Programmazione ASP

- Introduzione alla programmazione logica

La logica (dal greco *logos*, ovvero “parola”, “pensiero”, “idea”) è lo studio del ragionamento, rivolto in particolare a definire la correttezza dei procedimenti inferenziali del pensiero.

La logica proposizionale è una logica a due valori (vero e falso) basata su un insieme di proposizioni e su un insieme di connettivi attraverso i quali è possibile combinare le proposizioni per creare formule arbitrariamente complesse. Per proposizione intendiamo un’affermazione relativa ad un singolo fatto di cui può essere stabilita la verità. Il valore di verità di una formula ben formata può quindi variare a seconda di quanto stabilito dallo specifico assegnamento di verità utilizzato per le proposizioni presenti nella formula stessa.

I valori di verità sono assimilabili a costanti logiche, le proposizioni sono assimilabili a variabili logiche,i connettivi sono assimilabili ad operatori logici e le formule sono assimilabili ad espressioni logiche.

Gli elementi sintattici di base della logica proposizionale sono i seguenti:
- Un insieme di proposizioni.
- I seguenti connettivi logici unari:
  - Negazione: ¬.
- I seguenti connettivi logici binari:
  - Disgiunzione: ∨.
  - Congiunzione: ∧.
  - Implicazione: →.
  - Doppia implicazione: ↔.

La logica proposizionale non è sufficientemente espressiva per rappresentare ragionamenti relativi ad una moltitudine di oggetti, come “tutti gli oggetti godono di una certa proprietà” oppure “esiste almeno un oggetto che gode di una certa proprietà”. La logica proposizionale si concentra infatti sui connettivi logici e su come il valore di verità di una formula ben formata dipenda dal valore di verità delle sue sotto-formule immediate, senza considerare l’eventuale struttura interna delle proposizioni presenti nella formula. La logica dei predicati ovvia a questi inconvenienti includendo degli ulteriori operatori, detti quantificatori, in aggiunta ai connettivi logici della logica proposizionale e sostituendo le proposizioni con predicati (o atomi) applicati a termini. I termini, che individuano gli oggetti di interesse, sono espressi tramite costanti, variabili e funzioni definite sui termini. I predicati, che specificano le proprietà di interesse per gli oggetti precedentemente individuati, sono espressi tramite relazioni su insiemi di termini. I predicati sono assimilabili a funzioni logiche.

La logica dei predicati del primo ordine ha un quantificatore universale denotato con ∀ (“per ogni”) ed un quantificatore esistenziale denotato con ∃ (“esiste”), i quali possono fare riferimento solo a termini variabili che compaiono come argomenti di funzioni e predicati.

La programmazione logica nacque all’inizio degli anni ‘70 grazie soprattutto alle ricerche di due studiosi : Kowalski ne elaborò i fondamenti teorici. In particolare a lui si deve la scoperta della doppia interpretazione, procedurale e dichiarativa, delle clausole di Horn; e Colmeraurer, fu il primo a progettare e implementare un interprete per un linguaggio logico: il Prolog.

In programmazione logica un problema viene descritto con un insieme di formule della logica, dunque in forma dichiarativa. La programmazione classica, procedurale, è più adatta a problemi

quotidiani, ben definiti. Essa adotta un paradigma imperativo: richiede cioè che siano specificate delle rigorose sequenze di passi (algoritmi) che, a partire dai dati a disposizione, portino ad ottenere i risultati desiderati. Nella programmazione logica i dati e il controllo sono ben distinti, bisogna preoccuparsi di specificare solo la componente logica mentre il controllo spetta al sistema. Ciò significa che bisogna solo definire il problema senza comunicare alla macchina come risolverlo. Nella programmazione logica si deve abbandonare il modo di pensare orientato al processo. Il programma è una descrizione della soluzione, non del processo, e si costruisce descrivendo in un linguaggio formale l'area applicativa, ossia gli oggetti che in essa esistono, le relazioni fra loro e i fatti che li riguardano.

Il linguaggio formale alla base della programmazione logica è rappresentato dalle clausole di Horn. Un letterale è una formula atomica e diciamo che un letterale è positivo oppure negativo a seconda che sia della forma p oppure ¬p, con p una proposizione. Una clausola è una disgiunzione di letterali. Chiamiamo clausola di Horn una clausola che contiene al più un letterale positivo, la quale viene classificata come:

– Un fatto se è della forma p, che è equivalente a $I \rightarrow p$.

– Un vincolo se è della forma $\bigvee_{i=1}^{n} \neg p_i$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow O$.

– Una regola se è della forma $p \vee (\bigvee_{i=1}^{n} \neg p_i)$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow p$.

- Prolog

Nato nel 1972 grazie alle ricerche di Colmerauer e Kowalski, il Prolog (acronimo per PROgramming in LOGic) è uno dei primi linguaggi di programmazione logica. E' molto potente e flessibile, ed è adatto particolarmente all'Intelligenza Artificiale. Il programmatore pone la sua attenzione sugli oggetti e sulle relazioni che li legano, esprimendo la conoscenza ad essi relativa sotto forma di fatti e di regole, per poi poter porre domande. L'interprete Prolog tenta poi di rispondere alle domande ponendole in relazione con i fatti e le regole della base di conoscenza.

Dunque, le componenti fondamentali del Prolog sono:
- dichiarazioni di fatti sugli oggetti e le loro relazioni
- dichiarazioni di regole sugli oggetti e le loro relazioni
- domande sugli oggetti e le loro relazioni.

Tramite esse il programmatore interagisce col linguaggio: con le prime due scrive i programmi, con l'ultima li esegue. Programmare in Prolog equivale a descrivere il dominio del problema tramite fatti e regole sotto forma di clausole di Horn.

Un predicato (o atomo) ha forma p(t1, .. , tn) dove p è il nome del predicato, t1, .. , tn sono i termini e n (>= 0) è l'arietà del predicato. I fatti Prolog sono predicati rappresentanti clausole di Horn non condizionali. Essi, cioè, esprimono un'affermazione che non è vincolata alla preventiva verifica di un insieme di condizioni.

Esempio.
persona(marco). → marco è una persona

Una relazione è, in genere, definita come l'insieme di tutte le sue istanze. Il Prolog tratta ogni relazione come una pura entità sintattica. Ciò vuol dire che il significato di una relazione è dato dal

programmatore (intendendo con ciò l'ordine degli argomenti, il modo in cui vengono messi in relazione, il nome), ed è a suo carico usarla sempre in modo coerente con tale significato.

Esempio.
genitore(stefano, marco) → stefano è genitore di marco

Le regole Prolog sono clausole di Horn condizionali, cioè esprimono un'affermazione che è vincolata alla preventiva verifica di un insieme di condizioni. Un fatto è una cosa sempre, incondizionatamente, vera; una regola specifica una cosa vera a patto che la condizione sia soddisfatta. L'uso dei quantificatori (∀, ∃) viene espresso dall'uso delle variabili nelle regole. Le regole indicano situazioni di carattere generale servendosi di oggetti e relazioni tra essi. Una regola è costituita da una parte condizione (corpo, a destra) e una parte conclusione (testa, a sinistra), separate dal simbolo :- ("Se", che rappresenta l'implicazione). Una regola si dice safe se ogni variabile che appare in testa, in un letterale negativo o in un'operazione di comparazione (<, >, <=, …) appare anche in un letterale positivo. In un programma logico sono ammesse solo regole safe. In un contesto guidato dagli obiettivi come quello del Prolog, ogni regola di un programma si presta ad una duplice interpretazione. Il significato dichiarativo della regola è che la formula di testa è vera se la formula del corpo è vera. Il significato procedurale della regola è che per raggiungere l'obiettivo della formula di testa bisogna prima raggiungere l' obiettivo della formula del corpo. Si distinguono perciò due tipi di predicati: i predicati EDB appaiono solo nel corpo delle regole o nei fatti, mentre i predicati IDB possono apparire anche nella testa di qualche regola. Un programma Prolog parte con i fatti rappresentanti predicati EDB e iterativamente deriva i predicati IDB attraverso le regole. L'insieme dei fatti e delle regole costituisce la base di conoscenza. Una regola si definisce ricorsiva se tra i predicati nel corpo è presente il predicato in testa. La ricorsione è tipicamente usata nella programmazione logica per ottenere la chiusura transitiva di una relazione. La chiusura transitiva di una relazione R è un'altra relazione che aggiunge ad R tutti quegli elementi che, pur non essendo in relazione direttamente fra loro, possono essere raggiunti da una "catena" di elementi tra loro in relazione.

Esempio:
antenato(X,Y) :- genitore(X,Y). → se X è genitore di Y allora X è un antenato di Y
antenato(X,Y) :- genitore(X,Z), antenato(Z,Y). → chiusura transitiva di genitore

Il primo passo, nella programmazione Prolog, consiste nella creazione della base di conoscenza tramite un apposito ambiente di editing. Successivamente sarà possibile interrogarla in quello che viene chiamato ambiente di query. Nell'ambiente di query il Prolog avverte che è pronto a rispondere a delle eventuali domande riguardanti la base di conoscenza in suo possesso. Tali domande, in Prolog, vanno sotto il nome di goal (ossia obiettivi da dimostrare) o query (ossia interrogazioni da soddisfare).

Per risolvere problemi definiti attraverso clausole di Horn, il Prolog impiega il metodo di risoluzione di Robinson basato sulla strategia SLD. L'esecuzione di un programma Prolog viene attivata da un obiettivo. L'obiettivo è una clausola di Horn data da un vincolo ed è descritto sintatticamente come una regola senza testa in cui il simbolo :- è sostituito dal simbolo ?- .

Esempio.
?- antenato(francesco, marco). → francesco è un antenato di marco?

Il raggiungimento dell'obiettivo viene perseguito applicando il metodo di risoluzione basato sulla strategia SLD ai fatti e alle regole del programma e all'obiettivo dato. Dati un programma Prolog (insieme di clausole C) ed un obiettivo (clausola iniziale c1), bisogna trovare per tutte le formule atomiche dell'obiettivo considerate, dei fatti che siano unificabili con quelle formule o delle regole

le cui teste siano unificabili con quelle formule. Nel caso la risoluzione avvenga con una regola, bisogna poi procedere nello stesso modo con le formule atomiche del corpo della regola considerate. Poiché ogni formula atomica presente in un obiettivo potrebbe essere unificabile con più fatti e teste di regole di un programma, in Prolog la risoluzione della formule atomiche vengono esaminate rispetto a tutte le clausole del programma considerate nell'ordine in cui sono scritte. Perciò ,per motivi di efficienza, nei programmi conviene elencare i fatti prima delle regole. Posta una domanda, il Prolog risponde Yes o No per confermarne o meno la validità all'interno della base di conoscenza e, nel caso in cui la domanda contenga una variabile, Prolog risponde anche, se la risposta è affermativa, per quali oggetti la domanda è soddisfatta.

L'esecuzione di un programma Prolog su un obiettivo può essere rappresentata graficamente tramite un albero e-o, così chiamato perché ogni nodo rappresenta un obiettivo (congiunzione) e ha tanti nodi figli quanti sono i fatti e le regole alternativi tra loro con i quali può avvenire la risoluzione (disgiunzione). Durante l'applicazione della risoluzione, l'albero viene costruito in profondità per via dell'ordine LIFO imposto dalla strategia SLD.

La radice dell'albero e-o contiene l'obiettivo iniziale, mentre ogni foglia può contenere # oppure un obiettivo alla cui formula atomica più a sinistra non è applicabile la risoluzione. Ogni ramo dell'albero è etichettato con i legami creati per unificazione durante la corrispondente applicazione della regola di inferenza per risoluzione. Ogni cammino dalla radice ad una foglia contenente # è un cammino di successo, mentre tutti gli altri cammini sono cammini di fallimento o cammini infiniti.

Consideriamo il seguente programma Prolog:

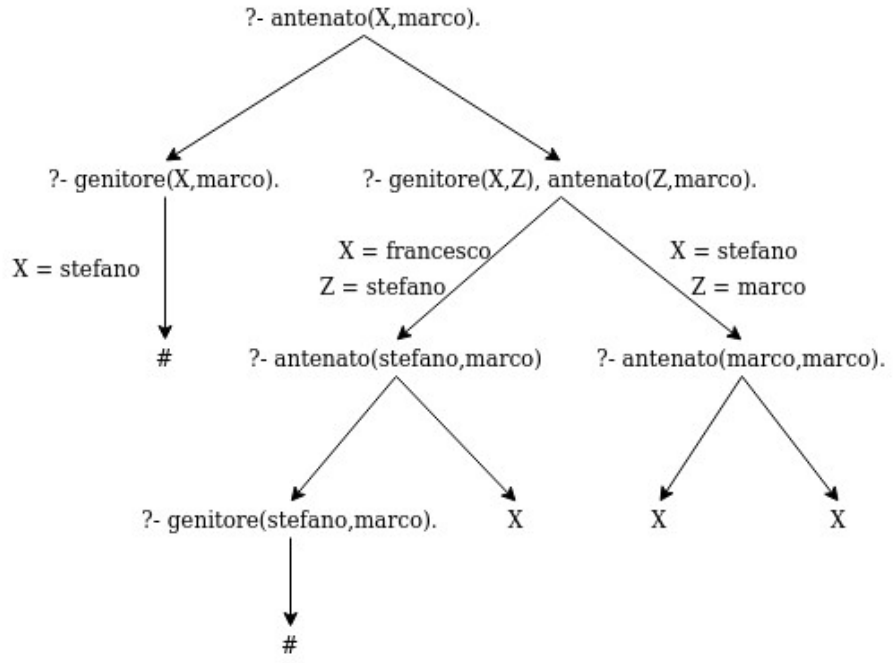
```
genitore(francesco, stefano).
genitore(stefano, marco).
antenato(X, Y) :- genitore(X, Y).
antenato(X, Y) :- genitore(X, Z), antenato(Z, Y).
```

Consideriamo poi l'obiettivo:
?- antenato(X, marco).

Osserviamo che l'obiettivo contiene una variabile, dunque l'esecuzione del programma consiste nel trovare tutti gli antenati della persona specificata, sulla base delle informazioni fornite dal programma. La risposta del Prolog sarà:

```
antenato(stefano,marco).
antenato(francesco,marco).
```

e l'albero e-o risultante è il seguente:



- Programmazione logica disgiuntiva

Un termine (un predicato, una regola o un programma) è detto ground se non contiene nessuna variabile. Dato un programma logico P, chiamiamo Universo di Herbrand (Up) l'insieme di tutte le costanti in P. Chiamiamo Base di Herbrand l'insieme di tutti i possibili predicati ground costruibili dai predicati in P con le costanti nell'Universo di Herbrand (considerando anche la negazione).

Un'interpretazione I è un insieme di atomi ground. Un letterale positivo è vero rispetto ad I se appartiene ad I, falso altrimenti; mentre un letterale negativo è vero rispetto ad I se non appartiene ad I, falso altrimenti.

Un'interpretazione I è un modello per P se per ogni regola in P, se il corpo è vero (rispetto a P) allora lo è anche la testa.

Esempio.
a :- b,c.
c :- d.
d.

I1 = {b,c,d}, I2 = {a,b,c,d} I3 = {c,d}

I2 e I3 sono modelli, mentre I1 non lo è perché il corpo della prima regola è vero rispetto a I1 (b e c appartengono a I1) mentre la testa è falsa (a non appartiene a I1).

L'intersezione di tutti i possibili modelli è detto modello minimale. Un programma positivo (ovvero che non contiene negazione in nessuna regola) ha un unico modello minimale, mentre un programma con negazione potrebbe averne diversi.

Dato un programma P e un interpretazione I, si definisce conseguenza immediata di I l'insieme di tutti gli atomi che compaiono in testa a qualche regola in P di cui il corpo è vero rispetto ad I. Nella

prima iterazione si analizzano i corpi delle regole con i predicati nell'insieme di partenza (ovvero i fatti). Se un corpo risulta vero, si deriva la testa. L'insieme di tutte le teste così derivate si unisce poi all'insieme di partenza e verrà usato nell'iterazione successiva. Il processo termina quando non vengono più generati nuovi predicati. Questo processo prende il nome di Tp ed è usato per l'individuazione dei modelli minimali di un programma logico.

Esempio.

a :- b.
c :- d.
e :- a.

Tp(0) = {b} → interpretazione di partenza (fatti)

Tp(1) = {a}. Tp(2) = {e}. → modello minimale = { b, a, e }

In un programma ground e positivo si ha la certezza che il Tp raggiunga un punto fisso e che l'insieme dei predicati IDB sia finito. In un programma non ground il Tp inizia con i predicati EDB e applicando iterativamente questo procedimento deriva i predicati IDB.

La negazione può causare problemi se unita alla ricorsione. La stratificazione è un vincolo che i programmi devono rispettare per escludere la negazione all'interno della ricorsione. Per chiarire la stratificazione facciamo uso del grafo delle dipendenze. In questo grafo i nodi sono i predicati IDB e la presenza di un arco da un nodo a un nodo b è data dalla presenza di a nel corpo di una regola in cui b è in testa. Un arco a → b viene marcato (con il simbolo -) se a è un letterale negativo. Un programma si dice stratificato se il grafo delle dipendenze non contiene cicli in cui appare un arco marcato.

Un' interpretazione I si dice essere un Answer Set per un programma positivo P se è un modello minimale per P. Per i programmi non positivi bisogna considerare il programma ridotto. Dato un problema P e un'interpretazione I, P ridotto si ottiene cancellando tutte le regole con un letterale negativo falso rispetto ad I e cancellando tutti i letterali negativi nelle rimanenti regole. Un Answer Set di un programma P è un'interpretazione I tale che essa sia un Answer Set per P ridotto. Gli Answer Set sono anche chiamati modelli stabili.

Esempio.

P = {a :- d, not b. b :- not d. d.}
I = {a, d}
P ridotto = {a :- d. d.}

Nella prima regola viene rimosso not b perché è un letterale negativo e la seconda regola viene rimossa perché not d è un letterale falso rispetto a I. I è un Answer Set per P ridotto dunque lo è anche per P.

Una importante estensione della programmazione logica è la programmazione logica disgiuntiva. Essa permette di avere nelle teste delle regole una disgiunzione di predicati, con il significato che se il corpo è vero allora almeno uno dei predicati presenti nella disgiunzione della testa sarà vero. Sintatticamente la disgiunzione è rappresentata dal simbolo | nella testa delle regole.

Nella programmazione logica disgiuntiva i vincoli non sono intesi come in Prolog, ovvero come obbiettivi da raggiungere, bensì come strumenti con cui selezionare i modelli da scartare. I vincoli scartano i modelli in cui tutti gli atomi nel corpo del vincolo sono verificati contemporaneamente. Sintatticamente i vincoli si presentano come delle regole senza testa.

Esempio:

madre(X,Y) | padre(X,Y) :- genitore(X,Y). → X è il padre o la madre di Y
:- madre(X,Y), genere(X, maschio). → non è possibile che X sia la madre di Y se X è maschio

I programmi sono quindi un insieme di regole e vincoli. Le soluzioni del programma sono dati dagli Answer Set risultanti. Le regole disgiuntive generano gli Answer Set, i vincoli selezionano quelli da scartare. Questo approccio prende il nome di Guess and Check. La programmazione logica disgiuntiva secondo la semantica dei modelli stabili è anche conosciuta con il nome di Answer Set Programming (ASP) [bonatti].

…

- Estenzioni

Il linguaggio ASP fin qui introdotto può essere usato per risolvere problemi di ricerca complessi, ma non fornisce strumenti per affrontare problemi di ottimizzazione, ad esempio problemi in cui oltre a risolvere il problema di ricerca si desidera minimizzare o massimizzare una data funzione. Per rimediare a queste limitazioni sono stati introdotti i vincoli deboli. Nel linguaggio base, i vincoli rappresentano una condizione che deve essere soddisfatta affinché l' Answer Set sia valido, e per questa ragione sono anche chiamati vincoli forti. I vincoli deboli, invece, permetto di esprimere una condizione preferibilmente da non violare, con lo scopo di preferire quegli Answer Set che minimizzano la violazione di questo tipo di vincoli. In aggiunta, può essere specificato un peso e un livello di priorità alla violazione del vincolo debole. In questo modo, ogni volta che un Answer Set viola un vincolo debole, gli viene aggiunto il relativo peso del vincolo al livello di priorità specificato, potendo così individuare l' Answer Set ottimo. La prima preferenza viene fatta per livello. Partendo dal livello più alto, vengono scartati gli Answer Set che hanno riscontrato pesi sul livello corrente, mentre ce ne sono altri che invece non hanno riscontrato pesi su questo livello. Si passa poi al livello inferiore e si ripete l'operazione finché non si giunge al livello minimo, ovvero il livello per cui tutti gli Answer Set rimanenti hanno riscontrato almeno un peso. A questo punto viene preferito quello che minimizza la somma totale dei pesi su questo livello. L'aggiunta dei vincoli deboli permette di estendere l'approccio Guess and Check aggiungendo la parte di ottimizzazione, in una tecnica di programmazione che prende il nome di GCO (Guess/Check/Optimize), in cui la parte Guess definisce lo spazio di ricerca, la parte Check verifica l'ammissibilità delle soluzioni e la parte Optimize specifica i criteri di preferenza.

Sintatticamente i vincoli deboli si presentano come quelli forti in cui il simbolo :- è sostituito dal simbolo :~ , e il peso e il livello sono espressi in seguito al vincolo in accordo con la specifica sintassi (nell' esempio sottostante sarà usata la sintassi del solver DLV2 in cui in aggiunta alla coppia peso@livello vengono elencate le variabili per le quali si intende instanziare il vincolo [leone]).

Un'altra importante estensione al linguaggio ASP è l'aggiunta delle funzioni aggregate. Ci sono alcune semplici proprietà, spesso richieste nei programmi, che non possono essere espresse in maniera naturale usando la sintassi ASP. Specialmente le proprietà che fanno uso degli operatori aritmetici richiedono una modellazione pesante usando esclusivamente la sintassi ASP di base. Osservazioni simili sono state fatte in contesti come la gestione dei database, nei quali è stato

appunto aggiunto l'uso di funzioni aggregate, e quando l'uso della programmazione ASP si è diffusa largamente il bisogno di integrarvi le funzioni aggregate è apparso necessario. Si definisce un insieme simbolico di elementi attraverso i predicati e poi si chiama la funzione desiderata su quell'insieme. Le funzioni più comuni sono il conteggio (#count), la somma (#sum), il minimo (#min) e il massimo (#max).

Esempio:

```
numero(1). numero(2).                    → fatti
scelgo(X) | nonScelgo(X) :- numero(X).   → guess
:- #count{ X : scelgo(X) } > 1.          → check
:~ nonScelgo(X). [X@X, X]                → optimize
```

Answer Sets:

```
A1 = { scelgo(1), scelgo(2) }              →  pesi  [2:2] [1:1]  → inammissibile
A2 = { scelgo(1), nonScelgo(2) }           →  pesi  [2:2] [0:1]  →  secondo
A3 = { nonScelgo(1), scelgo(2) }           →  pesi  [0:2] [1:1]  →  ottimo
A4 = { nonScelgo(1), nonScelgo(2) }  →  pesi  [2:2] [2:1]  →  terzo
```

Notiamo che A1 è inammissibile perché la funzione #count ritorna il valore 2 e il vincolo forte viene violato. L' Answer Set ottimo è A3 perché il livello minimo è il livello 1, a A2 e A4 vengono scartati in quanto hanno dei pesi a livello 2.

…

Programmzione PDDL

- Introduzione ai problemi di pianificazione

La pianificazione rappresenta un'importante attività di problem solving ed è una componente particolarmente importante dell'intelligenza artificiale. Con questo termine si intende l'elaborazione di un piano di azione per raggiungere un determinato obiettivo.

Un problema di pianificazione è dunque un problema di ricerca di una sequenza di azioni che, rispettando i vincoli imposti dal problema, porti al raggiungimento di un obiettivo. Problemi di questo tipo si compongono di tre elementi: uno stato iniziale (la situazione di partenza), uno stato finale (la situazione in cui vorremmo arrivare, l'obiettivo da raggiungere) e un insieme di azioni che, quando applicate, provocano un cambiamento di stato.

Un'azione può essere o meno applicabile in un certo stato e, se applicata, ha delle conseguenze sugli elementi del dominio e porta in un altro stato. Gli stati, all'interno del problema, sono delle descrizioni del dominio in cui stiamo operando e ne rispettano i vincoli. Le azioni consistono in un insieme di precondizioni e in un insieme di effetti. Le prime sono dei vincoli che definiscono se una azione può essere applicata o meno in un certo stato, i secondi rappresentano il risultato dell'esecuzione di una azione in termini di cosa cambia all'interno del dominio. Risolvere un problema di pianificazione vuol dire chiedersi se esiste una sequenza ordinata di azioni che porti dallo stato iniziale a quello finale.

Per realizzare un piano, con il tempo si sono tentati principalmente due approcci: un approccio manuale e uno automatico.

La pianificazione manuale prevede la costruzione del piano da parte di un operatore umano. Il risultato di una operazione di questo tipo dipende molto dalle capacità e dall'esperienza dell'operatore nell'ordinare i compiti rispettando i vincoli del problema. Naturalmente l'utilizzo di questo approccio risulta molto complesso e costoso in termini di tempo, ed è possibile solamente per problemi non particolarmente articolati, in cui il numero di variabili e di vincoli è piccolo, e per problemi riguardanti ambienti completamente deterministici e prevedibili. Inoltre, anche riuscendo a trovare un piano, è molto improbabile che esso sia ottimale, per via dell'enorme quantità di fattori di cui l'operatore deve tener conto.

La pianificazione automatica, al contrario, delega il compito di costruire il piano interamente al sistema intelligente. Questo approccio permette di risparmiare molto tempo rispetto a quello manuale e può garantire una soluzione ottima. Non si può però considerare come una tecnica perfetta. È possibile infatti che una soluzione trovata dal pianificatore automatico, non sia poi effettivamente applicabile in quanto il sistema può non essere a conoscenza di informazioni non fornitegli perché impossibili da esprimere. Inoltre, anche la pianificazione automatica può funzionare solamente con domini deterministici e prevedibili.

La pianificazione, così come descritta fino ad ora, è detta classica o proposizionale, ed è in grado di trattare problemi non particolarmente complessi, nei quali c'è la necessità di capire cosa fare e in quale ordine farlo, ma non c'è bisogno di tener conto di fattori che durante l'esecuzione di un piano possono variare, quali il tempo o il consumo di risorse. Quando si trattano problemi reali, tuttavia, si ha a che fare con ambienti molto dinamici e scarsamente prevedibili, di cui si ha spesso una conoscenza solo parziale e in cui c'è la possibilità che si verifichino, a tempo di esecuzione, degli inconvenienti inaspettati che possono andare a modificare il mondo, rendendo inconsistente il piano realizzato in precedenza.

Fino a qualche anno fa, per via delle limitate capacità tecnologiche, la ricerca sulla pianificazione si è limitata quasi esclusivamente a problemi di questo tipo. Negli ultimi anni, invece, ci si è avvicinati sempre più alla pianificazione di problemi realistici. Quando si ha a che fare con problemi reali complessi sorge la necessità di gestire anche elementi variabili. Ogni azione che deve essere eseguita, infatti, ha una durata temporale e consuma una determinata quantità di risorse. In questi casi, perciò, occorre effettuare una pianificazione che tenga conto di tali vincoli, associando ad ogni azione il corrispondente tempo necessario per eseguirla e la quantità di risorse consumate. Soluzioni a problemi di questo tipo devono dunque soddisfare, oltre ai vincoli proposizionali, anche i vincoli numerici riguardanti le risorse consumabili. Inoltre devono tener conto dei vincoli numerici di ordine temporale, i quali possono essere associati sia alla durata del piano in sè, sia allo stesso utilizzo delle risorse, in relazione alla loro disponibilità rinnovabile/riutilizzabile o meno. Questi accenni lasciano intuire quanto la pianificazione con vincoli su risorse consumabili sia tutt'altro che banale e le ragioni per le quali essa sia stata affrontata solamente in un tempo recente.

- STRIPS - PDDL

A livello pratico, la pianificazione classica si può trattare come un problema di ricerca nello spazio degli stati. Per rappresentare un problema è ormai consuetudine utilizzare linguaggi STRIPS-like. STRIPS (Stanford Research Institute Problem Solver) è un linguaggio basato sulla logica proposizionale e sull'uso di predicati. Esso permette di rappresentare ogni stato attraverso l'insieme di predicati che sono validi in quel determinato stato. Le azioni sono invece rappresentate attraverso

precondizioni ed effetti. Le precondizioni includono i predicati che devono essere veri affinché si possa applicare l'azione. Gli effetti sono invece le conseguenze dell'applicazione dell'azione, ossia le modifiche che un'azione comporta sul dominio.

STRIPS fa un'assunzione di mondo chiuso, cioè suppone che ogni fatto non espressamente indicato sia falso. Inoltre, mentre si effettua una operazione, si assume che nel mondo non succeda nient'altro. Per questa ragione negli effetti di una azione si specificano solamente i cambiamenti che essa comporta, mentre tutto il resto non è esplicitato. Questo è importante in quanto nella maggior parte degli ambienti in cui si utilizza STRIPS, c'è una grande quantità di fatti che non cambiano nel tempo. Specificare solamente ciò che si modifica, dunque, permette di evitare di dover ricopiare completamente l'intero modello del mondo ogni volta che si passa da uno stato ad un altro. Gli effetti di una azione sono indicati come due liste: una DELETE-LIST, che contiene tutte le clausole che devono essere rimosse (e quindi implicitamente rese false), e una ADD-LIST, che contiene tutte le clausole che non sono presenti nello stato corrente e devono essere aggiunte (rese vere).

Una volta definito il problema, il compito del pianificatore è verificare se esiste una sequenza di azioni che faccia passare dallo stato iniziale a quello finale, e in tal caso, dire qual è. Per fare ciò il sistema intelligente esamina lo spazio degli stati, simulando, in ogni stato, l'applicazione di tutte le azioni applicabili (quelle le cui precondizioni sono soddisfatte), fino ad ottenere lo stato finale.

Intorno agli anni 2000, in seguito alle prime due edizioni dell' International Planning Competition (IPC), si è affermato all'interno della comunità internazionale, come linguaggio standard per la rappresentazione di domini, PDDL, un linguaggio sviluppato da Drew McDermott e da alcuni suoi colleghi. Attraverso tale standardizzazione si è cercato di incentivare il progresso nel campo della pianificazione, favorendo la comunicazione tra i vari gruppi di ricerca, lo scambio e la comparazione dei lavori da essi effettuati.

PDDL (Planning Domain Definition Language), è dunque un Linguaggio di Descrizione dei Domini di Pianificazione. Esso si ispira a STRIPS e alla formulazione dei problemi secondo la sua logica. In particolare PDDL contiene STRIPS, e i domini scritti in PDDL si possono scrivere in STRIPS. Ci sono comunque alcune differenze tra i due linguaggi. In PDDL gli effetti di una azione non sono esplicitamente divisi in ADD-LIST e DELETE-LIST ma gli effetti negativi sono indicati con una negazione e sono messi in congiunzione insieme a quelli positivi.

Lo sviluppo della pianificazione con vincoli di risorse è stato favorito, in particolare, dal rilascio di PDDL 2.1 il quale fornisce una notevole potenza di modellazione. Esso è infatti in grado di modellare classi di domini che richiedono una pianificazione dell'uso delle risorse e del tempo, tramite l'uso di fluenti numerici, permettendo di dividere, all'interno del problema, la parte predicativa da quella numerica.

Tra le principali qualità di PDDL vi è una divisione delle caratteristiche di un dominio da quelle dell'istanza di un problema. Esso permette, cioè, di dividere la descrizione di azioni dalla descrizione di oggetti specifici o di condizioni iniziali e finali. In questo modo è possibile utilizzare delle variabili per parametrizzare le azioni e ciò permette di riutilizzare un dominio con istanze di problemi diversi, favorendo un'analisi di scenari differenti applicati allo stesso dominio.

Esempio (dominio):

```
(define (domain blocks)
  (:requirements :strips :typing :fluents)
  (:types block)
  (:predicates (on ?x - block ?y - block) (ontable ?x - block)
```

```
              (clear ?x - block) (handempty) (holding ?x - block) )
  (:functions (moves) )
  (:action pick-up
     :parameters (?x - block)
     :precondition (and (clear ?x) (ontable ?x) (handempty))
     :effect (and (not (ontable ?x)) (not (clear ?x))
                  (not (handempty)) (holding ?x) (increase (moves) 1) ) )
  (:action put-down
     :parameters (?x - block)
     :precondition (holding ?x)
     :effect (and (not (holding ?x)) (clear ?x) (handempty)
                  (ontable ?x) (increase (moves) 1) ) )
  (:action stack
     :parameters (?x - block ?y - block)
     :precondition (and (holding ?x) (clear ?y))
     :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x)
                  (handempty) (on ?x ?y) (increase (moves) 1) ) )
  (:action unstack
     :parameters (?x - block ?y - block)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect (and (holding ?x) (clear ?y) (not (clear ?x))
                  (not (handempty)) (not (on ?x ?y)) (increase (moves) 1) ) ) )
```

In esso si può notare la descrizione delle azioni parametrizzate al fine di lasciare poi all'istanza del problema il compito di sostituire i parametri formali con dei parametri attuali. Un' importante caratteristica di PDDL riguarda quindi la definizione dei tipi. I tipi dei parametri di una azione vengono specificati esplicitamente all'interno di :parameters in cui una variabile è indicata con il punto interrogativo e il suo tipo è preceduto dal simbolo - . Le stesse variabili definite nell'elenco degli argomenti, sono poi utilizzate nella descrizione delle precondizioni e degli effetti dell'azione. I nomi dei tipi devono essere dichiarati attraverso :types e se si intende usarli è anche necessario dichiararlo all'interno dei requisiti (:requirements :typing …).

Esempio (problema):

```
(define (problem blocks-01)
  (:domain blocks)
  (:objects A B C D - block)
  (:init (clear A) (clear B) (clear C) (clear D) (ontable A)
         (ontable B) (ontable C) (ontable D) (handempty) (= (moves) 0) )
  (:goal (and (on D C) (on C B) (on B A) ) )
  (:metric minimize (moves))
)
```

In esso viene invece mostrata la descrizione di un problema molto semplice. In questo caso, si può notare come le descrizioni degli oggetti specifici, le condizioni iniziali e i goals, vengano indicate senza l'uso di parametri, in quanto stiamo configurando l'istanza di un problema.

Questi esempi sono espressi in una rappresentazione che è già propria di PDDL 2.1. Vengono già utilizzati, infatti, i fluenti numerici, e il loro uso va specificato all'interno dei requisiti (:requirements :fluents …). Elementi di questo tipo sono molto importanti, soprattutto per quanto riguarda la pianificazione di problemi realistici, in quanto permettono di modellare risorse non binarie. Essi sono indicati separatamente dai predicati. All'interno della definizione di un dominio i predicati vengono specificati in :predicates, mentre le funzioni numeriche sono inserite in :functions.

I predicati possono essere veri o falsi e il loro significato non è intrinseco, dipende invece dagli effetti che le azioni del dominio possono avere su di essi. Si possono distinguere a livello

concettuale (a livello sintattico non ci sono differenze) predicati statici e predicati dinamici. I primi sono predicati che non vengono cambiati da nessuna azione e che quindi dipendono solamente dallo stato iniziale del problema. I secondi invece sono quei predicati il cui valore viene alterato dalle azioni. Le funzioni, invece, possono essere viste come dei predicati a cui è associato un valore numerico anziché un valore booleano. Anche per esse, in generale, valgono le stesse considerazioni fatte per i predicati. Sia i predicati che le funzioni possono essere utilizzati come vincoli all'interno delle precondizioni di una azione, e i loro valori possono essere alterati dagli effetti. I vincoli numerici possono essere costruiti, usando operatori aritmetici, a partire da espressioni numeriche primitive.

In PDDL 2.1, oltre ai fluenti numerici, è stata anche introdotta la possibilità di definire delle metriche. Esse permettono di specificare i criteri su cui un piano deve essere valutato. L'introduzione di tali vincoli è dovuta alla possibilità di utilizzare dei fluenti numerici. Essi permettono di chiedere al sistema che il piano generato, oltre ad essere consistente, minimizzi o massimizzi una certa espressione. Naturalmente l'espressione da minimizzare (o massimizzare) può essere molto complessa, ottenendola dalla combinazione di più funzioni.