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.

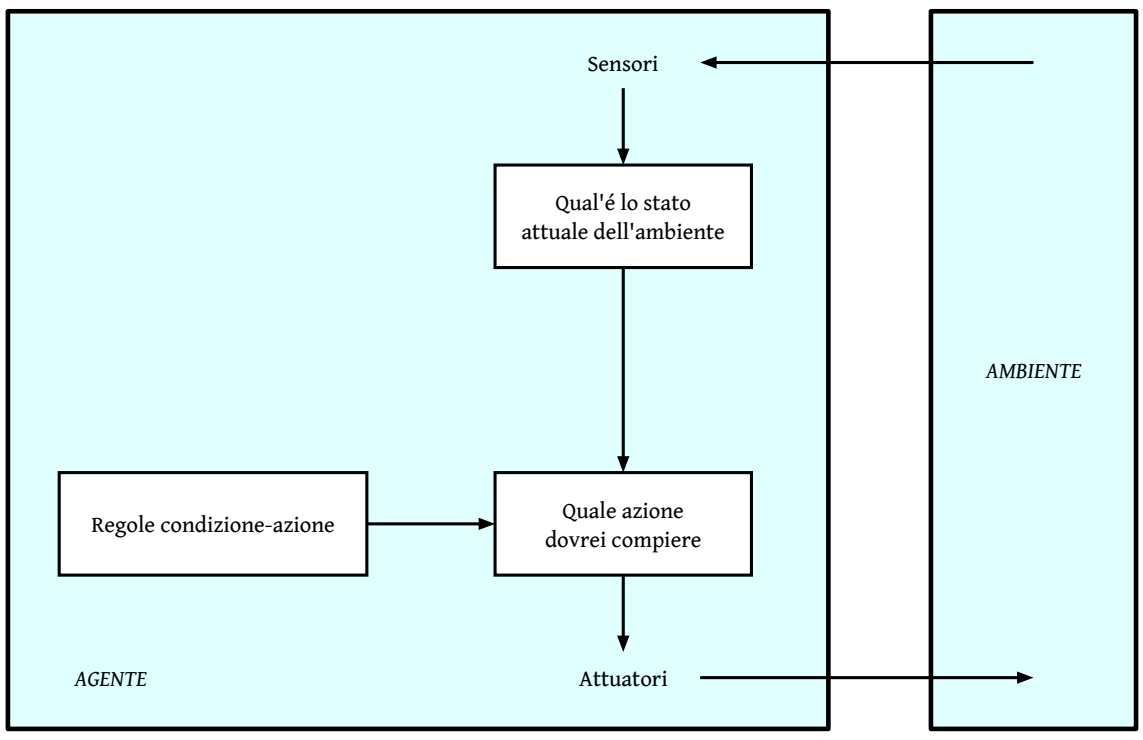
- Si consideri come ambiente il gioco degli scacchi e come agenti i giocatori umani (si assuma che le mosse non abbiano alcun limite di tempo). Tale ambiente é:
  1. Accessibile, perché ciascun giocatore ha completa conoscenza dello stato della partita;
  2. Deterministico, perché l'evoluzione degli 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
```



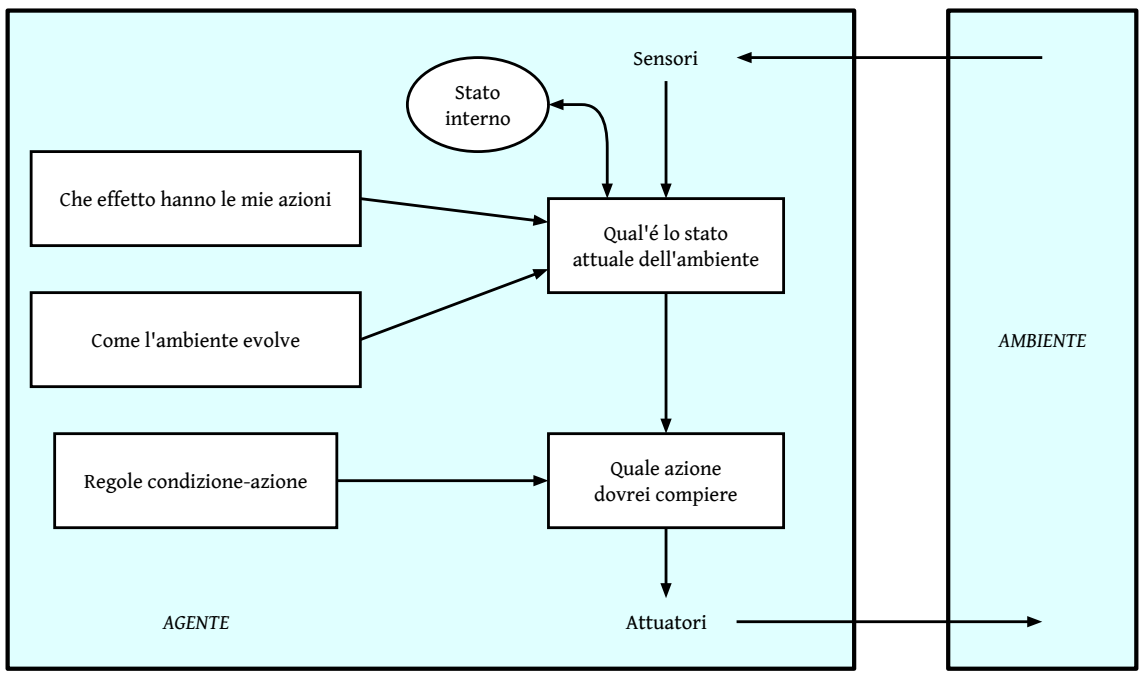
Gli agenti con riflessi semplici hanno una intelligenza limitata. Infatti, agenti di questo tipo operano correttamente solamente se l'azione da compiere che massimizza la funzione di prestazione puó essere determinata solo sulla base delle proprie percezioni, ovvero se l'ambiente é completamente accessibile. 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.

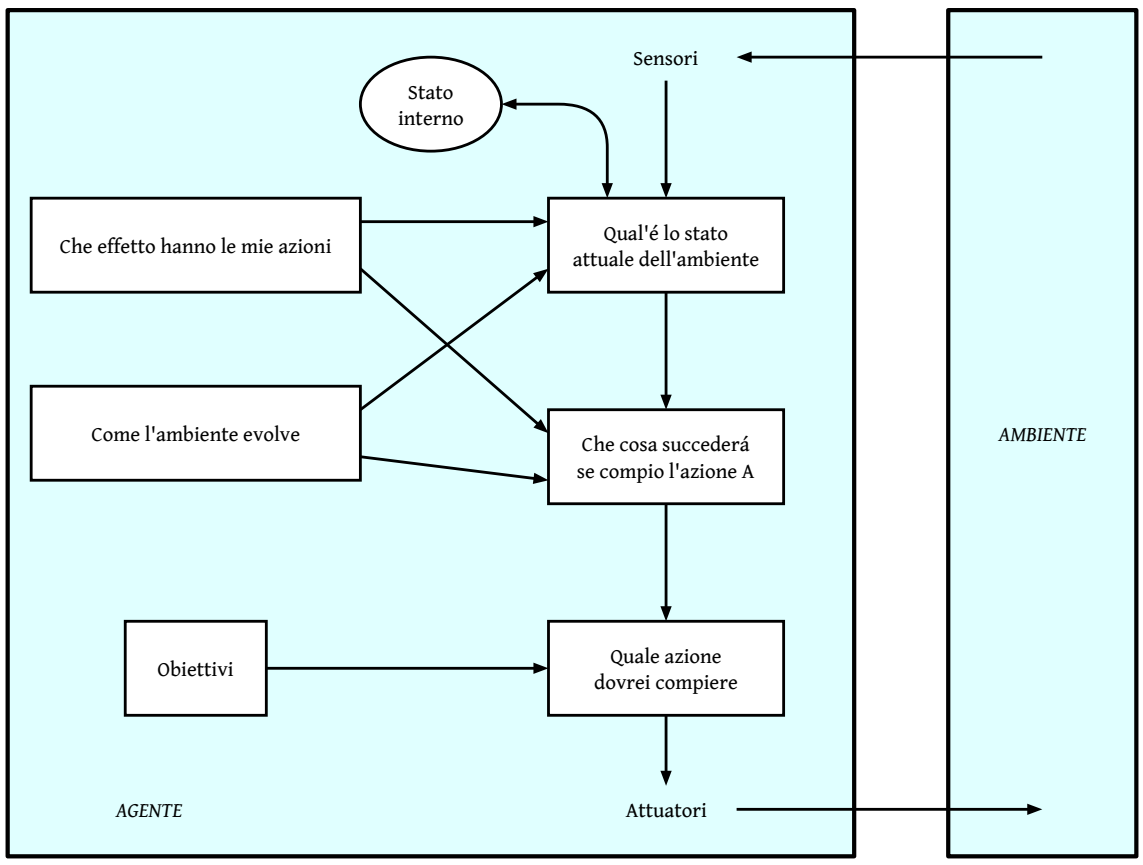
```
state <= the agent's current conception of the environment state
transition_model <= a description on how the next state depends on the current state and action
sensor_model <= a description on how the current world state is reflected in the agent's percepts
rules <= set of condition-action rules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

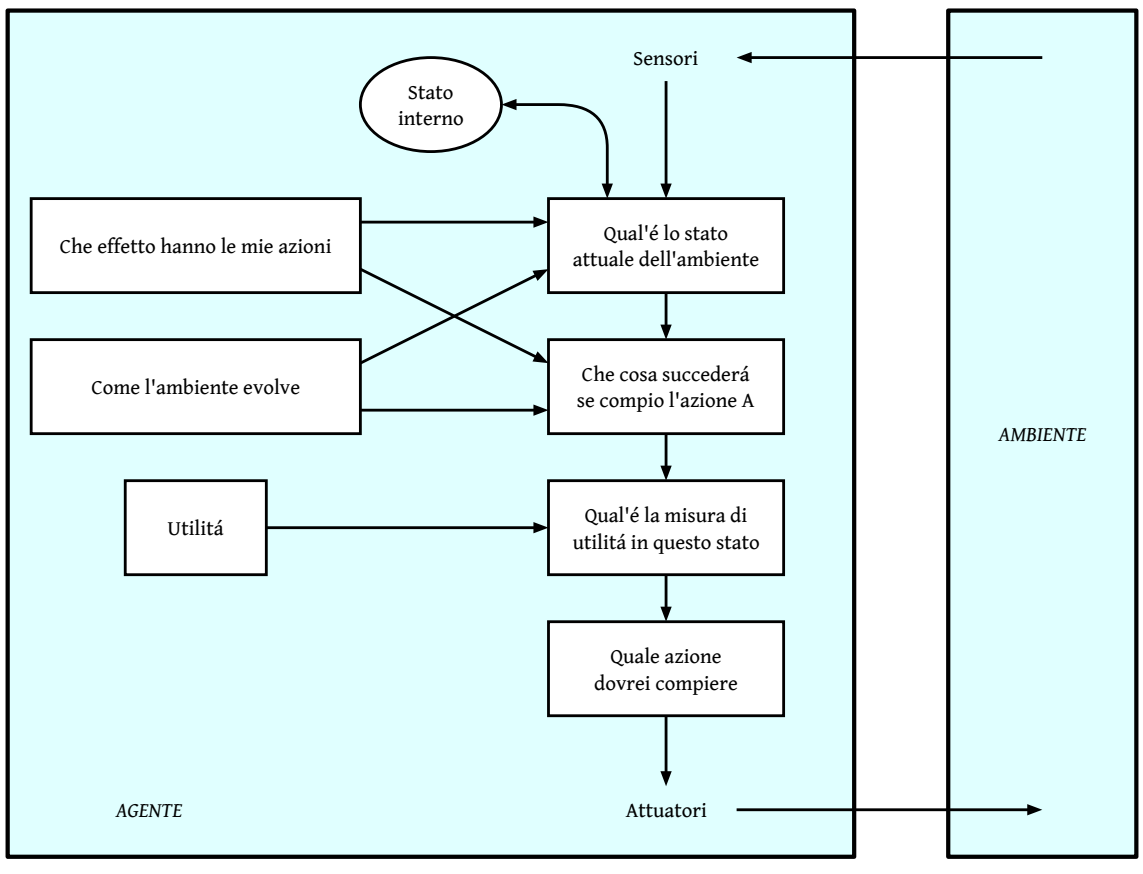
1.1.3 Agenti basati su un modello, ma basati su obiettivi

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

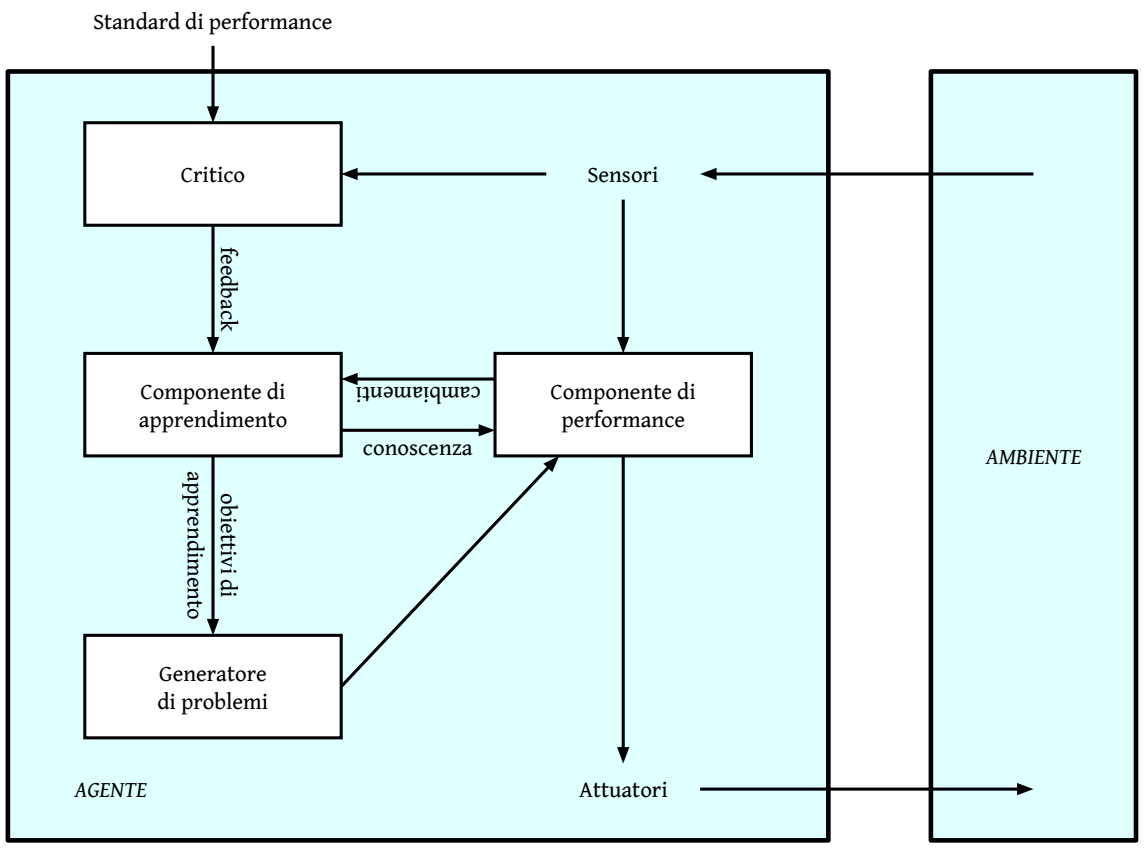
Non sempre é possibile costruire un agente razionale semplicemente spingendolo a raggiungere un obiettivo. Infatti, se tale obiettivo puó essere raggiunto tramite diverse sequenze di azioni, una potrebbe essere preferibile ad un'altra. Inoltre, un agente potrebbe dover perseguire piú obiettivi contemporaneamente 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.



Capitolo 2
Intelligenza artificiale simbolica

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.

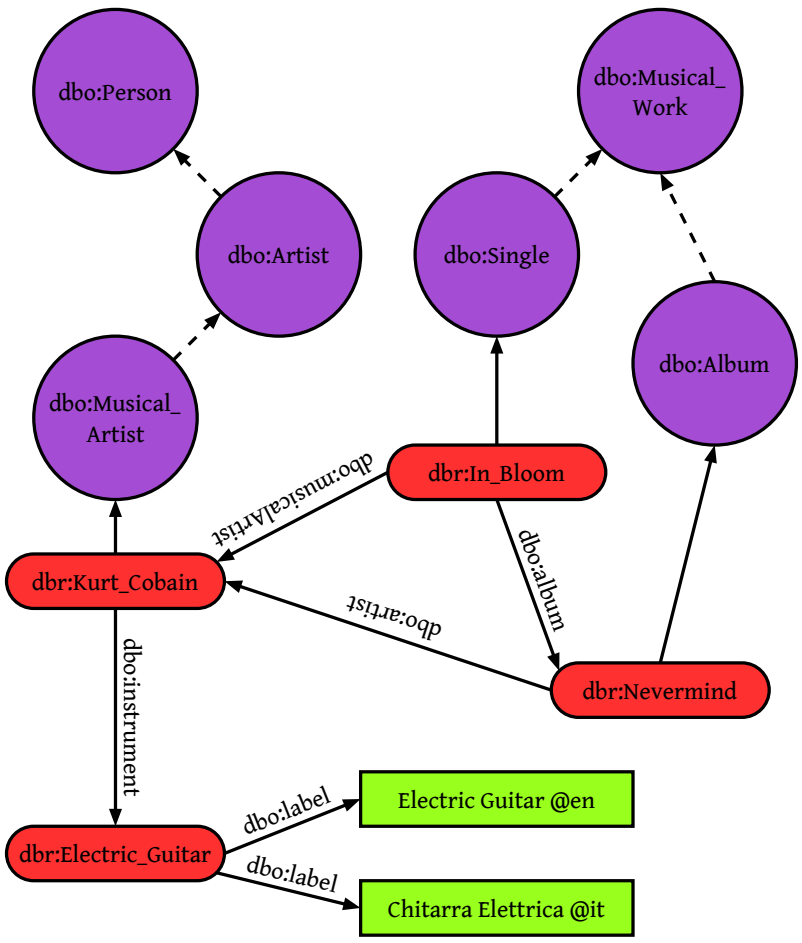
2.4.1 Triple

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 $\mathrm{Dom}(\mu)$. Dato un pattern di tripla Q, sia $Var(Q)$ l'insieme di tutte le variabili che compaiono in Q. Sia poi $\mu(Q)$ l'immagine di Q rispetto ad μ, ovvero il sottografo indotto da Q dove tutte le variabili $v \in \mathrm{Var}(Q) \cap \mathrm{Dom}(\mu)$ 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 $Q(G) = \{\mu \mid \mu(Q) \subseteq G \wedge \mathrm{Dom}(\mu) = \mathrm{Var}(Q)\}$.
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;
- $\cup$, che unisce le righe di due tabelle in un'unica tabella;
- $-$, che rimuove le righe della prima tabella che compaiono nella seconda;
- $\bowtie$, 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 $V \subseteq \mathrm{Var}(Q)$, 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 $V \subseteq \mathrm{Var}$ sia $\mu[V]$ la proiezione delle variabili V da μ, ovvero la mappatura μ' tale per cui $\mathrm{Dom}(\mu') = \mathrm{Dom}(\mu) \cap V$ e $\mu'(v) = \mu(v)$ per ogni $v \in \mathrm{Dom}(\mu')$. 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 $\mu_1(v) = \mu_2(v) perogni v \in \mathrm{Dom}(\mu_1) \cap \mathrm{Dom}(\mu_2)$, 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
Search and plan

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**;
- 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 `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`;
- Un **modello 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 `RESULT(s, a)` restituisce lo stato che viene raggiunto se viene eseguita `a` mentre ci si trova in `s`;
- Una **funzione di costo**, che associa un valore numerico a ciascuna transizione. Dati due stati `s` e `s'` ed una azione `a`, la funzione `ACTION-COST(s, a, s')` restituisce il costo che comporta il passare da `s` a `s'` applicando `a`. Tale funzione dovrebbe riflettere la misura di prestazione dell'agente.

Una sequenza di azioni forma un **percorso**; una soluzione é un percorso che ha come ultimo stato uno degli stati obiettivo. Il costo complessivo di un percorso é dato dalla somma dei costi che comporta ciascuna transizione che avviene nel percorso. Una soluzione é detta **ottimale** se ha il costo complessivo piú piccolo fra tutte le altre soluzioni. Per comoditá, si assuma che i costi siano valori positivi.
Lo spazio degli stati puó venire rappresentato sotto forma di grafo, dove i nodi del grafo corrispondono agli stati, gli archi corrispondono alle azioni che permettono di passare da uno stato all'altro ed il costo di tali azioni é l'etichetta dell'arco.
La ricerca di una soluzione per un problema di ricerca puó essere descritta sotto forma di algoritmo. Un **algoritmo di ricerca** é un algoritmo che, avendo in input un problema di ricerca, restituisce in output una soluzione per tale problema se tale soluzione esiste, oppure un errore se tale soluzione non esiste.
Un algoritmo di ricerca sovrimpone una struttura ad albero sul grafo dello spazio degli stati, formando vari percorsi a partire dallo stato iniziale, fra i quali si distingue quello che porta ad uno degli stati obiettivo. Ciascun nodo di tale albero corrisponde agli 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.

1. Nonostante questa situazione sembri irrealistica, diversi ambienti reali possono essere modellati in questo modo.

Si noti come lo spazio degli stati e l'albero di ricerca sono distinti. Lo stato degli spazi descrive il (potenzialmente infinito) insieme degli stati in cui l'ambiente puó trovarsi, e le azioni che permettono di operare le transizioni da uno stato all'altro. L'albero di ricerca descrive i percorsi che si snodano lungo questi stati che raggiungono lo stato obiettivo. Un albero di ricerca potrebbe avere piú percorsi che portano allo stesso stato, ma ogni nodo ha uno ed un solo percorso che permette di risalire da questo alla radice (come in ogni albero).
L'albero di ricerca, inizialmente costituito dal solo stato iniziale, viene costruito iterativamente. Dato un nodo dell'albero (rappresentante uno stato), é possibile **espanderlo** applicando la funzione `ACTIONS`, ottenendo quindi l'insieme di azioni che é possibile compiere se ci si trova in tale stato, ed applicando `RESULT` allo stato attuale e a ciascuna di tali azioni. Tutti i nodi cosí **generati**, detti **nodi figli** o **nodi successori**, vengono uniti al nodo attuale, detto **nodo genitore**, da un arco.
Una volta espanso un nodo, si sceglie uno dei nodi dell'albero da questo raggiungibile come nuovo nodo attuale (ovvero, si opera una transizione verso lo stato che tale nodo rappresenta) e si ripete l'operazione di espansione. L'insieme di tutti i nodi che possono essere scelti come nuovo nodo da espandere viene detto **frontiera** dell'albero di ricerca. Uno stato dello spazio degli stati si dice **raggiunto** se esiste un nodo nell'albero di ricerca a questo associato. Si noti come la frontiera separi il grafo dello spazio degli stati in due regioni: una *interna* dove ogni nodo (e lo stato ad esso legato) é stato espanso ed una *esterna* dove ogni nodo non é stato ancora raggiunto.
Ci si chiede allora quale criterio si dovrebbe adottare per scegliere quale nodo della frontiera deve diventare il nuovo nodo da espandere. Un approccio molto generico é detto **best-first search**: data una frontiera costituita dai nodi $\{a_1, ..., a_n\}$, viene scelto il nodo a_i che minimizza una certa **funzione di valutazione** $f(n)$. Si noti come possano esistere diversi nodi appartenenti alla frontiera che minimizzano tale funzione; in questo caso, la scelta di uno di questi non é rilevante.
A ciascuna iterazione dell'algoritmo, viene scelto il nodo (o uno dei nodi) che minimizza la funzione di valutazione: se lo stato che rappresenta é uno stato obiettivo, allora questo viene restituito, altrimenti vi si applica l'operazione di espansione. Ciascun nodo cosí generato viene aggiunto alla frontiera se non é stato ancora raggiunto, oppure viene riaggiunto se sta venendo raggiunto da un percorso avente costo complessivo inferiore di quello finora memorizzato per raggiungerlo. L'algoritmo restituisce un messaggio di errore se uno stato obiettivo é irraggiungibile oppure uno stato obiettivo non appena viene raggiunto. Diverse funzioni di valutazione danno origine a diversi algoritmi di ricerca.

```
function BEST-FIRST-SEARCH(initial-state, f)
    node <= /a new empty node/
    node.state <= initial-state
    frontier <= /a priority queue ordered by/ f/, with/ node /as an element/
    reached <= /a lookup table, with one entry with key/ initial-state /and value/ node
    while not IS-EMPTY(frontier) do
        node <= POP(frontier)
        if (IS-GOAL(node.state) = True) then
            return node
        ex <= EXPAND(node)
        foreach child in ex do
            s <= child.state
            if (s not in reached) or (child.path-cost < reached[s].path-cost) then
                reached[s] <= child
                ADD(frontier, child)
    return error

function EXPAND(node)
    s <= node.state
    foreach action in ACTIONS(s) do
        s' <= RESULT(s, action)
        cost <= node.path-cost + ACTION-COST(s, action, s')
        n_node <= a new empty node
        n_node.state <= s'
        n_node.parent <= node
        n_node.action <= action
        n_node.path-cost <= cost
        return n_node
```

Gli algoritmi di ricerca necessitano di una struttura dati per poter tenere traccia della struttura dell'albero di ricerca. Un nodo `node` dell'albero é rappresentato mediante una struttura dati avente quattro componenti:

- `node.state`: lo stato nello spazio degli stati a cui il nodo corrisponde;
- `node.parent`: il nodo dell'albero di ricerca che ha generato `node`;
- `node.action`: l'azione che é stata applicata a `node.parent.state` per generare `node`;
- `node.path-cost`: il costo complessivo del percorso dallo stato iniziale a `node`.

La frontiera puó venire salvata all'interno di una coda; le operazioni da eseguire su tale coda sono:

- `IS-EMPTY(frontier)` restituisce `True` se non vi sono piú nodi all'interno di `frontier` e `False` altrimenti;
- `POP(frontier)` rimuove il primo elemento di `frontier` e lo restituisce;
- `TOP(frontier)` restituisce il primo elemento di `frontier` senza rimuoverlo;
- `ADD(frontier, node)` aggiunge `node` nella posizione appropriata di `frontier`.

Gli stati raggiunti possono essere invece salvati all'interno di una hash table, dove ciascuna coppia chiave-valore é costituita da uno stato e dal nodo che questo rappresenta.
Si noti come uno stato possa essere rappresentato da piú di un nodo dell'albero di ricerca. In questo caso, si dice che tale stato é uno **stato ripetuto**; gli stati ripetuti possono dare origine a dei **cicli** nell'albero di ricerca nel momento in cui compaiono piú volte all'interno di uno stesso

percorso. Nonostante il numero di stati dello spazio degli stati sia finito, la presenza di uno o piú cicli comporta che l'albero di ricerca sia infinito, perché é sempre possibile percorrere un percorso contenente un ciclo infinite volte.
Un ciclo é un caso particolare di **percorso ridontante**, ovvero di un percorso che inizia e finisce negli stessi nodi di un altro percorso ma ha un costo maggiore, e quindi é del tutto irrilevante al fine di trovare un percorso ottimale. In genere, eliminare i percorsi ridondanti permette di portare a termine la ricerca molto piú velocemente, perché si evita di ripetere per piú volte la stessa computazione, ma l'individuare percorsi ridondanti richiede a sua volta di spendere risorse computazionali aggiuntive.
L'approccio che adotta la ricerca best-first é di tenere traccia di tutti gli stati giá raggiunti, potendo quindi individuare subito i percorsi ridondanti. Se l'unica forma di percorso ridontante che compare nel problema sono i cicli, un approccio piú conservativo prevede di risalire da un nodo a tutti i suoi nodi genitori fino a trovare un nodo che rappresenta il medesimo stato. Se invece il problema non presenta mai, o comunque quasi mai, dei percorsi ridontanti, allora l'algoritmo puó anche non implementare alcun tipo di verifica.
Un algoritmo di ricerca si dice **completo** se garantisce, per qualsiasi istanza del problema di ricerca, di trovare una soluzione, sia che questa sia uno stato raggiunto (soluzione trovata) sia che questa sia un messaggio di errore (soluzione non trovata). Se invece esiste almeno una istanza del problema per il quale questo non é in grado di fornire una risposta e si blocca indefinitamente, allora si dice che tale algoritmo é **incompleto**.
Un algoritmo di ricerca che opera su uno spazio degli stati finito é certamente completo, perché prima o poi tutti gli stati verrano raggiunti e, se nessuno di questi é uno stato obiettivo, restituisce un messaggio d'errore. Per poter essere completo, un algoritmo di ricerca che opera su uno spazio degli stati infinito deve essere **sistematico**, ovvero esplorare lo spazio in modo da poter raggiungere tutti gli stati che possono essere raggiunti a partire dallo stato iniziale.
Nell'ambito dell'informatica, si tende a misurare le prestazioni di un algoritmo che opera su un grafo in termini di $|E|$ e di $|V|$, ovvero rispettivamente la cardinalitá del suo insieme di archi e del suo insieme di vertici. Questo é l'approccio migliore nel caso in cui il grafo sia esplicitamente rappresentato da una struttura dati, mentre nel caso dei problemi di ricerca il grafo é reso implicitamente a partire dagli stati, dalle azioni e dal modello di transizione.
Per uno spazio degli stati implicito, si predilige misurare le prestazioni in termini di d, la **profonditá**, ovvero il numero di azioni in una soluzione ottimale; m, il massimo numero di azioni che possono comparire in un percorso; e b, il **fattore di branching**, ovvero il massimo numero di successori di un nodo che é necessario considerare.

3.2 Ricerca non informata

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

3.2.1 Breadth-First search

Quando tutte le azioni hanno il medesimo costo, un algoritmo di ricerca non informato che conviene utilizzare é **Breadth-First search**. In questo algoritmo, il nodo radice dell'albero di ricerca viene espanso, dopodiché ciascuno dei nodi che questo genera viene espanso, dopodiché ciascuno dei nodi che questi a loro volta generano viene espanso, ecc ... fino ad esaurire tutti i nodi. Questa strategia rispetta la proprietá di sistematicitá, e pertanto garantisce che l'algoritmo sia completo anche se lo spazio degli stati é infinito.
L'algoritmo `BREADTH-FIRST-SEARCH` puó essere implementato, in prima analisi, come una versione di `BEST-FIRST-SEARCH` in cui la funzione di valutazione $f(n)$ é la profonditá di n, ovvero il numero di azioni che é stato necessario per arrivare dalla radice a n. 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.
É possibile ottimizzare ulteriormente l'algoritmo osservando come non sia necessario tenere traccia dei percorsi precedentemente usati per raggiungere un certo nodo, perché per come l'algoritmo é strutturato, una volta che un nodo viene raggiunto non é possibile trovare un percorso migliore per raggiungerlo. Pertanto, non é necessario implementare `reached` come una hash table, ma é sufficiente che sia un insieme non ordinato. Inoltre, per lo stesso motivo, é possibile valutare se un nodo é un nodo obiettivo prima di controllare se tale nodo é giá stato raggiunto.

```
function BREADTH-FIRST-SEARCH(problem)
    node <= /a new empty node/
    node.state <= initial-state
    frontier <= /a FIFO queue, with/ node /as an element/
    reached <= {initial-state}
    while not IS-EMPTY(frontier) do
        node <= POP(frontier)
        ex <= EXPAND(node)
        foreach child in ex do
            s <= child.state
if (IS-GOAL(s) = True) then
                return child
            if (s not in reached) then
                reached <= reached u {s}
                ADD(frontier, child)
 return error
```

Breadth-first search, quando trova una soluzione, tale soluzione sará necessariamente trovata con il minor numero di passi possibili. Questo perché quando viene generato un nodo alla profonditá d, tutti i nodi a profonditá $d-1$, $d-2$, ecc ... sono giá stati generati (ed espansi). Pertanto, se uno di questi nodi avesse contenuto uno stato obiettivo, sarebbe giá stato trovato. Pertanto, se il costo di tutte le azioni del problema é lo stesso, Breadth-first search é necessariamente l'algoritmo piú efficiente possibile dal punto di vista del costo. Inoltre, é un algorimo completo, perché prima o poi tutti i nodi verranno raggiunti.
Per quanto riguarda la complessitá di breadth-first search, si osservi come a partire dal nodo iniziale vengano allora generati al piú b nodi, ed a sua volta a partire da ciascuno di questi vengono generati al piú b nodi, ecc ... Questo significa che ad una certa profonditá d, vengono generati

al piú b^d nodi. L'algoritmo prevede che tutti questi nodi debbano potenzialmente venire generati ed esplorati, a meno di trovare una soluzione prima di averli esauriti, pertanto la complessitá in termini sia di tempo che di spazio della ricerca breadth-first é $O(b^d)$. Questo significa che, per quanto breadth-first search sia completo, é molto inefficiente, a meno di operare su istanze molto piccole.

3.2.2 Depth-First search

Un algoritmo piú efficiente in termini di complessitá rispetto a Breadth-First search é **Depth-First search**. A differenza del precedente, che prevede di espandere i nodi della frontiera uno alla volta, Depth-First search prevede di espandere sempre il nodo piú profondo. Questo potrebbe essere implementato a partire da `Best-First search` scegliendo come funzione di valutazione il negativo della profonditá.
In realtá, Depth-First search viene in genere implementato non come una ricerca su grafo, ma come una ricerca ad albero che non tiene traccia degli stati raggiunti. Depth-First search procede immediatamente fino alla massima profonditá raggiungibile a partire dal nodo iniziale. Dopodiché, l'algoritmo opera un **backtracking**, ovvero "ritorna" al primo nodo lungo il percorso che non é stato ancora interamente espanso. Depth-First search restituisce comunque la prima soluzione che viene trovata, ma non é detto che questa sia la soluzione ottimale, perché potrebbe esserci una soluzione migliore a partire da uno dei nodi inesplorati.
Se lo spazio degli stati é finito, Depth-First 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 infinito, ma é possibile ottimizzare l'algoritmo per individuare tali cicli ed evitarli. Se lo spazio degli stati é infinito, l'algoritmo non é né sistematico né completo, perché l'algoritmo potrebbe rimanere bloccato nell'espandere lo stesso percorso indefinitamente, anche se non sono presenti cicli.
Per impedire che l'algoritmo si blocchi in una discesa infinita, é possibile impostare una profonditá massima, oltre la quale all'algoritmo viene impedito di procedere, obbligandolo a fare backtracking anche se il nodo in esame potrebbe venire espanso. In questo modo, si garantisce che l'algoritmo non possa bloccarsi indefinitamente, ma potrebbe potenzialmente perdere parte delle soluzioni se queste esistono ma molto in profonditá.
Sebbene Depth-First search possa potenzialmente non essere completo, nella pratica viene comunque spesso preferito a Breadth-First search. Questo perché la complessitá in tempo e spazio di Depth-First search ha un bound infinitamente inferiore: se l'albero é finito, la complessitá in tempo dell'algoritmo é proporzionale al numero degli stati, mentre la complessitá in spazio é $O(bm)$, dove b é il fattore di branching e m é la massima profonditá dell'albero.

3.3 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 in ogni direzione ("a caso"), gli algoritmi di ricerca informati possono orientare la loro computazione verso una determinata direzione.
Viene detta **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 in n e ha fine nello stato obiettivo piú vicino. Tale funzione é in genere specifica per ogni possibile istanza del problema in esame. Si noti come l'informazione restituita dalla euristica non suggerisca necessariamente di intraprendere l'azione che risulta in un percorso efficiente.

3.3.1 Greedy search

L'algoritmo **Greedy search** é un algoritmo di ricerca informato che sceglie sempre il nodo che ha il minor valore di $h(n)$ fra tutti i nodi raggiungibili, assumendo che sia anche uno dei nodi che costituiscono il percorso piú efficiente. Puó essere quindi implementato a partire da Best-First search scegliendo $h(n)$ come $f(n)$.
La performance di Greedy search dipende molto da quanto l'algoritmo é in grado di fare una buona predizione sulla base dell'euristica. Se l'euristica porta quasi sempre ad un percorso favorevole, la complessitá in termini di tempo e spazio puó scendere fino a $O(bm)$. Se l'euristica porta quasi sempre ad un percorso sfavorevole, di fatto Greedy search opera in maniera quasi indistinguibile da Depth-first search, perché vengono esplorati molti (se non tutti) nodi in profonditá, e la complessitá diviene $O(|V|)$. Per lo stesso motivo, Greedy search é completo se lo spazio degli stati é finito, mentre é incompleto se lo spazio degli stati é infinito.

3.3.2 A* search

Il piú comune algoritmo di ricerca informato é **A* search** (pronuncia: "A-star search"), implementabile a partire da Best-first search usando come funzione di valutazione $f(n) = g(n) + h(n)$, dove $g(n)$ é il costo totale del percorso che va dal nodo radice a n. Di fatto, A* search é una combinazione di Uniformed Cost search e di Greedy Search.
A* search é un algoritmo completo; se A* search sia anche ottimale dipende dalle caratteristiche della funzione di euristica. In particolare, una euristica si dice **ammissibile** se approssima sempre i costi per difetto.

Fintanto che come euristica di A* search viene scelta una euristica ammissibile, A* search é ottimale.

Dimostrazione. Si supponga per assurdo che A* search possa restituire un percorso non ottimale anche se viene scelta una euristica ammissibile. Sia C^* l'effettivo costo del percorso ottimale di una qualche applicazione di A* search dallo stato radice ad un certo stato obiettivo e sia C il costo stimato dalla funzione di valutazione per il medesimo percorso.

Per quanto assunto nell'ipotesi di assurdo, deve aversi $C > C^*$. Deve allora esistere un certo nodo n sul percorso ottimale ma che non é stato espanso: questo perché se tutti i nodi sul percorso ottimale fossero stati espansi, allora la funzione di valutazione avrebbe restituito C^* e non C. Siano $g^*(n)$ e $h^*(n)$ rispettivamente l'effettivo costo del sottopercorso ottimale che va dallo stato di partenza a n e l'effettivo costo del sottopercorso ottimale che va da n al piú vicino stato obiettivo.

Si ha quindi $C^* = g^*(n) + h^*(n)$. Inoltre, avendo assunto che n si trovi su un sottopercorso ottimale, deve aversi che $g(n)$ e $g^*(n)$ coincidono. É quindi possibile scrivere $f(n) = g(n) + h(n) = g^*(n) + h(n)$. Avendo assunto che l'euristica é ammissibile, deve aversi $h(n) \leq h^*(n)$, e pertanto $C = f(n) \leq g^*(n) + h^*(n)$. Questo é peró in contraddizione con l'ipotesi di assurdo, pertanto occorre assumere che A* search restituisca sempre una soluzione ottimale quando viene scelta una euristica ammissibile.

Una euristica $h(n)$ si dice **consistente** se, per ogni nodo n e per ogni successore n' di n generato dall'azione a, vale [2]:

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

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.
Ci si chiede allora come si possa costruire una euristica per un dato problema. Se a partire da un problema se ne costruisce una versione "semplificata" rimuovendo le restrizioni imposte all'agente, ovvero aumentando il numero di azioni a questo disponibili, si dice che si ottiene un **problema rilassato**.
Il grafo dello spazio degli stati del problema rilassato é un supergrafo del grafo dello spazio degli stati del problema originale, perché aumentare il numero di azioni disponibili all'agente comporta l'aggiunta di nuovi archi al grafo. Per questo motivo, ogni soluzione ottimale del problema originale é anche una soluzione per il problema rilassato, ma il problema rilassato potrebbe avere soluzioni di costo ancora inferiore che nel problema originale non sono presenti, perché l'aggiunta di nuove azioni potrebbe condurre a delle scorciatoie. Quindi, il costo di una soluzione ottimale di un problema rilassato fornisce un limite inferiore al costo delle soluzioni del problema originale, e puó essere quindi usata come euristica per il problema originale.
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.

3.4 Classical planning

Il **Planning Classico** consiste nel trovare una sequenza di azioni che permettono di raggiungere un determinato obiettivo in un ambiente discreto, deterministico, statico e accessibile. A differenza dei problemi di ricerca, che richiedono una euristica ad-hoc per ciascun dominio, il linguaggio del planning é indipendente dal dominio del problema in esame.
Similmente ai problemi di ricerca, un **problema di planning** é definito a partire 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 $a_0, \ldots, a_n$ che mappa s_0 su S_G. In altri termini, esiste una sequenza di stati $s_0 \ldots, s_{n+1}$ 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 $i = 0, \ldots, n$.
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.

2. É facile verificare che questa é una forma di disuguaglianza triangolare.

Il problema viene codificato in un linguaggio indipendente dal dominio. Questi linguaggi permettono di descrivere azioni, sensori, obiettivi e situazione iniziale mediante delle rappresentazioni schematiche che non necessitano di alcuna conoscenza specifica sul dominio del problema.
Una volta definito il linguaggio, 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. In questo modo, é possibile focalizzare l'attenzione sulla sola conoscenza, tralasciando i dettagli implementativi dell'agente, come ad esempio quale algoritmo usa per formulare le deduzioni. In questo modo, ogni inferenza puó essere potenzialemente calcolata dall'agente, fintanto che é possibile formularla nel linguaggio formale di riferimento. L'approccio opposto é quello **imperativo**, dove l'agente viene istruito nel dettaglio su quali passi compiere per ciascuno stato in cui l'agente si trova [3].
Un linguaggio molto semplice appartenente a questa famiglia é **STRIPS** (**Stanford Research Institute Problem Solver**). Un problema codificato nel linguaggio STRIPS é una quadrupla $P = (F, O, I, G)$:

- Un insieme F di **condizioni** (variabili proposizionali, istanziate);
- Un insieme O di **operatori** (azioni). Ogni operatore a é a sua volta una tripla $\text{Prec}(a) = \alpha, \text{Add}(a) = \beta, \text{Del}(a) = \gamma$. α é un insieme di **precondizioni**, ovvero di condizioni che devono essere vere affinché sia possibile applicare l'operatore. β é un insieme di condizioni che vengono rese vere dall'azione (vengono aggiunte allo stato corrente). γ é un insieme di condizioni che vengono rese false dall'azione (vengono rimosse dallo stato corrente);
- Uno **stato iniziale** $I \subseteq F$, costituito da tutte le condizioni che sono inizialmente vere (vale la closed-world assumpion; tutto ció che non é inizialmente vero é assunto falso);
- Una specifica dello **stato obiettivo**, riportato come una coppia $\langle N, M \rangle$ la quale riporta, rispettivamente, quali condizioni devono essere vere e false affinché uno stato possa essere considerato uno stato obiettivo.

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 $\text{Prec}(a) \subseteq s$;
- Lo stato successivo s^j é dato da $s - \text{Del}(a) + \text{Add}(a)$;
- 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:

$$F = \{p, q, r\} \quad I = \begin{Bmatrix} p \\ \end{Bmatrix} \begin{matrix} \text{Prec}(a) = \{p\}, \text{Add}(a) = \{q\}, \text{Del}(a) = \{\} \\ \text{Prec}(b) = \{q\}, \text{Add}(b) = \{r\}, \text{Del}(b) = \{q\} \end{matrix} \quad G = \{q, r\}$$

- 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.

STRIPS non permette di usare variabili, perché tutti i componenti devono essere nominati esplicitamente. Questo rende STRIPS molto semplice, ma al contempo molto prolisso (per quanto non prolisso quanto riportare tutti gli stati esplicitamente).
Una estensione di STRIPS che permette l'uso di variabili é **Planning Domain Definition Language** (**PDDL**) [4]. Un problema in PDDL é formato da

3. I nomi *dichiarativo* e *imperativo* sono in analogia con gli omonomimi paradigmi di programmazione.
4. La sintassi di PDDL é simile a quella di Lisp.

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` .
I problemi di planning possono essere risolti come problemi di ricerca euristica [5]. I problemi di ricerca euristica sono problemi NP-Completi, per

5. 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.

quanto comunque risolvibili in tempo accettabile anche per grandi istanze.
COME RISOLVO UN PROBLEMA DI PLANNING HELLO???
Come giá anticipato, non tutti i problemi non possono essere formulati in un linguaggio di planning. Altri problemi sono invece intrinsecamente complessi e, sebbene sia possibile formularli in PDDL o STRIPS, 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.