $f(n) \leq g(A)$. Si osservi inoltre come $h(A)$, essendo A uno stato obiettivo, e nullo, pertanto $f(A) = g(A) + 0 = g(A)$. E allora possibile scrivere $f(n) \leq f(A)$.
Essendo per definizione il percorso dalla radice a B piu costoso del percorso dalla radice ad A, si ha $g(A) < g(B)$. Inoltre, essendo B uno nodo obiettivo, deve aversi $f(B) = g(B)$; sostituendo nella precedente, si ha $g(A) < f(B)$. Essendo pero $f(A) = g(A)$, si ha $f(A) < f(B)$. Avendosi pero trovato che $f(n) \leq f(A)$, e possibile concludere che $f(n) < f(B)$.

Dato che A* search espande il nodo della frontiera che minimizza $f(n)$, e certo che venga espanso prima n di B. Essendo pero n un qualsiasi nodo lungo il percorso ottimale dalla radice ad A, significa che tutti i nodi che si trovano lungo il percorso ottimale dalla radice ad A verranno espansi prima di B. Tuttavia, n potrebbe coincidere con A, pertanto e possibile assumere che A venga espanso prima di B. Questo prova che A* search e un algoritmo ottimale.

Se A* search su grafi utilizza una euristica consistente, A* search e ottimale.

Capitolo 4
Problemi di planning

4.1 Planning classico

Si definisce **Problema di Planning** un problema dove si richiede di trovare una strategia che permetta ad un agente di raggiungere un determinato stato obiettivo. Sebbene possa sembrare indistinguibile da un problema di ricerca, ha con questo due notevoli differenze: la prima e che i problemi di planning sono rappresentati mediante linguaggi specifici, detti **linguaggi di planning**, il secondo che il focus del problema e maggiormente improntato sugli stati, sulle azioni e sul concetto di *tempo*.
Il problemi di planning piu semplici sono quelli in cui lo stato iniziale e completamente conosciuto, in cui l'effetto delle azioni e prevedibile ed in cui le percezioni dell'agente sono infallibili (in altri termini, i problemi che operano su un ambiente discreto, deterministico, statico e accessibile).
Problemi di planning con queste caratteristiche sono chiamati **problemi di planning classico**, e sono costituiti dai seguenti elementi:

- Uno **spazio degli stati** S, finito e discreto;
- Uno **stato iniziale** (noto) $s_0 \in S$;
- Un insieme di **stati obiettivo** $S_G \subseteq S$;
- Un insieme di **azioni** $A(s) \subseteq A$ applicabile in ciascuno stato $s \in S$;
- Una **funzione di transizione deterministica** $s^j = f(a, s)$ per ogni $a \in A(s)$;

Un **plan** é una sequenza di azioni $a_0, ..., a_n$ che mappa s_0 su S_G. In altri termini, esiste una sequenza di stati $s_0..., s_{n+1}$ di modo che $a_i \in A(s_i), s_{i+1} = f(a_i, s_i)$ e $s_{n+1} \in S_G$ per $i = 0, ..., n$. Un plan viene detto **ottimale** se minimizza la somma $\sum_{i=0}^{n} c(a_i, s_i)$, ovvero la somma dei costi di ciascuna azione di cui tale plan é costituito.
I linguaggi di planning hanno il vantaggio di essere indipendenti dal dominio del problema in esame. Questo permette a problemi completamente diversi di essere approcciati con le stesse metodologie, fintanto che e possibile rappresentare mediante tale linguaggio tutti i suoi elementi [1]. Inoltre, questa rappresentazione e molto piu ricca della rappresentazione estensionale degli insiemi, pertanto, ad esempio non e piu necessario enumerare esplicitamente tutte le transizioni degli stati nella funzioni di transizione.
Una volta definito il linguaggio di planning, é possibile utilizzarlo per codificare un insieme di informazioni in una knowledge base. L'approccio usato per la costruzione di un agente é *dichiarativo*: é sufficiente istruirlo con le nozioni contenute nella KB per poi fare deduzioni ed ottenere risposte.
Questi linguaggi sono fondamentalmente delle derivazioni delle logiche classiche, come la logica proposizionale o la logica predicativa. Il motivo per cui non vengano usate direttamente queste ultime e che il loro formalismo male si adatta a questo tipo di problemi, dato che per descrivere un certo stato sarebbe necessario enumerare esplicitamente ogni condizione che puo verificarsi ed assegnarvi un valore di verita. D'altro canto, potrebbero verificarsi situazioni in cui lo stesso concetto verrebbe meglio espresso utilizzando un costrutto della logica del primo ordine piuttosto che dai piu ristretti linguaggi di planning.
Avere problemi codificati in linguaggi standard permette di risolverli usando procedure a loro volta standard. Infatti, la descrizione del problema viene fornita ad un **planner**, in genere un software che usa tale descrizione per dedurre in maniera del tutto automatica una soluzione. L'ulteriore vantaggio e che per risolvere il problema non e necessario concentrarsi sui dettagli implementativi del planner, come ad esempio quale algoritmo usa per calcolare una soluzione, ma solamente sulla descrizione del problema.
I problemi di planning possono essere risolti come problemi di ricerca [2]; infatti, e sempre possibile convertire un problema di planning P in un problema di ricerca su grafi equivalente $S(P)$, mappando ciascuno stato (insieme di atomi) sui nodi di un grafo direzionato. Una soluzione per $S(P)$ sara, in una forma opportuna, una soluzione per P.
Dato che, in genere, $S(P)$ é infinitamente piú complesso di P, per risolverlo in maniera efficiente é necessario adoperare un algoritmo di ricerca che fa uso di una euristica. Per costruire una euristica per $S(P)$ é possibile utilizzare il medesimo metodo usato finora, ovvero costruendo, a partire da $S(P)$, un problema rilassato. Tuttavia, per il modo in cui i linguaggi di planning sono strutturati, é invece possibile derivare in maniera del tutto automatica una euristica applicabile ad $S(P)$ a partire da P, a prescindere da quale problema P sia. Inoltre, essendo P piu semplice di $S(P)$, ricavare una euristica per $S(P)$ a partire da P e anche piu efficiente che ricavarla da una versione rilassata di $S(P)$.
Come giá anticipato, non tutti i problemi non possono essere formulati in un linguaggio di planning, e non sono pertanto risolvibili con questo approccio. Altri problemi sono invece intrinsecamente complessi e, sebbene sia possibile formularli come problemi di planning, verrebbero comunque risolti in maniera subottimale. In questi casi, é preferibile un approccio imperativo, dove il codice é pensato ad-hoc per il problema in esame.

1. Diversi problemi reali, come ad esempio la risoluzione del cubo di Rubik, sono effettivamente rappresentabili in forme di questo tipo.
2. Un approccio alternativo prevede di riformulare i problemi di planning come **problemi di soddisfacibilitá booleana** (**boolean satisfiability problem**, **SAT**), ovvero il problema di determinare se esiste una interpretazione che soddisfi una data formula booleana.

4.1.1 STRIPS

Un linguaggio molto semplice per la definizione di problemi di ricerca e **STRIPS** (**Stanford Research Institute Problem Solver**). Un problema codificato in STRIPS é una quadrupla $P = (F, O, I, G)$:

- Un insieme F di **atomi**, ovvero di variabili booleane;
- Un insieme O di **operatori** (azioni). Ciascun operatore o e costituito da una tripla:
  - $Prec(o)$, una lista che contiene tutti gli atomi che devono essere veri affinche o possa essere eseguito;
  - $Add(o)$, una lista che contiene tutti gli atomi che, dopo l'applicazione di o, diverranno veri;
  - $Del(o)$, una lista che contiene tutti gli atomi che, dopo l'applicazione di o, diverranno falsi.
- Uno **stato iniziale** $I \subseteq F$, costituito da tutti gli atomi che sono inizialmente veri (vale la closed-world assumpion; tutto ció che non é inizialmente vero é assunto falso);
- Uno **stato finale** $G \subseteq F$, costituito da tutti gli atomi che devono essere veri affinche sia possibile enunciare il raggiungimento dell'obiettivo.

Una soluzione per STRIPS non e altro che una sequenza ordinata di azioni che permette di trasformare la lista di atomi iniziali in una delle liste di atomi obiettivo.
STRIPS e simile alla logica proposizionale, ma e meno espressivo. Infatti, sebbene sia possibile imporre che lo stato obiettivo debba essere costituito da certi atomi veri, non e possibile imporre che debba essere costituito da certi atomi falsi. Allo stesso modo, sebbene sia possibile imporre una precondizione positiva (l'azione e applicabile se determinati atomi sono positivi) non e possibile imporre una precondizione negativa (l'azione non e applicabile se determinati atomi sono negativi) [3]. Inoltre, STRIPS non permette la definizione di variabili, perche in ciascun operatore gli atomi coinvolti devono essere sempre nominati esplicitamente.
Un problema $P = (F, O, I, G)$ scritto nel formalismo di STRIPS puó essere tradotto in un problema di ricerca equivalente $S(P)$ nel seguente modo:

- Gli stati $s \in S(P)$ equivalgono a collezioni di atomi di F;
- Lo stato iniziale s_0 di $S(P)$ equivale a I;
- Gli stati obiettivo di $S(P)$ equivalgono agli s tali per cui $G \subseteq s$;
- Le azioni a in $A(s)$ equivalgono alle operazioni O, di modo che $Prec(a) \subseteq s$;
- Lo stato successivo s^j é dato da $s - Del(a) + Add(a)$;
- I costi delle azioni $c(a, s)$ sono tutti pari a 1;

Naturalmente, una soluzione (ottimale) per P é anche una soluzione ottimale per $S(P)$. Dato che gli stati di $S(P)$ equivalgono a "combinazioni" di elementi di P, é facile verificare che se P ha n condizioni, il problema di ricerca equivalente $S(P)$ ha 2^n stati; il risparmio in termini di spazio che offre STRIPS é quindi notevole.

Si consideri il problema $P = (F, I, O, G)$ formulato nel linguaggio STRIPS, cosí costruito:

$$F = \{p, q, r\} \qquad I = \{p\} \qquad \begin{aligned} Prec(a) &= \{p\}, Add(a) = \{q\}, Del(a) = \{\} \\ Prec(b) &= \{q\}, Add(b) = \{r\}, Del(b) = \{q\} \end{aligned} \qquad G = \{q, r\}$$

- Partendo dallo stato iniziale, l'operazione b non é applicabile, perché le precondizioni non sono soddisfatte. É peró possibile applicare a, essendo le precondizioni soddisfatte, e q viene aggiunto allo stato iniziale. Lo stato attuale diventa $\{p, q\}$;
- L'operazione b diventa applicabile, perché le precondizioni sono ora soddisfatte. Applicando b viene aggiunto r e viene tolto q, ottenendo $\{p, r\}$;
- Applicando nuovamente a viene (ri)-aggiunto p, ottenendo $\{q, r, p\}$ e raggiungendo lo stato obiettivo.

Per costruire una euristica per $S(P)$ a partire da un problema P codificato in linguaggio STRIPS, e possibile eliminare tutte le precondizioni dalle operazioni di P, di modo che queste siano applicabili in (circa) ogni istante: questo approccio viene chiamato **ignore-precondition heuristic**. Innanzitutto, tutte le azioni vengono rilassate rimuovendo tutte le precondizioni e tutti gli effetti ad eccezione di quelli che sono presenti nell'obiettivo. Dopodiché, si conta qual'é il numero minimo di azioni necessarie affinché l'unione di tali azioni soddisfi l'obiettivo [4] e si usa tale valore

3. Esistono varianti di STRIPS che estendono il linguaggio per colmare queste e altre lacune rappresentative.
4. Questo é un esempio di **problema di copertura**.

come euristica.
In alternativa, é possibile eliminare le rimozioni da tutte le operazioni di *P*, di modo che il progresso verso il goal proceda in maniera monotona e senza che un'azione influisca sul progresso di un'altra: questo approccio é chiamato **ignore-delete-lists heuristic**. Si modifica *P* di modo che tutti gli obiettivi e tutte le precondizioni contengano solo aggiunte, dopodiché vengono eliminate tutte le rimozioni da ogni azione. La lunghezza di un plan ottimale per il problema rilassato cosí costruito viene utilizzata come euristica.
Quest'ultimo approccio e quello utilizzato dal planner **FF**, uno dei piu noti planner attualmente in uso. FF risolve un problema di planning mediante A* search; l'euristica h^{FF} usata per effettuare tale ricerca viene calcolata costruendo una versione rilassata del problema in cui le operazioni di cancellazione sono ignorate, che viene poi risolto (in tempo polinomiale). Tale soluzione viene poi estratta e la sua lunghezza utilizzata come euristica.
La procedura `computeRPG` costruisce il problema rilassato in questione a partire dall'insieme di azioni A del problema principale, del suo obiettivo g e del suo stato iniziale s_i. Il tempo t viene inizializzato a 0, mentre F_t, lo stato al tempo t, viene inizializzato ad s_i. La procedura continua finche gli atomi di g non sono interamente contenuti in F_t. La variabile A_t contiene tutte le azioni in A che, al tempo t, hanno tutte le precondizioni soddisfatte. Il nuovo stato F_t e uguale al precedente (F_{t-1}) ma a cui vengono aggiunti tutti gli effetti di tutte le azioni in A_t; se fra i due stati non c'e differenza, il problema e evidentemente irrisolvibile. L'output della procedura e una sequenza $[F_0, A_1, F_1, \ldots, A_t, F_t]$: F_t e il primo stato che contiene interamente g provando tutte le azioni possibili.

```
procedure computeRPG(A, si, g)
    F0 ← si
    t ← 0
    while (g ⊈ Ft) do
        t ← t + 1
        At ← {a ∈ A | precond(a) ⊆ Ft}
        Ft ← Ft-1
        foreach a ∈ At do
            Ft ← Ft ∪ effects+(a)
        if (Ft = Ft-1) then
            return err
    return [F0, A1, F1, …, At, Ft]
```

A partire da tale sequenza e dall'obiettivo g, la procedura `extractRPSize` calcola l'euristica sulla base della sua lunghezza. Nella variabile M viene salvato l'ultimo passo con cui e stata soddisfatta la condizione del goal. Per ogni passo temporale da 0 ad M, viene costruito un insieme G_t che contiene tutti gli atomi del goal che vengono soddisfatti nel t-esimo passo temporale. Infine, a partire dall'ultimo G_t, vengono propagate all'indietro le precondizioni delle azioni che hanno permesso di soddifare gli atomi contenuti in G_t.

```
procedure extractRPSize([F0, A1, F1, …, Ak, Fk], g)
    // if (g ⊈ Fk) then
    //    return err
    M ← max(firstLevel(gi, [F0, …, Fk]) | gi ∈ g)
    for t ← 0 to M do
        Gt ← {gi ∈ g | firstLevel(gi, [F0, …, Fk]) = t}
    for t ← M to 1 do
        foreach gt ∈ Gt do
            select a : firstLevel(a, [A1, …, At]) = t and gt ∈ effects+(a)
            foreach p ∈ precond(a) do
                GfirstLevel(p, [F0, …, Fk]) ← GfirstLevel(p, [F0, …, Fk]) ∪ {p}
    return number of selected actions
```

4.1.2 PDDL

Una estensione di STRIPS che permette l'uso di variabili é **Planning Domain Definition Language** (**PDDL**). Un problema in PDDL é formato da due componenti: un **dominio** ed una **istanza**. Il dominio contiene lo schema delle azioni, degli atomi ed i tipi degli argomenti:

```
(define (domain DOMAIN_NAME)
    (:predicates (PREDICATE_1_NAME ?A1 ?A2 … ?AN)
                 (PREDICATE_2_NAME ?A1 ?A2 … ?AN)
                 …)

    (:action ACTION_1_NAME
        [:parameters (?P1 ?P2 … ?PN)]
        [:precondition PRECOND_FORMULA]
        [:effect EFFECT_FORMULA]
    )
    (:action ACTION_2_NAME
       …)
    …)
```

I nomi dei predicati e delle azioni sono costituiti da caratteri alfanumerici e/o da trattini. I parametri dei predicati e delle azioni si distinguono dai nomi perché hanno un "?" come prefisso. I parametri usati nella dichiarazione dei predicati non hanno altra utilitá al di fuori di specificare il numero di argomenti che il predicato debba avere; fintanto che hanno nomi distinti, il nome scelto per i parametri non é rilevante. I predicati possono anche avere zero parametri.
Una precondizione puó essere espressa come:

- Una formula atomica: `(PREDICATE_NAME ARG1 … ARGN)`
- Una congiunzione di formule atomiche: `(and ATOM1 … ATOMN)`
- Una disgiunzione di formule atomiche: `(or ATOM1 … ATOMN)`
- La negazione di una formula atomica: `(not CONDITION_FORMULA)`
- Una formula con quantificatore universale: `(forall (?V1 ?V2 … ) CONDITION_FORMULA)`

- Una formula con quantificatore esistenziale: `(exists (?V1 ?V2 … ) CONDITION_FORMULA)`

In PDDL, gli effetti di una azione non sono distinti in *Add* e *Delete*. Le rimozioni vengono espresse sotto forma di negazioni. L'effetto di una azione puó essere espresso come:

- Una aggiunta: `(PREDICATE_NAME ARG1 … ARGN)`
- Una rimozione: `(not (PREDICATE_NAME ARG1 … ARGN))`
- Una congiunzione di effetti atomici: `(and ATOM1 … ATOMN)`
- Un effetto condizionale: `(when CONDITION_FORMULA EFFECT_FORMULA)`
- Una formula con quantificatore universale: `(forall (?V1 ?V2 … ) EFFECT_FORMULA)`

L'istanza contiene lo stato iniziale, lo stato obiettivo e tutti gli oggetti che figurano nel problema. Una istanza puó essere espressa come:

```
(define problem PROBLEM_NAME)
    (:domain DOMAIN_NAME)
    (:objects OBJ1 OBJ2 … OBJN)
    (:init ATOM1 ATOM2 … ATOMN)
    (:goal CONDITION_FORMULA)
)
```

La descrizione dello stato iniziale (`:init`) é semplicemente una lista di tutti i predicati che sono veri nello stato iniziale; tutti gli altri sono assunti falsi. A differenza delle precondizioni delle azioni, gli stati iniziali e obiettivo devono necessariamente essere *grounded*, ovvero non possono avere delle variabili come argomenti.
I tipi devono essere dichiarati prima che possano essere utilizzati. La dichiarazione di un tipo puó essere espressa come:

```
(:types NAME1 … NAMEN)
```

Per dichiarare il tipo di un parametro di un predicato o di una azione, si riporta `?X - TYPE_OF_X` . Una lista di parametri dello stesso tipo puó essere abbreviata come `?X ?Y ?Z - TYPE_OF_XYZ` .

4.2 Planning probabilistico: filtri Bayesiani

Si consideri una situazione in cui l'agente conosce con certezza l'effetto che hanno le sue azioni, ma non conosce con certezza lo stato in cui si trova. In ogni istante temporale t l'agente opera una misurazione, che usa per fornire una stima probabilistica del trovarsi nello stato che si aspetta. In risposta alla misurazione, compie una azione e cambia di stato (introducendo ulteriore rumore).
Si consideri come istante di tempo iniziale $t_0 = 0$. Siano:

- x_t lo stato in cui l'agente effettivamente si trova allo stato t;
- z_t la misurazione compiuta dall'agente all'istante t;
- μ_t l'azione compiuta dall'agente in risposta alla percezione nell'intervallo di tempo $(t-1;t]$.

Come giá detto, l'agente non puó conoscere con certezza in quale stato si trova, e deve limitarsi a dare una stima probabilistica. Sia allora $P(x_t)$ la probabilitá "in assoluto" che l'agente si trovi nello stato x al tempo t. É ragionevole assumere che la probabilitá che l'agente si trovi in un certo stato in un certo istante dipenda in una qualche misura dalle misurazioni e dalle azioni compiute in precedenza. In tal senso, ció che si ha interesse a calcolare non é tanto $P(x_t)$, quanto quella che viene detta **belief** ("fiducia"):

$$\text{bel}(x_t) = P(x_t \mid \mu_1, z_1, \mu_2, z_2, ..., \mu_{t-1}, z_{t-1}, \mu_t, z_t) \qquad \text{bel}^-(x_t) = P(x_t \mid \mu_1, z_1, \mu_2, z_2, ..., \mu_{t-1}, z_{t-1}, \mu_t)$$

La funzione a sinistra indica la probabilita che l'agente si trovi nello stato x al tempo t condizionata da tutte le azioni finora intraprese e da tutte le misurazioni compiute. La funzione a destra indica la stessa probabilita ma *prima* che l'ultima misurazione venga effettuata.
Si noti come μ_t e z_t partano da $t = 1$ e non da $t = 0$, dato che si assume che lo stato x_0 venga determinato a priori. Ovvero, lo stato iniziale dell'agente deve venirgli fornito "dall'esterno", assumendo che questo sia corretto con certezza, e poi sulla base di questo aggiorna di volta in volta la propria conoscenza sul mondo in termini probabilistici.
Un'assunzione molto forte (e in genere valida) che e possibile fare e la cosiddetta **assunzione Markoviana**. In termini molto generali, questa prevede che un processo stocastico che si evolve nel tempo non dipenda da tutte le osservazioni effettuate prima di un un certo istante di tempo, ma solamente da quella immediatamente precedente. Nel contesto in esame, questo equivale a dire che il grado di fiducia dell'agente al tempo t dipende solamente dalla misurazione compiuta e dall'azione intrapresa al tempo $t-1$. Se tale assunzione vale, e possibile semplificare le espressioni precedenti come:

$$\text{bel}(x_t) = P(x_t \mid \mu_t, z_t) \qquad \text{bel}^-(x_t) = P(x_t \mid \mu_t)$$

Al fine di costruire un agente in grado di aggiornare la propria conoscenza sul mondo di volta in volta (prendendo per buona l'assunzione Markoviana), e necessario che questo possa esprimere il proprio grado di fiducia in un certo istante in funzione del suo grado di fiducia nell'istante precedente [5].

5. Questo tipo di approccio viene spesso utilizzato nella robotica, dove la posizione di un robot in un ambiente ignoto parte da un valore noto e viene mano

Filtro Bayesiano per agenti probabilistici. Se sono valide le assunzioni Markoviane, e possibile esprimere il grado di fiducia dell'agente al tempo t in funzione del grado di fiducia dell'agente al tempo $t-1$ secondo la seguente equazione:

$$bel(x_t) = \eta P(z_t \mid x_t) \int P(x_t \mid \mu_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$$

Dimostrazione. Sia il grado di fiducia dell'agente al tempo t espresso come di consueto:

$$\text{bel}(x_t) = P(x_t \mid \mu_1, z_1, ..., \mu_t, z_t)$$

Applicando la regola di Bayes:

$$\text{bel}(x_t) = \eta P(z_t \mid x_t, \mu_1, z_1, ..., \mu_t) P(x_t \mid \mu_1, z_1, ..., \mu_t)$$

Prendendo come valida l'assunzione Markoviana, e possibile semplificare l'espressione come:

$$\text{bel}(x_t) = \eta P(z_t \mid x_t) P(x_t \mid \mu_1, z_1, ..., \mu_t)$$

Applicando la formula delle probabilitá totali:

$$\text{bel}(x_t) = \eta P(z_t \mid x_t) \int P(x_t \mid \mu_1, z_1, ..., \mu_t, x_{t-1}) P(x_{t-1} \mid \mu_1, z_1, ..., \mu_t) dx_{t-1}$$

Prendendo come valida l'assunzione Markoviana, e possibile semplificare l'espressione come:

$$\text{bel}\left(x_t\right) = \eta P\left(z_t \mid x_t\right) \int P\left(x_t \mid \mu_t, x_{t-1}\right) P\left(x_{t-1} \mid \mu_1, z_1, ..., \mu_t\right) dx_{t-1}$$

Che equivale a scrivere:

$$bel(x_t) = \eta P(z_t \mid x_t) \int P(x_t \mid \mu_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$$

Questa e una forma di filtro Bayesiano. Nel contesto degli agenti probabilistici, per implementarlo nella forma di un algoritmo, si preferisce separare il procedimento in due parti, l'osservazione e l'attuazione.
L'attuazione corrisponde al ricavare $\text{bel}(x_t)$ a partire da $\text{bel}^-(x_t)$. Questo puo essere fatto banalmente applicando la regola di Bayes all'espressione per $\text{bel}(x_t)$:

$$\text{bel}(x_t) = P(x_t \mid \mu_t, z_t) = \eta P(z_t \mid x_t) P(x_t \mid \mu_t) = \eta P(z_t \mid x_t) \text{bel}^{-1}(x_t)$$

L'osservazione corrisponde al ricavare $\text{bel}^-(x_t)$ a partire da $\text{bel}(x_{t-1})$. Questo puo essere fatto combinando il risultato appena ottenuto con la formula completa per il filtro Bayesiano:

$$bel(x_t) = \eta P(z_t \mid x_t) \int P(x_t \mid \mu_t, x_{t-1}) bel(x_{t-1}) dx_{t-1} = \eta P(z_t \mid x_t) \text{bel}^{-1}(x_t) \;\Rightarrow\; \text{bel}^{-1}(x_t) = \int P(x_t \mid \mu_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$$

Traducendo la ricorsione in una iterazione, si ottiene il seguente algoritmo:

```
procedure BAYESIAN-FILTER(bel(X), d)
    η ← 0
    if (d e una percezione z) then
        foreach x in X do
            bel'(x) ← p(z | x) bel(x)
            η ← η + bel'(x)
        foreach x in X do
            bel'(x) ← η^-1 bel'(x)
    else if (d e una azione η) then
        foreach x in X do
            bel'(x) ← ∫ P(x | η, x') bel(x') dx'
```

a mano aggiornata ogni qual volta che l'agente si sposta, ricalcolando la probabilita che l'agente si trovi nella coordinata spaziale che i suoi calcoli rilevano.

```
    return bel'(X)
```

Si noti come il calcolo di bel(x_t) debba venire ripetuto per tutti i possibili stati $x \in X$ in ciascuna iterazione. Questo puo essere un calcolo estremamente esoso, specialmente se il numero di stati sono molti. In genere, il procedimento viene ottimizzato escludendo tutti gli stati che hanno associata una probabilita molto bassa, ed effettuando l'aggiornamento solamente degli stati che hanno probabilita oltre una certa soglia minima.

4.3 Planning probabilistico: MDP

Viene chiamato **Markov Decision Process** (**MDP**) un problema di ricerca in cui vale l'assunzione Markoviana e dove l'ambiente é accessibile ma non deterministico. Ovvero, l'agente sa sempre in che stato si trova ma non ha la certezza che compiere una azione porterá allo stato che si aspetta. Un MDP é costituito da:

- Un insieme discreto di stati S;
- Un insieme discreto di azioni A;
- Uno stato iniziale $s_0 \in S$;
- Un modello di transizione $T(s, a, s')$, con $a \in A$ e $s, s' \in S$. Questo indica qual'é la probabilitá che venga effettivamente raggiunto lo stato s' eseguendo a mentre ci si trova in s. In termini di calcolo delle probabilitá, essendo valida l'assunzione Markoviana, $T(s, a, s')$ equivale a $P(s' \mid s, a)$.
- Una **funzione di ricompensa** $R(s, a, s')$, che associa un valore numerico a ciascuna transizione. Tale valore rappresenta quanto é "vantaggioso" per l'agente compiere la transizione da s a s' mediante a;
- Opzionalmente, uno stato finale $s_f \in S$.

A differenza dei problemi di ricerca, dove si cerca di minimizzare la funzione di costo, negli MDP si cerca di massimizzare la funzione di ricompensa. Infatti, negli MDP lo stato finale non e sempre presente, perche l'obiettivo dell'agente potrebbe semplicemente consistere nel raggiungere un certo valore per $R(s, a, s')$, a prescindere da quale sia lo stato in cui si trova quando questo accade.
La differenza piu rilevante fra i due si ha pero nella forma della loro soluzione. Nei problemi di ricerca la soluzione é una sequenza di azioni che conducono dallo stato iniziale ad uno stato obiettivo; negli MDP, questo approccio non e possibile, perché una certa sequenza di azioni é in grado di portare da uno stato ad un altro solamente con una certa probabilitá. Questo significa che eseguendo piu volte la stessa sequenza di azioni potrebbe venire raggiunto uno stato completamente diverso da esecuzione ad esecuzione.
L'azione da eseguire in un certo stato puó essere pensata come una variabile aleatoria, alla quale é associata una probabilitá per ciascun valore che questa puó assumere. Pertanto, una soluzione per agenti probabilistici deve specificare cosa un agente debba fare in *ogni* stato in cui l'agente potrebbe trovarsi. A tale scopo, é necessario introdurre il concetto di **politica**, la soluzione di un MDP.
Una politica π e una funzione che mappa ciascuno stato del problema ad un'azione [6]; dato uno stato s, $\pi(s)$ é l'azione che la politica π "raccomanda" di eseguire se ci si trova in s. La politica migliore π^*, detta **politica ottimale**, e quella che in ogni stato raccomanda l'azione che restituisce il massimo valore di ricompensa possibile.
Ogni volta che una determinata politica viene eseguita a partire dallo stato iniziale, la natura stocastica dell'ambiente porta a generare diverse sequenze di azioni, ciascuna con una propria probabilità. La "qualità" di una politica viene pertanto misurata a partire dall'utilità *attesa* delle possibili sequenze di azioni generate da tali politiche. Pertanto, la politica ottimale e quella che massimizza tale valore di utilita attesa.

Data la sequenza ordinata S di n stati $[s_0, s_1, ..., s_n]$, sia $U(S)$ la ricompensa complessiva ricavata dall'attraversare ordinatamente tale sequenza di stati. Esistono diverse regole per calcolare la ricompensa complessiva; la piu semplice è la **ricompensa additiva**, dove la ricompensa totale è semplicemente la somma delle ricompense associate ai singoli stati:

$$U(S) = U([s_0, s_1, ..., s_n]) = R(s_0, \pi(s_0), s_1) + R(s_1, \pi(s_1), s_2) + ... + R(s_{n-1}, \pi(s_{n-1}), s_n) = \sum_{i=0}^{n-1} R(s_i, \pi, s_{i+1})$$

Il problema di questa forma di assegnazione della ricompensa e che, in molte situazioni, incentiva l'agente a non raggiungere mai l'obiettivo, perche puo guadagnare ricompensa quasi indefinitamente. Una regola alternativa è la **ricompensa con discount**, dove la ricompensa associata al trovarsi in un determinato stato decresce di una certa percentuale ad ogni cambio di stato. Indicando con γ un valore compreso fra 0 e 1, la ricompensa totale adoperando tale metodo è data da:

$$U(S) = U([s_0, s_1, ..., s_n]) = R(s_0, \pi, s_1) + \gamma R(s_1, \pi, s_2) + ... + \gamma^{n-1} R(s_{n-1}, \pi, s_n) = \sum_{i=0}^{n-1} \gamma^i R(s_i, \pi, s_{i+1})$$

γ determina quanta priorità debba dare l'agente al raggiungere determinati stati in una determinata iterazione. Se γ è un valore prossimo a 0, le ricompense date dagli stati raggiunti nelle prime iterazioni hanno un peso molto maggiore sul valore di ricompensa complessivo rispetto a quelle fornite dagli stati raggiunti nelle ultime iterazioni. Se γ è un valore prossimo ad 1, le ricompense date dagli stati raggiunti nelle prime iterazioni e nelle ultime iterazioni hanno un peso comparabile. Se γ è esattamente 1, non vi è alcuna differenza nel raggiungere uno stato in una certa iterazione piuttosto che in un'altra, e la regola con discount coincide di fatto con la regola additiva.

6. Il motivo per cui e stato specificato che gli MDP hanno un numero finito di stati e che altrimenti non sarebbe possibile definire una politica.

Usando la regola con discount, la ricompensa complessiva U è un valore limitato, e pertanto non puo crescere indefinitamente.

Dimostrazione. Sia data la sequenza S di n stati $[s_0, s_1, ..., s_n]$ che l'agente attraversa in un problema MDP. Essendo per questo S finita, deve esserlo anche la sequenza di ricompense $[R(s_0, \pi(s_0), s_1), R(s_1, \pi(s_1), s_2), R(s_{n-1}, \pi(s_{n-1}), s_n)]$. Ne consegue che esiste un elemento di tale sequenza, sia questo R_{max}, che e maggiore o uguale a tutte le altre ricompense.
Si noti come $1 + \gamma + \gamma^2 + ... + \gamma^n$ sia una serie geometrica; se γ è un valore compreso fra 0 e 1, vale:

$$\lim_{n\to+\infty} 1 + \gamma + \gamma^2 + ... + \gamma^n = \lim_{n\to+\infty} \sum_{i=0}^{n} \gamma^i = \frac{1}{1-\gamma} \Rightarrow 1 + \gamma + \gamma^2 + ... + \gamma^{n-1} \leq \frac{1}{1-\gamma}$$

Moltiplicando ambo i membri per R_{max}:

$$R_{max} \sum_{i=0}^{n-1} \gamma^i \leq R_{max}\left(\frac{1}{1-\gamma}\right) \Rightarrow R_{max} + R_{max}\gamma + R_{max}\gamma^2 + ... + R_{max}\gamma^{n-1} \leq \frac{R_{max}}{1-\gamma}$$

Essendo R_{max} maggiore di tutti i valori in $[R(s_0, \pi(s_0), s_1), ..., R(s_{n-1}, \pi(s_{n-1}), s_n)]$, é possibile effettuare la seguente minorazione:

$$R(s_0, \pi(s_0), s_1) + \gamma R(s_1, \pi(s_1), s_2) + \gamma^2 R(s_2, \pi(s_2), s_3) + ... + \gamma^{n-1} R(s_{n-1}, \pi(s_{n-1}), s_n) \leq \frac{R_{max}}{1-\gamma}$$

Ovvero, $U \leq R_{max} / (1-\gamma)$. Essendo $0 < \gamma < 1$ ed essendo sia R_{max} che U valori positivi, U e effettivamente limitato, e non puo crescere oltre tale limite.

Si assuma di utilizzare la regola con discount. Ci si chiede come ricavare, dato un MDP, la politica migliore. Innanzitutto, per ricavare la politica migliore non e possibile semplicemente massimizzare la funzione U, perche questa opera su una sequenza di stati e, come gia detto, una sequenza di stati non e una soluzione accettabile per un MDP. La politica migliore e invece quella che massimizza la ricompensa totale *attesa*.
Sia r_t la ricompensa ottenuta dall'agente in un MDP al tempo t (a prescindere da quale sia lo stato raggiunto che la ha indotta). r_t puo essere pensato come una variabile aleatoria che assume un valore diverso a seconda di ciascuna esecuzione dell'MDP. Sia R_t la somma parziale con discount di tutte le ricompense ottenute a partire da r_t:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{+\infty} \gamma^i r_{t+i}$$

R_t e essa stessa una variabile aleatoria. Per indicarne il valore a cui tende per $i \to + \infty$, se ne calcola il valore atteso:

$$\mathbb{R}_t = E[R_t] = E\left[\sum_{i=0}^{+\infty} \gamma^i r_{t+i}\right] = \sum_{i=0}^{+\infty} E\left[\gamma^i r_{t+i}\right]$$

Per la linearita del valore atteso, e possibile riscrivere l'equazione in forma ricorsiva:

$$\mathbb{R}_t = \sum_{i=0}^{+\infty} E\left[\gamma^i r_{t+i}\right] = E\left[r_t\right] + \sum_{i=1}^{+\infty} E\left[\gamma^i r_{t+i}\right] = E\left[r_t\right] + \gamma \mathbb{R}_{t+1}$$

Sia s_0 lo stato in cui viene ottenuta la ricompensa r_t. La ricompensa complessiva ottenuta dall'agente partendo dallo stato s_0 e seguendo le azioni suggerite dalla politica π viene indicata con $U^\pi(s_0)$:

$$U^\pi\left(s_0\right) = E\left[\sum_{t=0}^{+\infty} \gamma^t R\left(s_t, \pi\left(s_t\right), s_{t+1}\right)\right]$$

U^π puo essere calcolato per tutti gli stati possibili.
sia S_t^π una variabile aleatoria che indica lo stato che raggiunge l'agente adoperando una certa politica π a partire dallo stato s_0 al tempo t. La distribuzione di probabilitá lungo la sequenza di stati $S_1, S_2, ...$ é determinata a partire dallo stato s_0, dalla politica π e dal modello di transizione, e viene indicata con $U^\pi(s_0)$:

$$U^\pi(s_0) = E\left[R\left(s_0, \pi\left(s_0\right), s_1\right) + \gamma R\left(s_1, \pi\left(s_1\right), s_2\right) + \gamma^2 R\left(s_2, \pi\left(s_2\right), s_3\right) + ...\right] = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi, s_{t+1})\right]$$

Dove il valore atteso é calcolato rispetto alla distribuzione della sequenza di stati indotta dall'applicare π a s_0. Sia $\pi^*_{s_0}$ la politica migliore fra tutte quelle applicabili a partire da s_0: questa non é altro che la politica π che massimizza $U^\pi(s_0)$:

$$\pi^*_{s_0} = \text{argmax}_\pi(U^\pi(s))$$

Fintanto che viene impiegata la regola con discount, é possibile dimostrare che la politica ottimale non dipende da quale stato viene usato come stato di partenza. Questo significa che, per qualsiasi stato s, il valore di utilita associato a tale stato é semplicemente $U^{\pi^*}(s)$, a prescindere di come tale stato viene raggiunto.
La funzione $U(s)$ permette all'agente di scegliere la prossima azione da compiere sulla base del principio di massima utilitá attesa, ovvero che massimizza la somma pesata dalla probabilitá di compiere una transizione verso un certo stato fra la ricompensa che viene ottenuta raggiungendolo e la penalitá introdotta dalla regola con discount:

$$\pi^*(s) = \text{argmax}_{a \in A(s)} \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma U(s')\Big]$$

Da questo segue una diretta relazione che sussiste fra l'utilitá di uno stato e l'utilitá degli stati vicini, ovvero quelli che l'agente puó raggiungere a partire da questo: il valore di utilitá di uno stato é dato dalla somma fra il valore atteso della ricompensa portata dalla prossima transizione sommata all'utilitá (scontata) dello stato di arrivo, assumendo che l'agente scelga una politica ottimale. L'equazione risultante, che permette di esprimere la funzione di utilitá degli stati in forma ricorsiva, prende il nome di **equazione di Bellman**:

$$U(s) = \max_{a \in A(s)} \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma U(s')\Big]$$

Un'altra quantitá importante é la **funzione di azione-utilitá**, o **Q-function**, che riporta l'utilitá attesa dal compiere una certa azione in un certo stato. Il legame fra Q-function e funzione di utilitá é immediato:

$$U(s) = \max_a Q(s,a)$$

Inoltre, é possibile estrarre la politica ottimale a partire dalla Q-function come segue:

$$\pi^*(s) = \text{argmax}_a Q(s,a)$$

É possibile costruire una equazione di Bellman anche per la Q-function, notando come il valore atteso totale per il compiere una azione é dato dalla somma fra la ricompensa immediata e la penalitá del raggiungere il nuovo stato, che a sua volta é esprimibile in termini della Q-function:

$$Q(s,a) = \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma U(s')\Big] = \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma \max_{a'} Q(s',a')\Big]$$

Risolvendo una equazione di Bellman per U o per Q é possibile ricavare una politica ottima per un problema di planning probabilistico. Nello specifico, le equazioni di Bellman sono alla base di uno dei metodi usati per risolvere un problema MDP chiamato **value iteration**.
Per ciascuno stato s di un MDP dovrebbe venire calcolato $U(s)$ attraverso l'equazione di Bellman. Se il numero di stati dell'MDP é n, questo consiste nel risolvere un sistema di n equazioni in n incognite. Se tale sistema fosse un sistema di equazioni lineari questo sarebbe computazionalmente possibile, ma tale sistema non é lineare, perché nell'equazione di Bellman compare l'operatore max, che non é lineare.
Value iteration aggira il problema "stimando" il valore di $U(s)$ per ciascuno stato di iterazione in iterazione fino ad ottenerne una approssimazione accettabile. Sia $U_i(s)$ il valore di utilitá per lo stato s alla i-esima iterazione; viene chiamato **aggiornamento di Bellman** l'aggiornamento di tale valore sulla base del precedente:

$$U_{i+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s'} T(s,a,s')\Big[R(s,a,s') + \gamma U_i(s')\Big]$$

Inizialmente, i valori di $U(s)$ vengono impostati ad un valore casuale (in genere a 0), e le iterazioni proseguono fintanto che la differenza fra l'utilitá stimata fra una iterazione e quella successiva non é trascurabile.

```
S <= a set of states
A <= a set of actions A(s)
T <= the transition function T(s', a, s)
R <= the reward function R(s', a, s)
γ <= the discount function
ε <= the maximum error allowed in the utility of any state


function VALUE-ITERATION(S, A, T, R, γ, ε)
    δ <= 0
    foreach s in S do
        U[s] <= 0
        U'[s] <= 0


    do
        foreach s in S do
            foreach a in A[s] do
                U'[s] <= max(Q-VALUE(S, A, T, R, U, γ))
            if (|U'[s] - U[s]| > δ) then
```

```
            δ <= |U'[s] - U[s]|
while (δ > ε (1 - γ) / γ)

return U
```

Occorre peró dimostrare che, dopo un numero sufficiente di iterazioni, value iteration restituisce effettivamente una stima corretta dei valori di $U(s)$.

L'aggiornamento di Bellman é una contrazione.

Dimostrazione. Per semplicitá, si consideri l'aggiornamento di Bellman come un operatore B. É quindi possibile scrivere:

$$U_{i+1} \leftarrow BU_i$$

Occorre definire una metrica per lo spazio dei vettori U. Sia $||U||$ il valore assoluto della componente di U avente modulo maggiore:

$$||U|| = \max_s |U(s)|$$

La metrica $d(U, U')$ viene allora definita come $||U - U'||$, ovvero il valore assoluto della differenza fra le componenti aventi modulo maggiore delle due utilitá. Allora:

$$||BU - BU'|| \leq \gamma ||U - U'||$$

Essendo $\gamma \in (0, 1)$, si ha che l'aggiornamento di Bellman é una contrazione rispetto al fattore γ e alla metrica d.

Un approccio alternativo a value iteration é **policy iteration**. Questo si basa sul presupposto che una politica ottimale puó essere ottenuta anche da una funzione di utilitá inaccurata. Policy iteration alterna i seguenti due step in ciascuna iterazione i:

- **Policy evaluation**: data una politica π_i, viene calcolato $U_i = U^{\pi_i}$, la funzione di utilitá in ciascuno stato se venisse applicata π_i;
- **Policy improvement**: viene calcolata una nuova politica π_{i+1}, migliore di π_i, a partire da U_i.

L'algoritmo termina quando la politica π_i non é piú in grado di influire sul risultato di U_i. Quando questo accade, si ha che U_i é (approssimativamente) un punto fisso per l'aggiornamento di Bellman, ed é quindi una soluzione per l'equazione di Bellman, e la politica π_i che la ha generata é una politica ottima. Essendo il numero di politiche finito e venendo le politiche migliorate ad ogni iterazione, é garantito che l'algoritmo termini.

```
S <= a set of states
A <= a set of actions A(s)
T <= the transition function T(s', a, s)
R <= the reward function R(s', a, s)
γ <= the discount function


function POLICY-ITERATION(S, A, T, R, γ)
    foreach s in S do
        U[s] <= 0
    stop <= true
    π <= RANDOM-POLICY()


    do
        U <= POLICY-EVALUATION(S, A, T, R, γ)
        foreach s in S do
            foreach a in A[S] do
            a* <= argmax(Q-VALUE(S, A, T, R, γ))
        if (Q-VALUE(S, a*, T, R, γ) > Q-VALUE(S, γ[s], T, R, γ)) then
            π[s] <= a*
            stop <= false
while (stop)

return π
```

Implementare POLICY-EVALUATION é piú semplice che risolvere l'equazione di Bellman "per intero" (come viene fatto da value iteration), perché l'azione che compare nell'equazione non é una incognita. Infatti, questa é la azione che viene raccomandata dalla politica π_i nello stato s, quindi é una informazione nota:

$$U_i(s) = \sum_{s'} T(s, \pi_i(s), s') \Big[R(s, \pi_i(s), s') + \gamma U_i(s') \Big]$$

Questo semplifica l'equazione eliminando l'operatore max e rendendola una equazione lineare. Se il numero di stati é n vi saranno n equazioni lineari in n incognite, e risolverle con metodi algebrici standard richiede un tempo di esecuzione pari a $O(n^3)$.

Capitolo 5
Supervised Learning

5.1 Introduzione all'apprendimento supervisionato

Il **supervised learning** e una tecnica di apprendimento in cui l'agente apprende sia sulla base dell'input che sulla base dell'output. Il nome "supervised" viene dal fatto che tale tecnica necessita di dei dati che presentano gia la caratteristica che l'agente deve individuare, in modo da apprendere quali sono i pattern che portano un dato a possedere tale caratteristica. Il supervised learning prevede sia una fase di **addestramento**, in cui i pattern vengono individuati, ed una fase di **testing**, in cui si valuta se l'algoritmo e in grado di prevedere il valore della caratteristica su dati ignoti. Naturalmente, e necessario assumere che i dati noti ed i dati ignoti abbiano la stessa distribuzione statistica.
Piu formalmente, sia D un insieme di n elementi. Ciascun elemento possiede m attributi $A_1, A_2, ..., A_m$ ed e membro di una classe. Ciascuno di questi elementi puo essere rappresentato da una $m + 1$-upla, ovvero come $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)$. $\boldsymbol{x}_i$ rappresenta i valori degli attributi dell'i-esimo elemento, mentre y_i rappresenta la classe a cui l'elemento appartiene. Il valore del j-esimo attributo dell'i-esimo elemento viene indicato con $x_{i,j}$.
Si assuma l'esistenza di un legame fra i valori di $\boldsymbol{x}_i$ e quelli di y_i. Deve allora esistere una funzione (ignota) f tale che $f(\boldsymbol{x}_i) = y_i$, ovvero in grado di determinare qual'e il valore di y_i anche se questo non e noto, solamente sulla base di $\boldsymbol{x}_i$.
Il supervised learning costruisce una funzione h, chiamata **ipotesi**, che cerca di stimare f osservando i pattern che sussistono fra i valori di $\boldsymbol{x}_i$ e quelli di y_i. In questo modo, se viene fornito in input ad h una m-upla qualsiasi, questa funzione e in grado di determinare il valore di y_i anche se questo e ignoto. Il supervised learning apprende sia dall'input che dall'output in questo senso: l'input ($\boldsymbol{x}_i$) e l'output (y_i) sono riferiti alla funzione h, ed entrambi sono necessari alla sua costruzione.
Il problema risolto dal supervised learning prende una propria denominazione in base a qual'e il tipo di dato di y_i. Se y_i e un valore numerico, il problema e detto **problema di regressione**, in analogia con il concetto di regressione matematica. Se y_i e un valore booleano, il problema e detto **problema di classificazione**, perche e possibile interpretare y_i come una variabile che indica se l'i-esimo elemento di D appartiene (`True`) o non appartiene (`False`) ad una certa classe. Per convenzione, gli elementi che hanno `True` come valore di tale attributo sono detti **elementi positivi**, mentre mentre quelli che hanno `False` vengono detti **elementi negativi**.
Il tipo di dato degli attributi determina solamente se questi possano essere usati come input di un determinato algoritmo oppure no. Infatti, alcuni metodi di apprendimento operano esclusivamente su attributi di tipi specifici, mentre altri possono operare su attributi eterogenei. E comunque possibile, in genere, convertire un attributo di un tipo in uno o piu attributi equivalenti mediante una **codifica**.
Una codifica molto semplice prende il nome di **one hot encoding**, applicabile quando il dominio dell'attributo generico e discreto e non troppo grande. Questa prevede di sostituire l'attributo con tante variabili binarie quanti sono i valori che l'attributo puo assumere: se un elemento del dataset aveva come i come valore dell'attributo, allora la i-esima di queste variabili binarie avra valore `1` e tutte le rimanenti valore `0`.

> Un esempio di supervised learning si ha nei filtri antispam dei client di posta elettronica. L'idea é quella di fornire al filtro un grande quantitativo di email contrassegnate come spam ed un grande quantitativo di email contrassegnate come non spam, di modo che questo possa estrarre dei pattern comuni nelle email spam e non spam. A questo punto, se viene fornita al filtro una mail qualsiasi, questo é (dovrebbe essere) in grado di determinare autonomamente se la mail é o non é spam.

Vi sono due situazioni patologiche molto comuni che possono presentarsi nella costruzione di un modello supervisionato, **underfitting** e **overfitting**. La prima si verifica quando il modello costruito predice male i dati passati, mentre la seconda si verifica quando il modello predice "cosi bene" i dati su cui e allenato da non tollerare alcun grado di impurita, tanto da fallire per molti dataset solo leggermente differenti Naturalmente, la prima situazione e peggiore della seconda, perche se l'overfitting produce un modello scadente ma comunque funzionante, l'underfitting produce un modello del tutto inutilizzabile.
Esistono sia algoritmi che costruiscono un modello, e che quindi distinguono fra fase di training e fase di test, chiamati **model-based learning**, sia algoritmi che non lo costruiscono, dove tutto e insieme, chiamati **individual-based learning**.
La tecnica piú semplice per allenare un dataset prende il nome di **holdout set**. Questa prevede di separare il dataset in due sotto-dataset: uno, chiamato **training set**, verrá usato per allenare il modello, mentre l'altro, chiamato **test set**, verra usato per testarlo. Questa tecnica e applicabile quando la dimensione del dataset a disposizione é sufficientemente grande da avere due sotto-dataset a loro volta grandi. Naturalmente, gli elementi del training set non devono figurare nella fase di test, cosi come gli elementi del test set non devono figurare nella fase di apprendimento, perche in entrambi i casi il modello avrebbe un evidente bias. Separare nettamente i due sotto-dataset permette di avere un modello funzionante ma che al contempo tollera un minimo margine di errore.
Una tecnica alternativa, chiamata **n-fold cross-validation**, é preferibile quando la grandezza del dataset a disposizione é limitata. Questa prevede di partizionare il dataset a disposizione in m sotto-dataset di uguale grandezza. Per m volte, viene scelto uno degli m sotto-dataset come test set ed i restanti $m-1$ sotto-dataset vengono combinati in un training set. A ciascuna iterazione della procedura e possibile associare una accuratezza; l'accuratezza complessiva viene calcolata come media di tutte le m accuratezze cosí ottenute. In genere, sono comuni partizionamenti in `5` (5-fold cross-validation) o in `10` (10-fold cross validation) sotto-dataset.
Nel caso in cui la dimensione del dataset a disposizione sia estremamente limitata, é possibile adottare un approccio radicale chiamato **leave-one-out cross-validation**, o **LOOCV**. L'approccio é di fatto un n-fold cross-validation dove la dimensione del test set é unitaria, ovvero se il dataset é composto da n elementi, il sotto-dataset utilizzato per la costruzione del modello ha dimensione $n-1$ mentre quello utilizzato per il testing

ha dimensione 1. Come per il caso precedente, si ripete l'operazione n volte e si ricava l'accuratezza complessiva a partire dalla media delle n accuratezze parziali.
Talvolta, trasversalmente a questi approcci, viene introdotto un terzo sotto-dataset, chiamato **validation set**. Questo viene utilizzato per valutare le prestazioni degli **iperparametri**, ovvero i parametri che determinano la scelta stessa del tipo di modello. Il modello i cui iperparametri sono i piu performanti viene poi allenato con il training set e testato con il test set come di consueto.

5.2 Apprendimento con modello: alberi di decisione

L'**apprendimento mediante alberi di decisione** e un esempio di model-based supervised learning pensato per la risoluzione di problemi di classificazione. Il modello generato da tale apprendimento e, per l'appunto, una struttura ad albero chiamato **albero di decisione**.

Sia i nodi interni che la radice di un albero di decisione sono etichettati con gli attributi $A_1, A_2, ..., A_m$ del dataset D usato per costruirlo. Da ogni nodo interno e dalla radice si diramano tanti archi quanti sono i valori che l'attributo del nodo da cui escono puo assumere, ciascuno etichettato con uno dei suddetti valori. I nodi foglia sono invece etichettati con `True` oppure con `False`. Si noti come possano esserci piu noti interni etichettati con lo stesso attributo.
Un percorso che va dalla radice ad un nodo foglia corrisponde ad uno degli elementi del dataset usato per costruire l'albero: ciascun arco che viene attraversato corrisponde al valore del relativo attributo, mentre il nodo foglia indica se l'elemento in questione appartiene oppure non appartiene alla classe. Quando all'algoritmo viene fornito un dato generico, questo percorre l'albero seguendo il percorso descritto dai suoi attributi ed emette un verdetto di conseguenza.
Ciascun percorso che va dalla radice ad un nodo foglia può essere tradotto in una formula equivalente nella logica proposizionale:

Se attributo1 vale ... e attributo2 vale ... e ... e attributo k vale ... allora appartiene / non appartiene alla classe

Formalmente, siano $A_1, A_2, ..., A_k$ gli attributi che figurano come etichette dei nodi non foglia lungo un certo percorso, e siano $a_1, a_2, ..., a_k$ i valori che questi assumono (letti lungo gli archi dell'albero). Sia poi b l'etichetta del nodo foglia. Si ha:

$$a_1 \wedge a_2 \wedge ... \wedge a_k \rightarrow b$$

Esistono diversi alberi di decisione che rappresentano la stessa tabella, alcuni più efficienti di altri. Trovare l'albero di decisione *migliore* per una tabella, ovvero quello avente il minimo numero di nodi, è un problema NP-completo. Esistono però algoritmi euristici che permettono di trovare un albero di decisione generico (non necessariamente ottimale) con tempo di esecuzione approcciabile.
É possibile costruire un albero di decisione ricorsivamente mediante un algoritmo greedy di tipo divide-et-impera. L'albero viene costruito in maniera top-down: inizialmente tutti gli individui si trovano nel nodo radice, poi vengono distribuiti lungo i nodi dell'albero operando delle partizioni.

```
procedure MAKE-DECISION-TREE(D, A, T)
1    if (tutti i membri di D appartengono alla stessa classe cj in C) then
2        crea un nodo foglia T avente cj come etichetta
3    else if (A = ∅) then
4     crea un nodo foglia T avente cj come etichetta, dove cj è la classe avente più membri in D
5  else
6        p0 = ENTROPY(D)
7        foreach Ai in {A1, A2, …, An} do
8            pi = ENTROPYAi(D)
9        done
10       Ag <= l'attributo in A1, A2, …, An che ha il massimo guadagno, ovvero massimo p0 - pi
11       if (p0 - pi) < threshold then
12           crea un nodo foglia T avente cj come etichetta, dove cj è la classe avente più membri in D
13       else
14           crea un nodo interno T sulla base di Ag
15           partiziona D in v sotto-dataset disgiunti D1, D2, …, Dv, dove v sono i valori assumibili da Ag
16           foreach Dj in {D1, D2, …, Dv} do
17               if (Dj != ∅) then
18                   crea un nodo Tj figlio di T relativo al j-esimo valore assumibile da Ag
19                   MAKE-DECISION-TREE(Dj, A - {Ag}, Tj)
```

L'algoritmo ha in input tre variabili: `D`, `A` e `T`. La prima rappresenta l'insieme di elementi da classificare nell'iterazione corrente, la seconda l'insieme di attributi non ancora analizzati e la terza la foglia che viene generata come sostituto al restante insieme di elementi a questo livello.
Inizialmente, `D` corrisponde all'intero dataset, mentre `A` corrisponde all'intero insieme di attributi (le etichette delle prime $n-1$ colonne della tabella); ad ogni iterazione, l'insieme `D` viene partizionato, mentre dall'insieme `A` viene estratto un attributo che verra analizzato e poi rimosso.
I casi base figurano nelle righe da 1 a 4, e sono due. Il primo caso (righe 1 e 2) si verifica quando tutti gli elementi di `D` appartengono alla stessa classe: in questo caso la classificazione non ha ambiguità, ed è sufficiente creare un nodo foglia che ha tale classe come etichetta e che contiene l'intero `D`. Il secondo caso (righe 3 e 4) si verifica quando `A` e vuoto ma non lo e `D`, ovvero quando tutti gli attributi sono stati analizzati ma rimangono ancora degli elementi da classificare. Anche in questo caso, l'intero `D` viene inserito nello stesso nodo foglia ma come etichetta viene scelta la classe che contiene il maggior numero di rappresentanti in `D`: questa e comunque una approssimazione, ma e il miglior compromesso possibile.
Se non si ricade in nessuno dei due casi base, significa che `D` non e vuoto (vi sono ancora degli elementi da classificare) ma non lo e nemmeno `A` (vi sono ancora degli attributi da considerare). Occorre allora scegliere un attributo sulla base del quale eseguire la partizione, rimuovere tale attributo e chiamare ricorsivamente l'algoritmo usando come `D` gli elementi rimasti e come `A` gli attributi meno quello appena esaminato.
La scelta dell'attributo influisce notevolmente sulla qualita dell'albero risultante. L'ideale sarebbe scegliere l'attributo che riduce al minimo l'incertezza nel (sott)dataset. A ciascun D è infatti possibile associare una misura di "impurità" chiamata **entropia** [1], che puo essere utilizzata

1. Il termine è associato al concetto analogo in fisica.

per guidare la scelta dell'attributo nell'iterazione corrente. Si dice che l'entropia di `D` e alta se *quasi* tutti i suoi elementi appartengono a classi diverse, mentre si dice l'entropia di `D` e bassa se *quasi* tutti i suoi elementi appartengono alla stessa classe. L'entropia e definita come segue:

$$\text{entropy}(D) = -\sum_{j=1}^{|C|} P(c_j)\log_2(P(c_j))$$

Il valore $P(c_j)$ indica la probabilitá che, scelto un elemento casuale dal dataset D, questo appartenga alla classe c_j. Questo valore, moltiplicato per il logaritmo in base due di sé stesso, viene calcolato per ogni classe esistente e sommati fra di loro. Essendo $P(c_j) \in (0, 1)$, si ha che $\log(P(c_j))$ é un numero negativo; questo viene peró reso positivo dal segno meno davanti alla sommatoria.
Il prodotto $P(c_j)\log_2(P(c_j))$ é complessivamente nullo sia nel caso in cui $P(c_j) = 0$, ovvero é certo che non esista alcun elemento di D che appartenga a c_j, sia nel caso in cui $P(c_j) = 1$, ovvero é certo che qualsiasi elemento di D appartiene a c_j. Infatti:

$$\lim_{x\to 0^+} P(c_j)\cdot\log_2(P(c_j)) = \lim_{x\to 0^+}\frac{\log_2(P(c_j))}{\frac{1}{P(c_j)}} = \lim_{x\to 0^+}\frac{\frac{1}{\ln(2)P(c_j)}}{-\frac{1}{P(c_j)^2}} = \lim_{x\to 0^+}\frac{-x}{\ln(2)} = 0 \qquad \lim_{x\to 1^-} P(c_j)\cdot\log_2(P(c_j)) = 1\cdot 0 = 0$$

Questo significa che il contributo portato dalla classe c_j all'entropia complessiva associata a D ha un valore non nullo solamente se la probabilitá che un elemento di D appartenga a c_j é un valore che non é né 1 né 0. In altre parole, c_j fa aumentare l'entropia associata a D solamente se un elemento di D *potrebbe* appartenere a c_j. In particolare, il massimo del contributo all'entropia di D fornito da c_j si ha quando la probabilitá che un elemento di D appartenga a c_j é circa un terzo:

$$\frac{d}{dx}(P(c_j)\log_2(P(c_j))) = 0 \;\Rightarrow\; \log_2(P(c_j)) + \frac{1}{\ln(2)} = 0 \;\Rightarrow\; \log_2(P(c_j)) = \frac{-1}{\ln(2)} \;\Rightarrow\; P(c_j) = 2^{-1/\ln(2)} \approx 0.3679$$

Se nell'iterazione corrente venisse scelto per compiere il partizionamento di D l'attributo A_i, che puó assumere v valori distinti, questo genererá v sottoinsiemi $D_1, D_2, ..., D_v$. É possibile calcolare l'entropia che avrebbe D se si scegliesse di partizionarlo sulla base di A_i come:

$$\text{entropy}_{A_i}(D) = \sum_{j=1}^{v}\frac{|D_j|}{|D|}\text{entropy}(D_j)$$

Dove ciascun termine della sommatoria corrisponde all'entropia nel j-esimo sotto-dataset "pesata" con il rapporto fra la sua dimensione e la dimensione dell'intero D. In questo modo, affinché un sotto-dataset contribuisca considerevolmente al valore totale dell'entropia di D dopo la partizione secondo A_i deve sia avere una sua alta entropia intrinseca sia essere grande (in rapporto all'intero D).
Si noti come, a prescindere da quale attributo venga scelto, $\text{entropy}_{A_i}(D)$ sara sempre inferiore a a $\text{entropy}(D)$. Questo significa che l'entropia di un dataset, mano a mano che i suoi elementi vengono partizionati, si riduce sempre di piu.
La differenza fra l'entropia di D prima che avvenga la partizione ($\text{entropy}(D)$) e l'entropia di D se lo si partizionasse sulla base di A_i ($\text{entropy}_{A_i}(D)$) indica quanta informazione verrebbe "guadagnata" se si scegliesse A_i come attributo per il partizionamento:

$$\text{gain}(D, A_i) = \text{entropy}(D) - \text{entropy}_{A_i}(D)$$

Questo significa che l'attributo che meglio conviene scegliere per partizionare D in una certa iterazione é quello che massimizza il valore di $\text{gain}(D, A_i)$, sia questo A_g. A questo punto (riga 14), `T` diviene un nodo interno dell'albero (un nodo di decisione), mentre (riga 15) l'insieme `D` viene partizionato in tanti sottoinsiemi quanti sono i possibili valori che A_g puo assumere. Dopodiche, l'algoritmo, costruisce un sottoalbero T_j figlio di T per ciascun j-esimo sottoinsieme (riga 17), per poi richiamare ricorsivamente l'algoritmo per ciascuno di questi (righe 18 e 19), passando D_j come primo parametro, tutti gli attributi rimasti meno A_g come secondo e T_j come terzo.
Se pero tale guadagno e troppo piccolo, e preferibile operare come se non vi fossero piu attributi a disposizione e costruire un nodo foglia che contiene tutti gli elementi di `D` etichettato con la classe piu rappresentata (righe 11 e 12). Questo certamente migliora le prestazioni dell'algoritmo (si sacrifica parte della precisione per evitare ulteriori `v` chiamate ricorsive) ma soprattutto previene (in parte) l'overfitting. Infatti, se il partizionamento venisse fatto sempre e comunque, l'albero diverrebbe "troppo preciso" e terrebbe conto anche di variazioni infinitesime nei valori del dataset, che sono invece meglio spiegate dalla semplice presenza di rumore.
In genere, la presenza di overfitting in un albero di decisione e evidente quando l'albero ha troppi livelli e/o dei nodi con troppi figli, perché in genere questo si verifica se il dataset é molto rumoroso e quindi un certo attributo lo partiziona in moltissimi sottoinsiemi ciascuno con pochi rappresentanti. Esistono fondamentalmente due approcci per ridurre l'overfitting in un albero di decisione, chiamati **pre-pruning** e **post-pruning**.
Il pre-pruning prevede di bloccare l'espansione di un nodo di modo che l'albero non cresca, ed e quello che viene fatto alle righe 11 e 12 dell'algoritmo. Il post-pruning opera invece dopo che l'albero e stato costruito, cercando e rimuovendo i rami superflui. Ad esempio, e possibile scelto un massimo livello di profonditá ammissibile e tutto ció che sta al di sotto di questa viene unificato, andando per maggioranza.
Sebbene l'approccio sia stato illustrato per variabili discrete, questo puó essere usato anche per la costruzione di alberi dove il dataset contiene variabili continue. L'idea e quella di suddividere in piu intervalli, non necessariamente della stessa lunghezza, i valori che questa puo assumere, ed usare ciascun intervallo come fosse un valore discreto. Il numero e l'ampiezza di tali intervalli deve essere scelta sulla base della distribuzione statistica dei valori dell'attributo.

Essendo un algoritmo divide-et-impera, il tempo di esecuzione per la costruzione dell'albero di decisione e indicativamente logaritmico. Anche il tempo per un'utilizzo dell'albero e indicativamente logaritmico, perche ad ogni nodo interno (ad ogni decisione) lo spazio di possibilita viene quantomeno dimezzato.

5.3 Valutare modelli di classificazione

Una volta costruito un modello per risolvere un problema di classificazione, si ha interesse a valutarne la qualita, testandolo con uno o piu dataset di prova. Questo puo essere fatto sia mediante delle metriche **quantitative**, che forniscono un numero, che **qualitative**, che forniscono una descrizione. E importante sottolineare che tali metriche si riferiscono esclusivamente ai test set, mai ai training set, perche altrimenti si avrebbe un evidente bias.
Le principali metriche qualitative sono le seguenti:

- **Efficienza**, ovvero sia il tempo di esecuzione necessario per la costruzione del modello (**tempo di induzione**), sia il tempo di esecuzione impiegato dal modello nel venire utilizzato (**tempo di inferenza**);
- **Robustezza**, ovvero quanto bene il modello é in grado di gestire dati rumorosi (non cadere nell'overfitting) e se é in grado di gestire i dati mancanti (e come lo fa);
- **Scalabilitá**, ovvero quanto il modello riesce a contenere la crescita al crescere della dimensione dei dataset;
- **Interpretabilitá**, ovvero quanto il modello é in grado di "spiegare" il suo risultato a chi lo utilizza.
- **Compattezza**, ovvero quanto il modello riesce a descrivere il dataset in maniera conservativa (senza componenti ridondanti);

Per trattare le metriche quantitative e prima necessario introdurre la seguente matrice, detta **matrice di confusione**:

	Classificato come positivo	Classificato come negativo
Effettivamente positivo	TP (**True Positive**): il numero di esempi positivi classificati correttamente	FN (**False Negative**): il numero di esempi positivi classificati erroneamente
Effettivamente negativo	FP (**False Positive**): il numero di esempi negativi classificati erroneamente	TN (**True Negative**): il numero di esempi negativi classificati correttamente

La metrica quantitativa piu intuitiva e l'**accuratezza predittiva** A, ovvero il rapporto fra il numero di classificazioni corrette (sia positive che negative) ed il numero di elementi del dataset usato per il testing:

$$A = \frac{TP + TN}{TP + FP + TN + FN} = frac\text{Numero di classificazioni corrette}\text{Numero totale di elementi del test set}$$

Naturalmente, l'accuratezza predittiva é tanto maggiore quanto piú il suo valore tende ad 1.
L'accuratezza predittiva e piuttosto omnicomprensiva, nel senso che riassume tutti gli aspetti del dataset in un solo numero. Talvolta, si preferisce avere metriche quantitative che si riferiscono ai soli esempi positivi. A tale scopo, e possibile definire le metriche **precision** p (anche detta **sensitivity**) e **recall** r:

$$p = \frac{TP}{TP + FP} = \frac{\text{Esempi positivi classificati correttamente}}{\text{Esempi classificati positivi}} \qquad r = \frac{TP}{TP + FN} = \frac{\text{Esempi positivi classificati correttamente}}{\text{Esempi effettivamente positivi}}$$

p rappresenta quanto bene il modello é in grado di classificare correttamente gli esempi positivi (tralasciando i falsi negativi), mentre r rappresenta quanto il modello é in grado di "coprire" i dati (quanto poco tralascia gli esempi positivi, anche a costo di commettere un errore).
Per comoditá, é possibile combinare le due metriche in una sola, chiamata **F_1-value** (o **F_1-score**), che non é altro che la loro media armonica:

$$F_1 = \left(\frac{p^{-1} + r^{-1}}{2}\right)^{-1} = \frac{2}{p^{-1} + r^{-1}} = \frac{2}{\frac{1}{p} + \frac{1}{r}} = \frac{2pr}{p + r}$$

Questa metrica é di particolare interesse perché la media armonica di due valori tende ad essere vicina al piú piccolo dei due. Inoltre, dato che p e q compaiono sia al numeratore che al denominatore, il valore di F_1 é grande solamente se sia p che q sono a loro volta grandi.
Un'ulteriore metrica e la cosiddetta **specificity** s, definita come il rapporto fra il numero di esempi effettivamente negativi ed il numero totale di esempi classificati negativi (sia correttamente che non correttamente):

$$s = \frac{TN}{TN + FP} = \frac{\text{Esempi effettivamente negativi}}{\text{Esempi classificati negativi}}$$

Questa e la "controparte" di p rispetto agli esempi negativi.

5.4 Apprendimento senza modello: K-nearest neighbour

K-nearest neighbour (**kNN**) e una tecnica di individual-based learning che permette di classificare un dataset con il solo requisito di avere a disposizione una metrica per definire una distanza fra gli elementi di un dataset.
Si assuma di avere a disposizione un dataset D giá parzialmente classificato. Fissato un numero di vicini k e scelta una metrica opportuna per gli elementi di D, l'algoritmo procede come segue:

1. Preso un elemento $d \in D$ non ancora classificato, si calcoli la distanza fra d e tutti gli altri elementi di D;
2. Si costruisca l'insieme $N \subseteq D$ formato dai k elementi di D che hanno la piú piccola distanza da d;
3. Sia c la classe che figura piú spesso fra gli elementi di N. Se si verifica un conflitto, ovvero se esistono due o piu classi che figurano piu di tutte le altre in N ma che hanno fra loro la stessa cardinalita, c viene scelta come una qualsiasi di queste. All'elemento d viene assegnata la classe c;
4. Se esiste ancora almeno un elemento $d' \in D$ non classificato, l'algoritmo riparte considerando d'. Altrimenti, l'algoritmo termina.

L'idea dell'algoritmo é di stimare $P(c \mid d)$, ovvero la probabilitá che la classificazione corretta sia scegliere la classe c dato l'individuo d, con $absP \ / \ k$, ovvero il rapporto fra il numero di vicini di d con piú rappresentanti ed il numero totale di vicini di d. Essendo N un insieme estratto dai k vicini di d, il valore $absN \ / \ k$ é certamente compreso fra 0 e 1, ed é quindi un valore di probabilitá.
Il valore di k viene in genere scelto in maniera empirica, operando ad esempio un k-fold cross validation ed osservando quale valore di k rende i risultati migliori. É preferibile scegliere un numero dispari come valore di k, perché in questo modo é piú raro che possa verificarsi una situazione di conflitto.
La funzione di distanza dipende dal dominio del dataset in esame. Nel caso limite in cui gli attributi del dataset siano valori numerici e vada loro assegnato lo stesso peso, e possibile utilizzare la **distanza di Minkowski**. Indicando con p e il numero di attributi del dataset e con $\boldsymbol{d}_q$ l'elemento sul quale l'algoritmo kNN sta operando, la distanza di Minkowski fra $\boldsymbol{d}_q$ ed un altro elemento $\boldsymbol{d}_j \in D$ é data da:

$$L^p(\boldsymbol{d}_q, \boldsymbol{d}_j) = \left(\sum_i \left(|\boldsymbol{d}_{q,i} - \boldsymbol{d}_{j,i}| \right)^p \right)^{1/p}$$

Dove $\boldsymbol{d}_{j,i}$ indica l'i-esimo attributo dell'elemento $\boldsymbol{d}_j$.
Non dovendo costruire alcun modello, il tempo di induzione di k-nearest neighbour é, tecnicamente, nullo. D'altro canto, in ciascuna esecuzione dell'algoritmo occorre calcolare la distanza fra ciascun elemento del dataset e tutti gli altri elementi: anche applicando tecniche di programmazione dinamica, non e possibile scendere sotto un tempo di esecuzione lineare nella dimensione del dataset.
Si noti come non su tutti i domini sia possibile definire una nozione di distanza, pertanto in tali situazioni K-nearest neighbour e inapplicabile. Inoltre, anche avendo una distanza a disposizione, il semplice fatto che degli elementi del dataset siano vicini fra loro non implica necessariamente che esista fra loro una qualche correlazione semantica.
Per questi e altri motivi, kNN ha una applicabilitá piú limitata rispetto, ad esempio, agli alberi di decisione, che richiedono molte meno assunzioni. Inoltre, quando il numero di dimensioni é grande, diventa molto difficile trovare dei punti che siano vicini fra di loro. Nonostante questo, le prestazioni di kNN sono comunque competitive, addirittura superando, in certe situazioni, algoritmi molto piú elaborati.

5.5 Metodi ensemble

Fino ad ora sono stati considerati metodi di apprendimento che costruiscono un'ipotesi sulla base della quale vengono fatte delle predizioni. Alcune ipotesi hanno ottima precision e bassa recall (tutti i positivi sono effettivamente positivi, ma alcuni vengono tralasciati), mentre altre hanno ottima recall e bassa precision (non tralascia alcun caso ma parte dei positivi sono falsi positivi). L'idea dei **metodi ensemble** é di costruire una collezione di ipotesi $h_1, h_2, \ldots, h_n$, detta **ensemble**, e combinare le loro predizioni, ottenendo una nuova ipotesi piu preciso di quanto possa esserla ciascuna presa singolarmente.
Questo puó venire fatto in diversi modi, ad esempio calcolando la media sulle predizioni di ciascuna, scegliendone una di volta in volta secondo un sistema di voti oppure con un "ulteriore livello" di apprendimento, che analizza le ipotesi e apprende quali ipotesi tendono ad essere migliori. Ciascuna ipotesi singola prende il nome di **modello base**, mentre la loro combinazione **modello ensemble**.
Si noti come sia del tutto irrealistico assumere che le ipotesi siano fra loro indipendenti, dato che condividono sia gli stessi dati che le stesse assunzioni. Per questo motivo, se un errore é presente nella maggior parte dei modelli base, questo sará presente anche nel modello ensemble. Inoltre, per costruire un modello ensemble é necessario costruire n modelli base, e se il costo in termini di prestazione per la costruzione di un singolo modello é proibitiva, quello per la costruzione di n modelli lo é ancor di piu. Nonostante questo, il miglioramento qualitativo di un modello ensemple potrebbe comunque giustificare l'investimento.

5.5.1 Bagging

Il **Bagging** (contrazione di **Bootstrap AGGregatING**) é un metodo ensemble che prevede di generare diversi training set distinti da uno di partenza mediante estrazioni con reimmissione. Nello specifico, sia D il dataset a disposizione: per k volte, viene costruito un dataset S_i con $1 \leq i \leq k$ scegliendo per $|D|$ volte un elemento qualsiasi di D. Si noti come uno stesso elemento possa trovarsi piu volte all'interno dello stesso S_i.
Ciascun S_i viene utilizzato come training set per un'ipotesi h_i, ottenendo cosi k ipotesi distinte. Quando é necessario testare un input, questo viene valutato su tutte e k le ipotesi ed il giudizio finale é ottenuto combinando i risultati di tutte. Per un problema di classificazione, questo consiste nello scegliere il risultato su cui la maggior parte delle ipotesi concordano.
Il bagging puó essere applicato a qualsiasi classe di ipotesi, ma é particolarmente efficiente nei modelli **instabili** (come gli alberi di decisione), ovvero i modelli dove una piccola variazione nel training set comporta una variazione consistente nel modello. Nei modelli **stabili** (come k-nearest neighbour), il bagging puó addirittura generare un ensemble di ipotesi con una performance peggiore dei singoli modelli base.

5.5.2 Boosting

Il **Boosting** é la tecnica di ensemble learning piú popolare. Dato un dataset D, questo viene esteso aggiungendo a ciascun j-esimo elemento un peso w_j, che indica quanto tale elemento deve essere rilevante nel training dell'ipotesi. Un dataset i cui elementi hanno associato un peso é detto **dataset pesato**.
Inizialmente, a tutti gli elementi di D é associato un peso pari ad 1. Viene costruita una prima ipotesi h_1 usando un certo algoritmo di apprendimento; questa inevitabilmente classificherá incorrettamente una parte (idealmente piccola) del dataset. A tali dati viene incrementato il peso e viene costruita una nuova ipotesi sul dataset cosí modificato. Il procedimento viene ripetuto per k volte, con k fissato, generando k ipotesi. I valori di D difficili da classificare aumenteranno costantemente di peso fino a quando l'algoritmo non sará costretto a prendere in considerazione tali valori e generare un'ipotesi che la classifichi correttamente.
Il modello ensemble cosí costruito classifica i dati sulla base dei voti dei modelli base, come nel bagging, ma in questo caso i voti sono pesati: alle ipotesi che hanno performato meglio sui rispettivi training set vengono dati piú voti. Indicando con z_i il peso assegnato a ciascuna ipotesi, il risultato finale é dato da:

$$h(\boldsymbol{x}) = \sum_{i=1}^{K} z_i h_i(\boldsymbol{x})$$

L'idea alla base del boosting é implementata in diversi algoritmi. Fra questi, ADABOOST é quello in genere utilizzato quando i modelli base sono alberi di decisione. ADABOOST possiede una importante proprietá: se l'algoritmo di apprendimento che costruisce l'ipotesi é un **algoritmo di apprendimento debole**, ovvero che l'algoritmo restituisce una ipotesi la cui accuratezza sul training set é leggermente migliore dello scegliere a caso, allora il modello ensemble restituito da ADABOOST classifica i dati perfettamente se il numero di modelli base é sufficientemente grande. Sia dato un insieme di m esempi $\langle (x_1, y_1), ..., (x_m, y_m) \rangle$, con etichette $y_i \in Y = \{1, ..., k\}$. Fissato un numero di iterazioni T, l'algoritmo é il seguente:

1. Si inizializzi $D_1(i) = 1 \,/\, m$ per ciascun i;
2. Si inizializzi t ad 1;
3. Si costruisca un modello base h_t a partire dal testing set D_t invoncando un algoritmo di apprendimento debole;
4. Si calcoli ε_t, l'errore commesso da h_t su D_t:

$$\varepsilon_t = \sum_{h_t(x_i) \neq y_i} D_t(i)$$

5. Se $\varepsilon_t > 0.5$, allora viene impostato $T = t-1$ e si salta immediatamente all'ultimo punto;
6. Sia $\beta_t = \varepsilon_t(1-\varepsilon_t)$;
7. Si costruisca il dataset $D_{t+1}(i)$ da utilizzare per l'iterazione successiva:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{se } h_t(x_i) = y_i \\ 1 & \text{altrimenti} \end{cases}$$

   Dove Z_t é una costante di normalizzazione scelta di modo che D_{t+1} sia ancora una distribuzione;
8. Se $t < T$, si pone $t = t + 1$ e l'algoritmo riprende dal punto 3. Altrimenti, il modello ensemble é cosí costruito:

$$h_{\text{fin}}(x) = \text{argmax}_{y \in Y} \left(\sum_{h_t(x)=y} \log\left(\frac{1}{\beta_t}\right) \right)$$

Come nel caso del bagging, il boosting e particolarmente indicato quando le ipotesi base sono instabili. Si noti inoltre come il boosting sia molto suscettibile alla presenza di rumore, perche agli esempi molto distanti dalla varianza potrebbe venire data molta priorita.

Capitolo 6
Unsupervised Learning

6.1 Introduzione all'apprendimento non supervisionato

Il **supervised learning** e una tecnica di apprendimento in cui l'agente apprende sia sulla base dell'input che sulla base dell'output. Il nome "supervised" viene dal fatto che tale tecnica necessita di dei dati che presentano gia la caratteristica che l'agente deve individuare, in modo da apprendere quali sono i pattern che portano un dato a possedere tale caratteristica. Il supervised learning prevede sia una fase di **addestramento**, in cui i pattern vengono individuati, ed una fase di **testing**, in cui si valuta se l'algoritmo e in grado di prevedere il valore della caratteristica su dati ignoti. Naturalmente, e necessario assumere che i dati noti ed i dati ignoti abbiano la stessa distribuzione statistica.
Piu formalmente, sia D un insieme di n elementi. Ciascun elemento possiede m attributi $A_1, A_2, ..., A_m$ ed e membro di una classe. Ciascuno di questi elementi puo essere rappresentato da una m + 1-upla, ovvero come $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)$. $\boldsymbol{x}_i$ rappresenta i valori degli attributi dell'i-esimo elemento, mentre y_i rappresenta la classe a cui l'elemento appartiene. Il valore del j-esimo attributo dell'i-esimo elemento viene indicato con $x_{i,j}$.
Si assuma l'esistenza di un legame fra i valori di $\boldsymbol{x}_i$ e quelli di y_i. Deve allora esistere una funzione (ignota) f tale che $f(\boldsymbol{x}_i) = y_i$, ovvero in grado di determinare qual'e il valore di y_i anche se questo non e noto, solamente sulla base di $\boldsymbol{x}_i$.
Il supervised learning costruisce una funzione h, chiamata **ipotesi**, che cerca di stimare f osservando i pattern che sussistono fra i valori di $\boldsymbol{x}_i$ e quelli di y_i. In questo modo, se viene fornito in input ad h una m-upla qualsiasi, questa funzione e in grado di determinare il valore di y_i anche se questo e ignoto. Il supervised learning apprende sia dall'input che dall'output in questo senso: l'input ($\boldsymbol{x}_i$) e l'output (y_i) sono riferiti alla funzione h, ed entrambi sono necessari alla sua costruzione.
Il problema risolto dal supervised learning prende una propria denominazione in base a qual'e il tipo di dato di y_i. Se y_i e un valore numerico, il problema e detto **problema di regressione**, in analogia con il concetto di regressione matematica. Se y_i e un valore booleano, il problema e detto **problema di classificazione**, perche e possibile interpretare y_i come una variabile che indica se l'i-esimo elemento di D appartiene (`True`) o non appartiene (`False`) ad una certa classe. Per convenzione, gli elementi che hanno `True` come valore di tale attributo sono detti **elementi positivi**, mentre mentre quelli che hanno `False` vengono detti **elementi negativi**.
Il tipo di dato degli attributi determina solamente se questi possano essere usati come input di un determinato algoritmo oppure no. Infatti, alcuni metodi di apprendimento operano esclusivamente su attributi di tipi specifici, mentre altri possono operare su attributi eterogenei. E comunque possibile, in genere, convertire un attributo di un tipo in uno o piu attributi equivalenti mediante una **codifica**.
Una codifica molto semplice prende il nome di **one hot encoding**, applicabile quando il dominio dell'attributo generico e discreto e non troppo grande. Questa prevede di sostituire l'attributo con tante variabili binarie quanti sono i valori che l'attributo puo assumere: se un elemento del dataset aveva come i come valore dell'attributo, allora la i-esima di queste variabili binarie avra valore 1 e tutte le rimanenti valore 0.

> Un esempio di supervised learning si ha nei filtri antispam dei client di posta elettronica. L'idea é quella di fornire al filtro un grande quantitativo di email contrassegnate come spam ed un grande quantitativo di email contrassegnate come non spam, di modo che questo possa estrarre dei pattern comuni nelle email spam e non spam. A questo punto, se viene fornita al filtro una mail qualsiasi, questo é (dovrebbe essere) in grado di determinare autonomamente se la mail é o non é spam.

Vi sono due situazioni patologiche molto comuni che possono presentarsi nella costruzione di un modello supervisionato, **underfitting** e **overfitting**. La prima si verifica quando il modello costruito predice male i dati passati, mentre la seconda si verifica quando il modello predice "cosi bene" i dati su cui e allenato da non tollerare alcun grado di impurita, tanto da fallire per molti dataset solo leggermente differenti Naturalmente, la prima situazione e peggiore della seconda, perche se l'overfitting produce un modello scadente ma comunque funzionante, l'underfitting produce un modello del tutto inutilizzabile.
Esistono sia algoritmi che costruiscono un modello, e che quindi distinguono fra fase di training e fase di test, chiamati **model-based learning**, sia algoritmi che non lo costruiscono, dove tutto e insieme, chiamati **individual-based learning**.
La tecnica piú semplice per allenare un dataset prende il nome di **holdout set**. Questa prevede di separare il dataset in due sotto-dataset: uno, chiamato **training set**, verrá usato per allenare il modello, mentre l'altro, chiamato **test set**, verra usato per testarlo. Questa tecnica e applicabile quando la dimensione del dataset a disposizione é sufficientemente grande da avere due sotto-dataset a loro volta grandi. Naturalmente, gli elementi del training set non devono figurare nella fase di test, cosi come gli elementi del test set non devono figurare nella fase di apprendimento, perche in entrambi i casi il modello avrebbe un evidente bias. Separare nettamente i due sotto-dataset permette di avere un modello funzionante ma che al contempo tollera un minimo margine di errore.
Una tecnica alternativa, chiamata **n-fold cross-validation**, é preferibile quando la grandezza del dataset a disposizione é limitata. Questa prevede di partizionare il dataset a disposizione in m sotto-dataset di uguale grandezza. Per m volte, viene scelto uno degli m sotto-dataset come test set ed i restanti $m-1$ sotto-dataset vengono combinati in un training set. A ciascuna iterazione della procedura e possibile associare una accuratezza; l'accuratezza complessiva viene calcolata come media di tutte le m accuratezze cosí ottenute. In genere, sono comuni partizionamenti in 5 (5-fold cross-validation) o in 10 (10-fold cross validation) sotto-dataset.
Nel caso in cui la dimensione del dataset a disposizione sia estremamente limitata, é possibile adottare un approccio radicale chiamato **leave-one-out cross-validation**, o **LOOCV**. L'approccio é di fatto un n-fold cross-validation dove la dimensione del test set é unitaria, ovvero se il dataset é composto da n elementi, il sotto-dataset utilizzato per la costruzione del modello ha dimensione $n-1$ mentre quello utilizzato per il testing

ha dimensione 1. Come per il caso precedente, si ripete l'operazione n volte e si ricava l'accuratezza complessiva a partire dalla media delle n accuratezze parziali.
Talvolta, trasversalmente a questi approcci, viene introdotto un terzo sotto-dataset, chiamato **validation set**. Questo viene utilizzato per valutare le prestazioni degli **iperparametri**, ovvero i parametri che determinano la scelta stessa del tipo di modello. Il modello i cui iperparametri sono i piu performanti viene poi allenato con il training set e testato con il test set come di consueto.

6.2 Clustering basato su partizioni: K-means

Il **clustering basato su partizioni** suddivide un dataset in insiemi sempre piu piccoli. In questo tipo di clustering il dataset viene preso in esame per intero, a prescindere da quanto questo sia rumoroso. Questo semplifica il procedimento, perché non e necessario alcun preprocessing, ma allo stesso tempo rende i cluster molto suscettibili al rumore del dataset, in particolare, agli **outlier**, i dati isolati molto distanti dal resto;
k-means e un esempio di algoritmo di clustering basato su partizioni, che opera su dati esclusivamente numerici. L'algoritmo partiziona il dataset fornito in k cluster, con k fissato. Ciascun cluster ha un baricento, chiamato **centroide** (non necessariamente un elemento del dataset). L'algoritmo é presentato di seguito:

1. Sia scelga un valore k;
2. Si scelgano k elementi qualsiasi a partire dal dataset (detti **seed**): questi saranno i centroidi iniziali dei k cluster;
3. Per ciascun elemento del dataset che non é un centroide, si calcoli la distanza fra tale elemento e tutti i centroidi. L'elemento viene assegnato alla partizione il cui centroide ha la piú piccola distanza da questo. La distanza fra un elemento $\boldsymbol{x}_i$ ed un centroide $\boldsymbol{m}_j$ é data dalla consueta formula:

$$\text{dist}\left(\boldsymbol{x}_i, \boldsymbol{m}_j\right) = \|\boldsymbol{x}_j - \boldsymbol{m}_j\| = \left(\left(x_{i,1} - m_{j,1}\right)^2 + \left(x_{i,2} - m_{j,2}\right)^2 + \ldots + \left(x_{i,r} - m_{j,r}\right)^2\right)^{1/2}$$

4. Si ricalcolino i centroidi sulla base dell'assegnazione ai cluster cosí effettuata. Naturalmente, nello spazio euclideo, la media di un cluster é data dalla media aritmetica dei suoi valori:

$$\boldsymbol{m}_j = \frac{1}{|C_j|} \sum_{\boldsymbol{x}_i \in C_j} \boldsymbol{x}_j$$

5. Se é stato raggiunto un criterio di terminazione, l'algoritmo termina. Altrimenti, si riprende dal punto 3.

L'algoritmo non specifica un criterio di terminazione. Un criterio molto semplice consiste nel fissare un ε e valutare di quanto si discosta il nuovo valore dei centroidi (calcolato al punto 4) dal valore precedente: se questo scostamento é inferiore ad ε, l'algoritmo termina. Un criterio simile prevede di fissare un ε e di terminare l'algoritmo se il numero di elementi che vengono assegnati ad un cluster diverso alla fine della corrente iterazione é inferiore ad ε. Un terzo criterio, piu raffinato, e definito a partire dalla funzione **Sum of Squared Error**, o **SSE** (*Somma degli Errori Quadratici*):

$$\text{SSE} = \sum_{j=1}^{k} \sum_{\boldsymbol{x} \in C_j} \text{dist}\left(\boldsymbol{x}, \boldsymbol{m}_j\right)^2$$

Dove C_j indica il j-esimo cluster, $\boldsymbol{m}_j$ il centroide del cluster C_j e $\text{dist}\left(\boldsymbol{x}, \boldsymbol{m}_j\right)$ la distanza fra il punto $\boldsymbol{x}$ ed il centroide $\boldsymbol{m}_j$. L'idea e quella di fissare un ε e fermare l'algoritmo quando SSE calcolato nell'iterazione corrente si discosta meno di ε dal valore dell'iterazione precedente.
K-means é efficiente, dato che il suo tempo di esecuzione é $O(tkn)$, dove k é il numero di cluster, t é il numero di iterazioni e n é il numero di elementi del dataset; essendo k fissato e t (generalmente) piccolo, il tempo di esecuzione di k-means é quasi-lineare.
Idealmente, un cluster é considerabile un buon cluster quando ha sia una alta **coesione intra-cluster**, ovvero quando é un **cluster compatto** che una alta **coesione inter-cluster**, ovvero quando é un **cluster isolato**. Un cluster si dice compatto quando é piccola la distanza che hanno tutti i punti del cluster dal loro centroide, mentre si dice isolato quando é grande la distanza di ogni punto da tutti i punti dei cluster diversi dal proprio.
SSE e un esempio di statistica per la coesione intra-cluster. Per quanto riguarda la coesione inter-cluster, si consideri un elemento $\boldsymbol{x}_i$, che é stato assegnato al cluster C_I. Sia $A(\boldsymbol{x}_i)$ il valore del centroide del cluster a cui $\boldsymbol{x}_i$ appartiene, e sia invece $B(\boldsymbol{x}_i)$ la minima distanza media fra $\boldsymbol{x}_i$ e tutti i punti di D che non si trovano in C_I. Il cluster che ha tale distanza é detto **neighbouring cluster**, perché é il cluster in cui sarebbe piú ragionevole inserire $\boldsymbol{x}_i$ ad eccezione di C_I, essendo quello a questo piú vicino.

$$A(\boldsymbol{x}_i) = \boldsymbol{m}_I = \frac{1}{|C_I|} \sum_{\boldsymbol{x}_i \in C_I} \boldsymbol{x}_i \qquad B(\boldsymbol{x}_i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{\boldsymbol{x}_j \in C_J} \text{dist}\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$$

$A(\boldsymbol{x}_i)$ é una misura di quanto un elemento del dataset é vicino al centroide del cluster a cui appartiene, mentre $B(\boldsymbol{x}_i)$ é una misura di quanto un elemento del cluster é dissimile dagli elementi degli altri cluster. Pertanto, $\boldsymbol{x}_i$ si trova in un cluster adatto se $A(\boldsymbol{x}_i)$ é un valore piccolo mentre $B(\boldsymbol{x}_i)$ é un valore grande.
Prende il nome di **Silhouette** associata a i la quantitá $S(\boldsymbol{x}_i)$ cosí calcolata:

$$S(\boldsymbol{x}_i) = \begin{cases} 1 - A(\boldsymbol{x}_i) \,/\, B(\boldsymbol{x}_i) & \text{se } A(\boldsymbol{x}_i) < B(\boldsymbol{x}_i) \\ 0 & \text{se } A(\boldsymbol{x}_i) = B(\boldsymbol{x}_i) \\ B(\boldsymbol{x}_i) \,/\, A(\boldsymbol{x}_i) - 1 & \text{se } A(\boldsymbol{x}_i) > B(\boldsymbol{x}_i) \end{cases}$$

É facile verificare che $S(\boldsymbol{x}_i)$ é un valore strettamente compreso fra -1 e 1. Affinché $S(\boldsymbol{x}_i)$ sia vicino ad 1, $A(\boldsymbol{x}_i)$ deve essere un valore piccolo e $B(\boldsymbol{x}_i)$ deve essere un valore grande, pertanto se $S(\boldsymbol{x}_i) \approx 1$ allora $\boldsymbol{x}_i$ é stato ben classificato. Se invece $S(\boldsymbol{x}_i) \approx -1$, allora $A(\boldsymbol{x}_i)$ é grande e $B(\boldsymbol{x}_i)$ é piccolo, e quindi la classificazione é scadente. Se invece $S(\boldsymbol{x}_i) \approx 0$, allora $A(\boldsymbol{x}_i) \approx B(\boldsymbol{x}_i)$, e quindi l'elemento $\boldsymbol{x}_i$ potrebbe indifferentemente appartenere al suo cluster o al neighbouring cluster.
Si noti come K-means sia applicabile solamente a dataset i cui elementi hanno attributi con valori numerici; se non lo sono, occorre codificarli in attributi numerici equivalenti, e non c'e garanzia che la semantica possa essere mantenuta. Inoltre, k-means é applicabile ai soli dataset sui quali é possibile definire sia una distanza che una media fra i suoi elementi.
Dato che ogni elemento ha lo stesso peso nel computo della distanza dai centroidi, la presenza degli outlier destabilizza notevolmente il modo in cui i cluster vengono costruiti. Il problema puó essere mitigato operando anomaly detection sul dataset, prima di operare k-means, di modo da individuare quanti piú outlier possibili ed eliminarli. Oppure, in particolar modo se il dataset é molto grande e gli outlier non sono (o si assume che non siano) molti, é possibile estrarne un sottoinsieme mediante random sampling ed applicare k-means su questo, di modo da ridurre il piú possibile l'eventualitá che il sottoinsieme contenga un outlier.
Inoltre, il valore di k non dipende da una qualche proprieta intrinseca nel dataset, ma viene scelto arbitrariamente: una scelta scadente di k porta ad un clustering a sua volta scadente. Naturalmente e possibile ripetere piu volte l'algoritmo usando valori di k diversi ed osservando quale di questi fornisce il clustering piu convincente. Un problema simile e quello della scelta dei seed: cambiando seed si puo ottenere un clustering completamente diverso. Questo significa che due esecuzioni di k-means sullo stesso dataset e con lo stesso k possono dare risultati completamente diversi.
Infine, il raggruppamento di piú elementi sulla base di una distanza genera dei cluster che sono necessariamente delle iper-ellissi r-dimensionali con il centroide al loro centro. Tuttavia, non tutti i dataset si adattano a venire partizionati in iper-ellissi (o in generale a cluster che sono insiemi convessi), ed in questo caso k-means non sará mai in grado di fornire dei cluster che ben partizionano tale dataset.
Nonostante tutti i difetti sopra citati, k-means (e le sue varianti) rimane comunque l'algoritmo piú utilizzato per risolvere il problema del clustering grazie alla sua semplicitá ed alla sua efficienza. Inoltre, non sembrano esserci prove che un algoritmo di clustering sia migliore degli altri a prescindere dal dataset: in genere, le loro performance dipendono dalla forma del dataset e dal tipo dei loro attributi.

ID	X	Y
1	35.19	12.189
2	26.288	41.718
3	0.376	15.506
4	26.116	3.963
5	25.893	31.515

ID	X	Y
6	23.606	15.402
7	28.026	15.47
8	26.36	34.488
9	23.013	36.213
10	27.819	41.867

ID	X	Y
11	39.634	42.23
12	35.477	35.104
13	25.768	5.967
14	-0.684	21.105
15	3.387	17.81

ID	X	Y
16	32.986	3.412
17	34.258	9.931
18	6.313	29.426
19	33.899	37.535
20	4.718	12.125

6.3 Clustering basato su densitá: DBSCAN

Il **clustering basato su densitá** prevede di costruire dei cluster a partire da un dataset sulla base di come questi sono aggregati. I cluster sono regioni di spazio densamente popolate, separate da spazio poco popolato, di forma del tutto arbitraria.
Sia p un punto n-dimensionale e siano ε e MinPts due numeri strettamente positivi fissati. A partire da una data nozione di distanza, si definisce ε-**neighbourhood** (o ε-**vicinato**) l'insieme $N_\varepsilon(p)$ costituito da tutti i punti q che distano meno o pari a ε da p:

$$N_\varepsilon(p) = \{q \mid d(p,q) \leq \varepsilon\}$$

Si dice che un punto p ha una densitá alta se $N_\varepsilon(p)$ contiene almeno MinPts punti. Sulla base di MinPts é possibile classificare i punti di un insieme in tre categorie:

- Se un punto ha piú punti di MinPts nel suo ε-vicinato, é detto **core point**. Un core point é un punto che verrá scelto come centroide di un cluster;
- Se un punto ha meno punti di MinPts nel suo ε-vicinato ma si trova nell' ε-vicinato di un core point, é detto **border point**;
- Se un punto non é né un core point né un border point, é detto **noise point**. Un noise point é considerato rumore, un punto "non interessante" che verrá escluso dal clustering.

Un punto q é detto **direttamente raggiungibile** a partire da p se mathp é un core point e q si trova nell'ε-vicinato di p. Se un punto r é direttamente raggiungibile a partire da q e q é direttamente raggiungibile a partire da un punto p allora si dice che r é **indirettamente raggiungibile** a

partire da p (a prescindere che r sia direttamente raggiungibile da p o meno). Si noti come la raggiungibilitá, sia diretta che indiretta, non é una proprietá necessariamente simmetrica.
Un algoritmo di clustering basato su densitá molto semplice é **DBSCAN**. Dato un dataset D e fissati due valori strettamente positivi ε e MinPts, a ciascun elemento p di D é possibile associare un tipo: `not visited`, `visited` oppure `noise`. L'algoritmo é presentato di seguito:

```
DBSCAN(D, ε, MinPts)
    C ← nuovo cluster
    foreach p ∈ D do
        if p.type = "not visited" then
        p.type ← "visited"
        NeighbourOfP ← REGION-QUERY(p, ε)
        if (|NeighbourOfP| < MinPts) then
            p.type ← "noise"
        else
            C ← nuovo cluster
            EXPAND-CLUSTER(p, NeighbourOfP, C, ε, MinPts)
```

```
EXPAND-CLUSTER(p, NeighbourOfP, C, ε, MinPts)
    C ← C ∪ {p}
    foreach q ∈ NeighbourOfP do
        if (q.type = "not visited") then
            q.type ← "visited"
            NeighbourOfQ ← REGION-QUERY(q, ε)
            if (|NeighbourOfQ| >= MinPts) then
                NeighbourOfP ← NeighbourOfP ∪ NeighbourOfQ
        if (q non é membro di alcun cluster) then
            C ← C ∪ {q}


REGION-QUERY(p, ε)
    return tutti i punti nell'ε-vicinato di q, compreso p stesso
```

Quando `DBSCAN` viene invocato, viene inizializzato un cluster `C`, dopodiché viene iterativamente esaminato ogni elemento `p` del dataset `D` di tipo `not visited`. All'elemento `p` viene innanzitutto cambiato tipo in `visited`, dopodiché viene costruito l'ε-vicinato di tale elemento. Se tale insieme contiene meno elementi di `MinPts`, allora quel punto é certamente un noise point. Questo perché da una parte é troppo isolato per essere un core point, ma d'altra parte non é stato ancora visitato, e quindi non si trova nell'ε-vicinato di nessun altro punto, pertanto non puó nemmeno essere un border point.
Se l'ε-vicinato di `p` ha invece abbastanza elementi, allora tale punto deve essere un core point, e viene a tal scopo chiamata la procedura `EXPAND-CLUSTER`. Questa innanzitutto aggiunge `p` al cluster, dopodiché osserva tutti i punti `q` che si trovano nell'ε-vicinato di `p`. Se `q` é di tipo `not visited`, viene cambiato il loro tipo in `visited` e si osserva l'ε-vicinato di `q` a sua volta. Se l'ε-vicinato di `q` contiene piú elementi dell'ε-vicinato di `p`, i due insiemi vengono uniti, perché gli elementi dell'ε-vicinato di `q` sono indirettamente raggiungibili a partire da `p`. Se `q` non appartiene ad alcun cluster, allora viene aggiunto al cluster in esame.
I valori di ε e di MinPts devono essere scelti con cura, dato che influenzano di molto il clustering che ne risulta. Un valore di ε o di MinPts troppo piccolo potrebbe indurre un clustering dove quasi tutti i punti sono considerati noise point, e quindi dove quasi nessun punto viene effettivamente preso in considerazione. Un valore di ε o di MinPts troppo grande potrebbe indurre un clustering dove quasi tutti i punti sono inclusi nello stesso cluster. I valori dei parametri devono essere ricavati a partire dai dati stessi.
Come regola pratica, MinPts deve essere almeno pari al numero di attributi degli oggetti piú uno. Piú nello specifico, una scelta sicura per MinPts é il doppio del numero degli attributi, ma per dataset particolarmente grandi e/o rumorosi un valore maggiore si rivela essere una scelta migliore. Si noti come si scegliesse MinPts = 1, tutti i punti verrebbero identificati come core point, pertanto il clustering non avrebbe alcun senso.
Il valore di ε puó essere stimato costruendo un **k-distance plot**: fissato k come MinPts meno uno, lungo l'asse delle ascisse si riportano gli oggetti ordinati in ordine crescente per distanza dal loro k-esimo vicino, mentre sull'asse delle ordinate la distanza stessa. In genere, una curva costruita sulla base di questi dati ha inizialmente un andamento stabile per poi avere una crescita rapida: il valore di ε é scelto il punto della curva in cui si ha tale variazione di pendenza.
A differenza di altri algoritmi di clustering, come ad esempio K-means, DBSCAN ha una tolleranza al rumore nettamente superiore, ed é inoltre in grado di generare cluster di forma arbitraria (non solo insiemi convessi). Tuttavia, é molto sensibile al modo in cui i parametri ε e MinPts vengono fissati. Inoltre, mentre K-means é un algoritmo con tempo di esecuzione lineare nel numero degli elementi, DBSCAN é piu lento: un'implementazione naive dell'algoritmo calcolare la distanza per ciascuna coppia di elementi, quindi avente tempo quadratico; alcune implementazioni riescono a scendere a $O(n\log(n))$, ma comunque meno performante di K-means. Infine, DBSCAN non e in grado di costruire buoni cluster se la densita dei cluster differisce, perche DBSCAN assume che le regioni di spazio fortemente dense lo siano tutte allo stesso modo.

Capitolo 7
Deep Learning

7.1 Percettrone

Deep Learning é una ampia famiglia di tecniche per il machine learning dove le ipotesi prendono la forma di complessi circuiti algebrici fra loro interconnessi. Il termine "deep" si riferisce al fatto che i circuiti sono in genere organizzati in strati detti **layer**, il che significa che i percorsi computazionali dagli input agli output sono costituiti da diversi step.
Il deep learning ha origini nella modellazione matematica dei neuroni del cervello umano sotto forma di circuiti elettrici. Per questo motivo, le reti allenate mediante metodi di deep learning sono spesso anche chiamate **reti neurali** (**neural network**).
L'esempio di rete neurale piú semplice (e storicamente piú datata) é il **percettrone**, una rete neurale in grado di risolvere il problema di classificazione binaria. Questo opera su un input $\boldsymbol{x}$, il quale possiede k features, e determina se tale input appartiene ad una certa classe (é un esempio positivo) oppure se non vi appartiene (é un esempio negativo). Matematicamente, un percettrone é costituito da tre elementi:

- Una funzione $f(\boldsymbol{x})$, che restituisce un vettore k-dimensionale. Ciascuna componente di tale vettore é un numero intero (positivo o negativo) che rappresenta il valore che ha $\boldsymbol{x}$ rispetto a tale feature;
- Un vettore k-dimensionale w, dove ciascuna sua componente é un numero intero che rappresenta il peso da assegnare a ciascuna feature, ovvero quanto quella feature é "rilevante" nel computo della classificazione dell'input;
- Una funzione $\text{out}_w(\boldsymbol{x})$, che restituisce il responso del percettrone.

L'output del percettrone é un esclusivamente +1 oppure -1. Nel primo caso, significa che l'input appartiene alla classe, mentre nel secondo caso che non vi appartiene. Tale output é cosí calcolato:

$$\text{out}_w(\boldsymbol{x}) = \text{sign}\left(\sum_{i=1}^{k} w_i \cdot f_i(x)\right) = \text{sign}(w_1 f_1(x) + \ldots w_k f_k(x))$$

Si noti come la sommatoria nella formula non sia altro che il prodotto scalare fra il vettore k-dimensionale dei pesi w ed il prodotto k-dimensionale delle features $f(x)$. Pertanto, la formula ha anche una interpretazione geometrica: il valore di $\text{out}_w(\boldsymbol{x})$ sará positivo quando l'angolo formato dai vettori w e $f(x)$ é acuto, mentre sará negativo se questo é ottuso.
Sia la funzione $f(x)$ che l'input x stesso possono essere considerate note, ma lo stesso non si puó dire di w. Ovvero, quanto ciascuna feature debba essere "rilevante" agli occhi del percettrone non é necessariamente una informazione nota a priori. É possibile costruire un percettrone in grado di generare w a partire dai dati che gli vengono forniti. Per farlo, si assuma di avere a disposizione n input $x_1, x_2, \ldots, x_n$ per i quali giá é nota la loro classificazione, sia questa rispettivamente $y_1{}^*, y_2{}^*, \ldots, y_n{}^*$. Questo training set puó essere costruito utilizzando una qualsiasi tecnica di training (holdout set, k-fold cross validation, ecc ...).
Sia il vettore w k-dimensionale inizialmente nullo (tutte le sue componenti hanno valore 0). Ciascun input x_i viene classificato sulla base di w: se il risultato fornito dal percettrone coincide con la vera classificazione, ovvero se $\text{out}_w(x_i) = y_i{}^*$, non viene fatto nulla; se invece la classificazione restituita dal percettrone non é corretta, w viene modificato di modo che, se si tenta di riclassificare $\boldsymbol{x}$, il percettrone fornisce la risposta corretta. Nello specifico, w viene sostituito con:

$$w + \left(\sum_{i=1}^{k} w_i \cdot f_i(x)\right) \text{ se } y_i{}^* = +1 \qquad\qquad w - \left(\sum_{i=1}^{k} w_i \cdot f_i(x)\right) \text{ se } y_i{}^* = -1$$

Il motivo per cui questa sostituzione corregge la classificazione va cercata nell'interpretazione geometrica della sommatoria prima citata. Infatti, operando tale sostituzione si garantisce di ottenere un vettore risultante che ha la direzione opposta del precedente, e quindi il risultato $\text{out}_w(\boldsymbol{x})$ viene cambiato di segno.
Si noti peró come un vettore w cosí costruito sará sempre un vettore che passa per l'origine, e questo limita di molto le capacitá del percettrone. Per fare in modo che w si discosti dall'origine é necessario introdurre un **bias**, ovvero una quantitá che ne modifica il valore ma che non ha alcuna correlazione con i valori delle features del dataset in esame. Indicando tale valore con b, si modifica la funzione del percettrone come:

$$\text{out}_w(\boldsymbol{x}) = \text{sign}\left(b + \sum_{i=1}^{k} w_i \cdot f_i(x)\right) = \text{sign}(b + w_1 f_1(x) + \ldots w_k f_k(x))$$

Di conseguenza, per correggere il valore di w durante la sua costruzione sulla base dei dati, viene usata l'espressione:

$$w + b + \left(\sum_{i=1}^{k} w_i \cdot f_i(x)\right) \text{ se } y_i{}^* = +1 \qquad\qquad w - b - \left(\sum_{i=1}^{k} w_i \cdot f_i(x)\right) \text{ se } y_i{}^* = -1$$

Il percettrone é un esempio di **classificatore lineare**, ovvero un classificatore che discrimina gli input sulla base di una combinazione lineare. Nello specifico, il percettrone costruisce una retta nell'iperpiano k-dimensionale che lo partiziona in due regioni: una che contiene tutti gli

elementi positivi ed una che contiene tutti gli elementi negativi. Un dataset per il quale esiste (almeno) una retta avente questa caratteristica é detto **linearmente separabile**, e non tutti i dataset possiedono questa proprietá.

Teorema di convergenza del percettrone. Se un dataset D é linearmente separabile, allora é garantito che un percettrone, compiendo un numero finito di errori, sia in grado di classificarlo.

É possibile estendere il percettrone per permettergli di classificare un dataset in piú classi, fintanto che il loro numero é noto a priori. Si assuma pertanto di avere un dataset i cui elementi sono da suddividere in m classi, enumerate a partire da 1. Un percettrone di questo tipo non ha un solo vettore w, ma bensí m vettori $w_1, w_2, ..., w_m$.
Indicando con b il bias, con k il numero di features e con y' il numero che identifica la classe, si ha:

$$y' = \text{argmax}_y\left(b + \sum_{i=1}^{k} w_{y,i} \cdot f_i(x)\right) = \text{argmax}_y\left(b + w_{y,1} f_1(x) + ... w_{y,k} f_k(x)\right)$$

Ovvero, la classe a cui viene assegnato $\boldsymbol{x}$ é quella che massimizza la somma fra il bias ed il prodotto scalare fra il vettore dei pesi di tale classe e $f(\boldsymbol{x})$.
Per costruire i vettori dei pesi w_y a partire dai dati, si procede come é stato fatto per il percettrone a singola classe, con la differenza che in questo caso occorre correggere i valori di ciascun vettore. Si assuma di avere a disposizione n input $x_1, x_2, ..., x_n$ per i quali giá é nota la classe a cui appartengono, siano queste rispettivamente $y_1^*, y_2^*, ..., y_n^*$. A partire da m vettori $w_1, ..., w_m$ tutti inizialmente nulli, si cerca di classificare ciascun input x_i sulla base di tali vettori. Sia y_i' il risultato del percettrone: se questa coincide con la vera classe a cui x_i appartiene, ovvero se $y_i' = y_i^*$, non viene fatto nulla; se invece la classificazione restituita dal percettrone non é corretta, ciascun vettore w_j viene modificato di modo che, se si tenta di riclassificare $\boldsymbol{x}$, il percettrone fornisce la risposta corretta. Nello specifico:

$$w_j + b + \left(\sum_{i=1}^{k} w_{i,j} \cdot f_i(x)\right) \text{ per la classe } y_i^* \qquad w_j - b - \left(\sum_{i=1}^{k} w_{i,j} \cdot f_i(x)\right) \text{ per tutte le altre}$$

Si noti inoltre come, dato un dataset linearmente separabile, possa esistere piú di una retta in grado di partizionarlo. Fra queste, quella da considerarsi migliore é quella che ha la massima distanza dagli elementi del dataset piú "esterni", ovvero quelli che si trovano piú vicino alla partizione opposta. L'algoritmo per la costruzione di un vettore dei pesi non garantisce di trovare la retta migliore, restituendone invece una qualsiasi (per quanto comunque corretta).
Per avere la garanzia di ottenere sempre la retta migliore, é possibile rifarsi ad un algoritmo chiamato **MIRA** (**Margin Infused Relaxed Algorithm**), che non é altro che un raffinamento dell'algoritmo di generazione dei vettori w. MIRA introduce una costante τ che modifica le componenti dei vettori w in maniera abbastanza incisiva da correggere la classificazione dell'input ma al contempo abbastanza conservativa da non far discostare troppo la direzione di w:

$$w_j + b + \tau\left(\sum_{i=1}^{k} w_{i,j} \cdot f_i(x)\right) \text{ per la classe } y_i^* \qquad w_j - b - \tau\left(\sum_{i=1}^{k} w_{i,j} \cdot f_i(x)\right) \text{ per tutte le altre}$$

Questa costante viene ricavata a partire da:

$$\tau = \min_w \frac{1}{2} \sum_y \left(\|w_y - w_y'\|\right)^2$$

Appendice

7.2 Teoria della probabilita

Siano A e B due eventi, e siano $P(A)$ e $P(B)$ le probabilitá che gli eventi rispettivamente A e B si verifichino. Valgono i seguenti assiomi:

$$0 \leq P(A) \leq 1 \quad P(\text{True}) = 1 \quad P(\text{False}) = 0 \quad P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

É possibile dimostrare che la probabilitá che un evento A avvenga é uguale alla somma tra la probabilitá che sia l'evento A che un certo evento B avvengano e la probabilitá che sia l'evento A che l'evento $\neg B$ avvengano. Questa proprietá é anche detta **formula di disintegrazione**:

$$P(A) = P(A \vee B) + P(A \vee \neg B)$$

Combinando la formula di disintegrazione con la formula per la probabilitá condizionata si ottiene la cosiddetta **formula delle probabilitá totali**:

$$P(A) = P(A \mid B) \cdot P(B) + P(A \mid \neg B) \cdot P(\neg B)$$

$P(A \mid B)$ e $P(B \mid A)$ devono necessariamente soddisfare i due assiomi fondamentali della probabilitá, pertanto dovrá valere:

$$P(A \mid \neg B) = 1 - P(\neg A \mid \neg B) \qquad P(B \mid \neg A) = 1 - P(\neg B \mid \neg A)$$

É importante puntualizzare che $P(A \mid B)$, la probabilitá che A si verifichi sapendo che si é verificato B, non é necessariamente uguale a $P(B \mid A)$, la probabilitá che B si verifichi sapendo che si é verificato A. Le due sono peró collegate dalla **formula di Bayes**:

$$P(X \mid Y) = \frac{P(Y \mid X) \cdot P(X)}{P(Y)}$$

Nel caso in cui $P(Y)$ sia una costante, dato che questa non dipende da X (o da $P(X)$) viene spesso riportata come costante di normalizzazione. In particolare, con $P(Y)^{-1} = \eta$, si ha:

$$P(X \mid Y) = \eta P(Y \mid X) P(X)$$

Tale formula riveste grande importanza nel campo dell'intelligenza artificiale perché é alla base di una tecnica di inferenza statistica chiamata **inferenza Bayesiana**. Data una certa ipotesi, é possibile aggiornarla mano a mano che nuove osservazioni vengono condotte, pesando quanto ciascuna osservazione debba essere presa in considerazione. In tal senso, la formula puó essere interpretata in questo modo:

- X é una ipotesi la cui probabilitá é stata stimata sulla base di un certo numero di osservazioni precedenti;
- $P(X)$ é la **probabilitá a priori**, ovvero la stima della probabilitá di X *prima* di aver integrato l'informazione portata da Y;
- Y é una nuova osservazione, che influirá in maniera piú o meno incisiva sul futuro valore di $P(X)$;
- $P(X \mid Y)$ é la **probabilitá a posteriori**, ovvero la stima della probabilitá di X *dopo* aver integrato l'informazione portata da Y;
- $P(Y \mid X)$ é la **funzione di verosimiglianza**. In funzione di Y con X fissato, indica quanto é compatibile la presenza dell'osservazione Y rispetto all'ipotesi X;
- $P(Y)$ é la **verosimiglianza marginale**, ed indica la probabilitá di osservare Y a prescindere da quale sia l'ipotesi X. Viene anche chiamata semplicemente **evidenza**.

Riassumendo:

$$\text{Posteriori} = \frac{\text{Verosimiglianza} \times \text{Priori}}{\text{Evidenza}}$$

Un'assunzione molto forte che e possibile fare e la cosiddetta **assunzione Markoviana**. In termini molto generali, questa prevede che un processo stocastico che si evolve nel tempo non dipenda da tutte le osservazioni effettuate prima di un un certo istante di tempo, ma solamente da quella immediatamente precedente.
Si consideri la probabilita $P(x \mid z_1, ..., z_n)$, ovvero la probabilita che si verifichi l'evento x sapendo che si sono verificati $z_1, ..., z_n$. Se e valida l'assunzione Markoviana, e possibile applicare la regola di Bayes per esprimere $P(x \mid z_1, ..., z_n)$ come:

$$P(x \mid z_1, ..., z_n) = \frac{P(z_n \mid x)P(x \mid z_1, ..., z_{n-1})}{P(z_n \mid z_1, ..., z_{n-1})} = \eta_n P(z_n \mid x)P(x \mid z_1, ..., z_{n-1}) = \eta_n P(z_n \mid x)\frac{P(z_{n-1} \mid x)P(x \mid z_1, ..., z_{n-2})}{P(z_{n-1} \mid z_1, ..., z_{n-2})}$$
$$= \eta_n P(z_n \mid x)\eta_{n-1}P(z_{n-1} \mid x)P(x \mid z_1, ..., z_{n-2}) = ... = p(x)\prod_{i=1}^{n}\eta_i P(z_i \mid x)$$

Questo e un esempio di inferenza statistica chiamata **filtro Bayesiano**.
Data una variabile aleatoria X, viene detto **valore atteso** (o **valore medio** o **speranza matematica**) di X il valore $E[X]$ cosí calcolato:

$$E[X] = \begin{cases} \sum_{s \in S} sp(s) & \text{se discreta} \\ \int_{-\infty}^{+\infty} uf(u)\,du & \text{se continua} \end{cases}$$

Nel caso in cui X sia una variabile discreta, $E[X]$ é dato dalla sommatoria di tutti i valori che X puó assumere moltiplicati per la probabilitá che assumano quel valore. Se invece X é una variabile aleatoria continua, $E[X]$ é dato dall'integrale calcolato su tutti i punti su cui é definita moltiplicati per la funzione di densitá calcolata in quel punto.
É interessante notare come $E[X]$ sia un valore che dipende dai risultati dell'esperimento a cui é associato, pertanto é esso stesso una variabile aleatoria (e quindi una funzione). Inoltre, il valore medio non é necessariamente uno dei valori assunti dalla variabile aleatoria stessa, e nemmeno é garantito che esista. Nello specifico, questo accade quando la sommatoria o l'integrale da cui viene ricavato non convergono.

Il valore atteso é una funzione lineare: prese due variabili aleatorie X e Y e due coefficienti reali a e b, vale $E[aX+bY] = aE[X]+bE[Y]$.

7.3 Contrazioni

Siano date una metrica d ed un fattore $c < 1$. Un operatore F viene detto **contrazione** se, applicandolo a due elementi del suo dominio, si ottengono due valori la cui distanza (rispetto a d) é inferiore al prodotto fra c e la distanza (rispetto a d) fra i due valori originari. Formalmente, si ha che F é una contrazione se vale:

$$d(F(x), F(y)) \leq c \cdot d(x, y)\ \forall x, y \in \text{Dom}(F)$$

Se un operatore é una contrazione, allora ammette al piú un solo punto fisso.

Dimostrazione. Si supponga per assurdo che l'operatore F ammetta due punti fissi, siano questi z e z'. La distanza fra i due é data da $d(z, z')$, mentre la distanza fra le rispettive applicazioni di F é data da $d(F(z), F(z'))$. Per definizione di punto fisso, si ha peró $F(z) = z$ e $F(z') = z'$; questo significa che $d(z, z') = d(F(z), F(z'))$, ovvero che la distanza fra z e z' non cambia quando F viene a questi applicata. Dato che questo viola la proprietá di contrazione, deve aversi che tale coppia di punti fissi non possa esistere.
É facile verificare che la stessa situazione si presenta se viene scelto un qualsiasi numero di punti fissi superiore a 2, pertanto occorre concludere che il numero di punti fissi di una contrazione possa essere esclusivamente 1 oppure 0.

Se una contrazione ammette un punto fisso, allora una sua applicazione ripetuta ad un qualsiasi elemento del suo dominio converge a tale punto fisso. Ovvero, dato un operatore F ed il suo punto fisso x_0, vale:

$$\lim_{n \to +\infty} F^n(x) = F(F(F(...(F(x))))) = x_0\ \forall x \in \text{Dom}(F)$$

La funzione $f(x) = x \,/\, 2$, che dimezza il valore passato in input, é una contrazione rispetto alla distanza euclidea. Infatti, dati due elementi del suo dominio x e y dove $x \leq y$:

$$d(F(x), F(y)) \leq c \cdot d(x, y) \;\Rightarrow\; d\left(\frac{x}{2}, \frac{y}{2}\right) \leq c \cdot d(x, y) \;\Rightarrow\; \frac{x}{2} - \frac{y}{2} \leq c(x - y) \;\Rightarrow\; \frac{x}{2} - \frac{y}{2} - cx + cy \leq 0 \;\Rightarrow\; \left(\frac{1}{2} - c\right)x \leq \left(\frac{1}{2} - c\right)y \;\Rightarrow\; x \leq y$$

Ha inoltre uno ed un solo punto fisso in 0. Infatti, $f(0) = 0 \,/\, 2 = 0$.