EmbASP, un framework per l'integrazione della programmazione logica in sistemi esterni

Indice

1 Introduzione

Cos'è l'Intelligenza Artificiale? In modo semplicistico potremmo definire l'Intelligenza Artificiale come l'abilità di un sistema tecnologico di risolvere problemi o svolgere compiti e attività tipici della mente e dell'abilità umane. Guardando al settore informatico, potremmo identificare l'Intelligenza Artificiale come la disciplina che si occupa di realizzare macchine (hardware e software) in grado di "agire" autonomamente (risolvere problemi, compiere azioni, ecc.).

Il fermento attuale attorno a questa disciplina si spiega con la maturità tecnologica raggiunta sia nel calcolo computazionale sia nella capacità di analisi in real-time di enormi quantità di dati e di qualsiasi forma. L'interesse della comunità scientifica per l'Intelligenza Artificiale ha però origini molto lontane: il primo vero progetto di Intelligenza Artificiale (ormai nota con l'acronimo AI - Artificial Intelligence) risale al 1943 quando i due ricercatori statunitensi Warren McCulloch e Walter Pitts proposero al mondo scientifico il primo neurone artificiale. Ne seguì, nel 1949, il libro «L'organizzazione del comportamento» di Donald Olding Hebb, psicologo canadese, grazie al quale vennero analizzati in dettaglio i collegamenti tra i neuroni artificiali ed i modelli complessi del cervello umano. I primi prototipi funzionanti di reti neurali (cioè modelli matematici/informatici sviluppati per riprodurre il funzionamento dei neuroni biologici per risolvere problemi di Intelligenza Artificiale) arrivarono poi verso la fine degli anni '50. L'interesse del pubblico si fece maggiore grazie soprattutto al giovane Alan Turing che già nel 1950 cercava di spiegare come un computer possa comportarsi come un essere umano. La sua formalizzazione dei concetti di algoritmo e calcolo mediante l'omonima «macchina di Turing» da lui progettata, costiturono un significativo passo avanti nell'evoluzione verso il moderno computer, e per questo contributo è solitamente considerato il padre dell'Informatica e dell'Intelligenza Artificiale. Il termine Intelligenza Artificiale in realtà parte "ufficialmente" dal matematico statunitense John McCarthy, nel 1956, e con esso i primi linguaggi di programmazione (Lisp nel 1958 e Prolog nel 1973) specifici per l'AI. Da lì in poi la storia dell'Intelligenza Artificiale è stata abbastanza altalenante, caratterizzata da avanzate significative dal punto di vista dei modelli matematici ma con alti e bassi dal punto di vista della ricerca sull'hardware e sulle reti neurali. La prima grande svolta su quest'ultimo fronte è arrivata negli anni '90 con l'ingresso sul mercato "allargato" (arrivando cioè al grande pubblico) dei processori grafici, le Gpu (chip di elaborazione dati molto più veloci delle Cpu, provenienti dal mondo del gaming ed in grado di supportare processi complessi molto più rapidamente). L'ondata più recente è arrivata nell'ultimo decennio con lo sviluppo dei cosiddetti "chip neuromorfici", ossia microchip che integrano elaborazione dati e storage in un unico micro componente per emulare le funzioni sensoriali e cognitive del cervello umano.

Prendendo come base di partenza il funzionamento del cervello umano (pur sapendo che ancora oggi non se ne comprende ancora a fondo l'esatto meccanismo), un'Intelligenza Artificiale dovrebbe saper compiere alcune azioni/funzioni tipiche dell'uomo:

- agire umanamente (cioè in modo indistinto rispetto ad un essere umano);
- pensare umanamente (risolvendo un problema con funzioni cognitive);
- pensare razionalmente (sfruttando cioè la *logica* come fa un essere umano);

- agire razionalmente (avviando un processo per ottenere il miglior risultato atteso in base alle informazioni a disposizione, che è ciò che un essere umano, spesso anche inconsciamente, fa d'abitudine).

Queste considerazioni sono di assoluta importanza perché permettono di classificare l'AI in due grandi "filoni" di indagine:

- Intelligenza Artificiale debole (weak AI). Identifica sistemi tecnologici in grado di simulare alcune funzionalità cognitive dell'uomo senza però raggiungere le reali capacità intellettuali tipiche dell'uomo (parliamo di programmi matematici di problem-solving con cui si sviluppano funzionalità per la risoluzione dei problemi o per consentire alle macchine di prendere decisioni);

- Intelligenza Artificiale forte (strong AI). In questo caso si parla di "sistemi sapienti" (alcuni scienziati si spingono a dire addirittura "coscienti di sé") che possono quindi sviluppare una propria intelligenza senza emulare processi di pensiero o capacità cognitive simili all'uomo ma sviluppandone una propria in modo autonomo.

Lo sviluppo dell'Intelligenza Artificiale ha avuto molta influenza in diversi campi dell'Informatica, poichè la realizzazione dei sistemi di AI ha portato allo sviluppo di tecniche e strumenti innovativi. Questo è certamente il caso dei linguaggi di programmazione: un campo nel quale molti linguaggi sono stati sviluppati inizialmente per rendere più efficiente la realizzazione di sistemi di AI. In particolare lo studio di metodi e tecniche per la rappresentazione della conoscenza ha portato alla realizzazione di ambienti basati sulla *logica*, che offrono al programmatore la possibilità di sviluppare un sistema formalizzando nel linguaggio di rappresentazione le conoscenze sul dominio e offrendo strumenti per derivare le conseguenze delle conoscenze specificate. I linguaggi utilizzati per rappresentare la conoscenza sono detti linguaggi di programmazione dichiarativi, nel senso che consentono la specifica del problema, lasciando agli strumenti offerti dal sistema il compito di trovare algoritmicamente la soluzione.

In genere un elaboratore offre la possibilità di usare uno o più linguaggi di comandi (o linguaggi macchina) ed uno o più linguaggi di programmazione. Questi due tipi di linguaggi hanno, nella maggior parte dei casi, scopi distinti: i linguaggi di comandi vengono utilizzati per la creazione, l'esecuzione e la gestione dei programmi; i linguaggi di programmazione vengono utilizzati dall'utente per scrivere i programmi che risolvono i problemi specifici alle sue applicazioni. La distinzione tra linguaggio di comandi e linguaggio di programmazione deriva da esigenze di costruzione piuttosto che da un reale vantaggio per l'utente. Infatti, da un punto di vista realizzativo, è conveniente separare le operazioni che consentono l'utilizzo dell'elaboratore (linguaggio di comandi), dal linguaggio di programmazione, che permette la specifica di operazioni di carattere più generale. I linguaggi di programmazione evoluti che sono stati introdotti a partire dagli anni '50 sono detti linguaggi ad alto livello; essi sono stati progettati con lo scopo di rendere il linguaggio indipendente dalle caratteristiche dell'elaboratore e più vicino alla logica del programmatore. I programmmi scritti in un linguaggio ad alto livello non sono direttamente eseguibili dall'elaboratore, ma devono essere prima tradotti da un interprete o da un compilatore nel linguaggio macchina dell'elaboratore. Osserviamo inoltre che i linguaggi ad alto livello consentono di utilizzare lo stesso programma anche su macchine di tipo diverso.

Linguaggi di questo tipo sono detti linguaggi imperativi o anche linguaggi di Von Neumann, perchè fanno riferimento all'omonimo modello di calcolo. Come i linguaggi di basso livello, essi fanno riferimento al modello di esecuzione delle istruzioni fornito dall'elaboratore: un programma è cioè costituito da una sequenza di istruzioni il cui effetto è quello di modificare il contenuto della memoria dell'elaboratore. I linguaggi per l'AI appartengono ad altre categorie di linguaggi di programmazione che prescindono dal modello di funzionamento dell'elaboratore, cercando di fornire un mezzo espressivo per specificare il compito da eseguire in modo semplice e sintetico. Questi tipi di linguaggi vengono anche detti non Von Neumann, proprio perchè sono svincolati dal modello su cui si basano i linguaggi imperativi, e fanno uso della *logica*.

Lo scopo di questa tesi è la presentazione di EmbASP, un framework sviluppato dal dipartimento di Matematica e Informatica dell'Università della Calabria, che permette l'integrazione della programmazione logica in sistemi esterni. Verranno esposti i principi fondamentali dei due linguaggi logici attualmente supportati da EmbASP (ASP e PDDL) e verrà esposto come viene fatta la loro integrazione nei tre linguaggi di programmazione in cui il framework è implementato (Java, Python e C#). La tesi è strutturata come segue:

- Nel Capitolo 2 verrà esposta la programmazione ASP (Answer Set Programming), partendo dai principi base della programmazione logica. Sarà esposto il linguaggio Prolog, che rappresenta le origini del linguaggio ASP. Sarà in seguito analizzata la semantica del liguaggio ASP e saranno introdotte la programmazione logica disgiuntiva e le principali estenzioni sintattiche.

- Nel Capitolo 3 varrà esposta la programmazione PDDL (Planning Domain Definition Language), introducendo i problemi di pianificazione e presentando poi STRIPS, un linguaggio basato sulla logica in grado di affrontare tali problemi. Verrà poi esposto come PDDL implementa STRIPS e come PDDL2.1 favorisce la pianificazione con vincoli sulle risorse.

- Nel capitolo 4 sarà presentato EmbASP. Ne sarà analizzata la struttura e verranno esposti degli esempi pratici del suo funzionamento.

- Nel capitolo 5 verranno esposte le principali attività effettuate durante il periodo di tirocinio da me svolto presso il dipartimento di Matematica e Informatica dell'Università della Calabria, durante il quale ho avuto l'opportunità di collaborare con alcuni membri del dipartimento per contribuire all'evoluzione di EmbASP.

2 Programmazione ASP

2.1 Introduzione alla programmazione logica

La logica (dal greco logos, ovvero "parola", "pensiero", "idea") è uno strumento di espressione che trae origine dall'esigenza di formalizzare il ragionamento umano. La logica è tradizionalmente una delle discipline filosofiche, ma essa riguarda anche numerose attività tecniche e scientifiche, tra cui matematica, semantica e informatica.

Nella logica proposizionale esistono due valori (I cioè vero, e O cioè falso) e si basa 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.

Gli elementi sintattici di base della logica proposizionale sono i seguenti:

- Un insieme di proposizioni.
- Il seguente connettivo logico unario:
  - Negazione: $\neg$
- I seguenti connettivi logici binari:
  - Disgiunzione: $\bigvee$
  - Congiunzione: $\bigwedge$
  - Implicazione: $\rightarrow$
  - Doppia implicazione: $\leftrightarrow$

La logica proposizionale non è, però, 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.

La logica dei predicati ha un quantificatore universale denotato con $\forall$ ("per ogni") ed un quantificatore esistenziale denotato con $\exists$ ("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 : Robert 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 Alain 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 un predicato, e viene definito 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 $\bigvee$ $(\bigvee_{i=1}^{n} \neg p_i)$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow$ p.

2.2 Prolog

Nato nel 1972 grazie alle ricerche di Alain Colmerauer e Robert 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(t_1, ... , t_n) dove p è il nome del predicato, t_1, ... , t_n 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 ($\forall$, $\exists$) 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 appaiono 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 antenato di Y

antenato(X,Y) :- genitore(X,Z), antenato(Z,Y). → chiusura transitiva

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 [1]. 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 ed un obiettivo, bisogna trovare per tutti i predicati nell'obiettivo considerato, dei fatti che siano unificabili con quei predicati o delle regole le cui teste siano unificabili con quei predicati. Nel caso la risoluzione avvenga con una regola, bisogna poi procedere nello stesso modo con i predicati del corpo della regola considerati da sinistra a destra, seguendo l'ordine LIFO per coerenza con la strategia SLD. Per motivi di efficienza, conviene che le regole ricorsive siano ricorsive a destra, cioè che il predicato in testa di tali regole compaia nei predicati più a destra del corpo delle regole. Poiché ogni predicato presente in un obiettivo potrebbe essere unificabile con più fatti e teste di regole di un programma, in Prolog la risoluzione dei predicati viene esaminata rispetto a tutte le regole del programma, considerate nell'ordine in cui sono state 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 [1].

La radice dell'albero e-o contiene l'obiettivo iniziale, mentre ogni foglia può contenere # oppure un obiettivo al cui predicato 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 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).
```

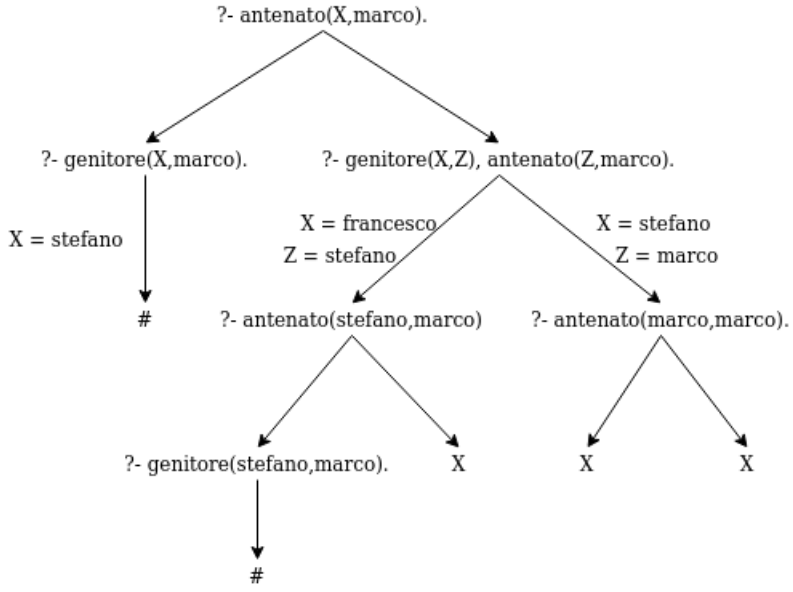
Consideriamo poi l'obiettivo:

```
?- 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:



2.3 Semantica ASP

Nel capitolo precedente abbiamo brevemente descritto il metodo di risoluzione SLD adottato nella realizzazione di interpreti Prolog reali. Ciò che in pratica abbiamo fornito è una

semantica operazionale della programmazione logica. Ci si può chiedere a questo punto se esista un modo per descrivere il significato di un programma logico senza fare riferimento a particolari strategie o metodi di risoluzione. Questo paragrafo illustra l'Answer Set Programming (ASP), un approccio alternativo introdotto nel 1993, il quale si estende più in generale a tutta la programmazione dichiarativa. Questo stile di programmazione risulta sufficientemente espressivo da essere proficuamente utilizzabile in diversi contesti.

L'Answer Set Programming, si differenzia in molteplici punti dalla programmazione Prolog, pur ereditandone quasi interamente la sintassi fin qui introdotta. Ciò che differenzia i due approcci è, da un lato il modo in cui viene assegnata la semantica ad un programma, dall'altro la procedura impiegata per trovare le soluzioni. Vediamo in modo informale una serie di punti in cui i due approcci si differenziano:

- In una regola ASP l'ordine dei predicati non ha alcuna importanza. Ciò è in contrasto con le convenzioni adottate in Prolog. Abbiamo infatti visto come il comportamento dell'interprete Prolog sia fortemente influenzato dall'ordine in cui i predicati occorrono in una regola. In ASP quindi il corpo di una regola si può considerare come un vero e proprio insieme di letterali.

- In Prolog l'esecuzione di un goal avviene in modo top-down e goal-directed: a partire dal goal si procede utilizzando le clausole e costruendo una derivazione che porta alla soluzione/risposta. Contrariamente a ciò, solitamente un interprete per l'ASP opera bottom-up, partendo dai fatti si procede verso le conclusioni che portano alla soluzione.

- La procedura risolutiva solitamente implementata in Prolog (risoluzione SLD), con le sue particolari scelte relative alle regole di selezione di letterali e clausole, può causare la generazione di computazioni infinite (anche in assenza della negazione). Ciò non accade invece per i solver ASP.

Le differenze tra Prolog e ASP nascono essenzialmente dal modo in cui viene assegnata la semantica ai programmi. Possiamo dire che l'ASP sia una forma di programmazione dichiarativa in cui si fissa la semantica dei programmi. Per descrivere la semantica di un programma ASP dobbiamo prima introdurre la nozione di *Answer Set*. Innanzitutto affrontiamo il caso più semplice dei programmi ground. Un termine (un predicato, una regola o un programma) è detto ground se non contiene nessuna variabile [2]. 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 predicati 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, la verità del corpo della regola implica la verità della testa.

Esempio:

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

d.

Interpretazioni:

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

Notiamo che 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. Dato un programma P e un interpretazione I, si definisce conseguenza immediata di I l'insieme di tutti i predicati che compaiono in testa a qualche regola in P di cui il corpo è vero rispetto ad I. Applicando questo metodo più volte ad un programma, 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.
b.

Tp(0) = {b} → interpretazione di partenza (fatti)
Tp(1) = {a}. → viene derivato a
Tp(2) = {e}. → modello minimale = {b, a, e}

In un programma positivo (ovvero che non contiene negazione in nessuna regola) e ground, si ha la certezza che il Tp raggiunga un punto fisso, riuscendo a trovare il modello minimale. In un programma positivo non ground, il Tp inizia con i predicati EDB e iterativamente deriva i predicati IDB. Se il programma contiene regole ricorsive, questo processo potrebbe anche non raggiungere un punto fisso, e continuare a generare nuovi predicati IDB all'infinito. Inoltre, in un programma non positivo, la negazione può causare problemi se unita alla ricorsione. Per escludere la negazione all'interno della ricorsione, i programmi devono essere stratificati. 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 relativo 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 generico 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.}

Notiamo che 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.

2.4 Estensioni

Una importante estensione della programmazione logica è la programmazione logica disgiuntiva. Essa è un potente formalismo per la rappresentazione della conoscenza e il ragionamento di senso comune, che consente di formalizzare, in modo semplice e naturale, problemi complessi. La programmazione logica disgiuntiva permette di avere nelle teste delle regole una disgiunzione di predicati, con il significato che se il corpo è vero allora uno dei predicati presenti nella disgiunzione della testa sarà vero. Quello che ne risulta è un linguaggio la cui alta espressività ha importanti implicazioni pratiche, consentendo la modellazione di situazioni reali, quali la rappresentazione di conoscenza incompleta. 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 gli Answer Set da scartare. I vincoli scartano gli Answer Set in cui sono verificati contemporaneamente tutti i predicati nel corpo del vincolo. Sintatticamente i vincoli si presentano come delle regole senza testa.

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).

Esempio:

a | b | c :- d. → guess
:- d, c. → check
d.

Answer Sets:

A1 = {d, a}
A2 = {d, b}
A3 = {d, c} → inammissibile

Notiamo che la regola disgiuntiva genera tre Answer Set, ma A3 risulta inammissibile in quanto viola il vincolo (d e c sono veri entrambi contemporaneamente). Gli Answer Set risultanti sono dunque A1 e A2.

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 [2]).

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 Set:

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.

3 Programmazione PDDL

3.1 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.

3.2 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

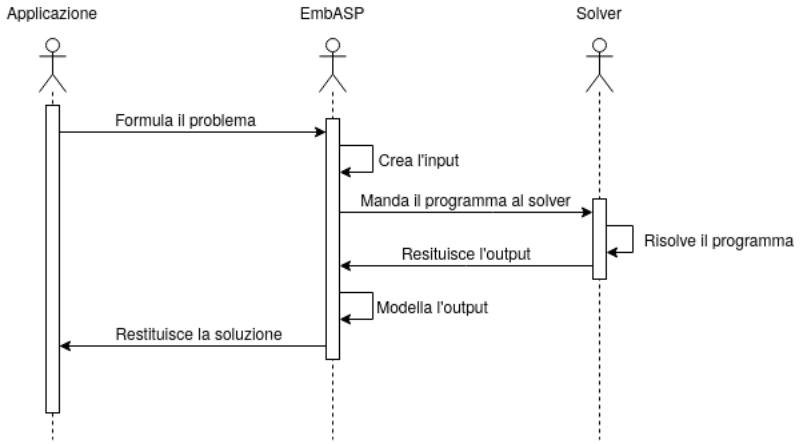
azione, e i loro valori possono essere alterati dagli effetti. I vincoli numerici possono essere costruiti, usando operatori aritmetici, a partire da espressioni numeriche primitive.

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.

4 EmbASP

4.1 Presentazione

EmbASP è un framework sviluppato dal Dipartimento di Matematica e Informatica dell'Università della Calabria che supporta l'integrazione delle programmazione logica in sistemi esterni. Esso cioè consente la creazione di applicazioni in cui sono trattati problemi complessi, usando la programmazione logica per la loro risoluzione. L'utilizzo di linguaggi logici all'interno di applicazioni generiche scritte il linguaggi di programmazione ad oggetti, permette di inserire al loro interno una componente intelligente, integrando maccanismi di Intelligenza Artificiale in applicazioni normalmente prive. Questo permette di modellare il problema in maniera molto intuitiva, affidando il compito di risolverlo al sistema, e lasciando al programmatore il solo compito di formularlo. Problemi complessi che normalmente richiederebbero algoritmi di alto livello per essere risolti, possono così essere dichiarati e risolti in maniera semplice e naturale. L'integrazione della programmazione logica avviene attraverso l'esecuzione di un solver, uno strumento informatico in grado di risolvere il problema logico. Dato in input il programma, il solver lo risolve e comunica in output la soluzione. EmbASP si occupa di creare l'input rappresentante il programma logico e trasformarlo in un formato in grado di essere passato al solver. Al termine della risoluzione EmbASP raccoglie l'output e lo modella in un formato con cui l'applicazione può interagire.

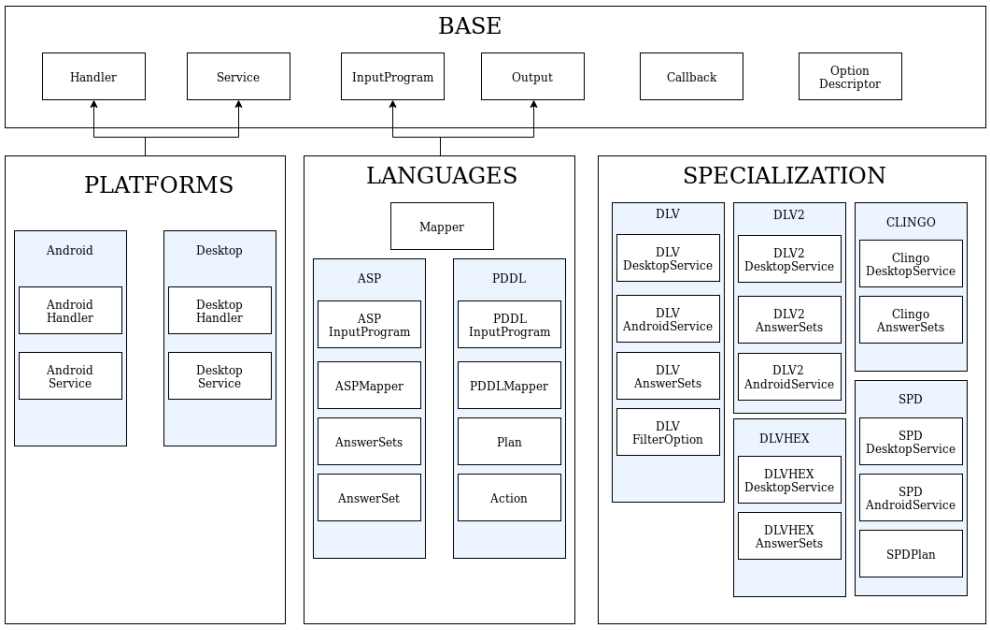


Un framework è una struttura astratta in cui vengono definiti gli elementi che lo compongono e le relazioni tra di essi. EmbASP è stato implementato in 3 linguaggi di programmazione: Java, Python e C#. Implementare un framework significa fornire un'implementazione concreta delle classi astratte. L'insieme delle classi concrete eredita le relazioni definite dal framework e si ottiene in questo modo un insieme di classi concrete messe in relazione tra di loro. EmbASP supporta due linguaggi logici: ASP e PDDL, dei quali si è discusso nei capitoli 2 e 3. Il linguaggio ASP è supportato grazie ai solver DLV, DLV2, DLVHEX e clingo. Questi solver sono file in grado di essere eseguiti da un computer. Possono essere scaricati dalla relativa fonte di appartenenza ed essere eseguiti localmente dal proprio computer. Essi richiedono in input il programma logico,

lo risolvono, e restituiscono in output la soluzione. Il linguaggio PDDL è supportato grazie al solver SPD (Solver.Planning.Domains). Esso usa un cloud server per l'esecuzione remota. Il problema viene generato localmente e inviato tramite internet ad un server che si occupa di risolverlo e restituire la soluzione.

4.2 Struttura

EmbASP è diviso in 4 moduli: Base, Platform, Languages e Specializations.



Nel modulo Base sono presenti le classi che rappresentano i componenti fondamentali del framework, e viene definito in che modo essi sono messi in relazione tra loro. Le classi in questo modulo hanno un alto livello di astrazione, cioè definiscono solo quali sono le funzionalità principali offerte dai singoli componenti, che sono implementati in modo più specifico nei moduli successivi. La classe Handler è il componente principale del framework in quanto gestisce l'intero processo. L'Handler colleziona le informazioni necessarie a creare l'input, fornitegli dall'applicazione attraverso la classe InputProgram. Attraverso la classe OptionDescriptor si posso specificare delle opzioni aggiuntive da cominucare al solver. Queste informazioni vengono poi passate alla classe Service che avvia l'esecuzione del solver e gli trasmette le informazioni. Terminata l'esecuzione, il Service riceve l'output dal solver e lo passa all'Handler. L'output viene poi modellato grazie alla classe Output o attraverso la classe Callback, in modo che l'applicazione possa analizzarlo e gestirlo secondo le proprie necessità.

Nel modulo Platform sono presenti le estensioni delle classi Handler e Service che gestiscono le piattaforme supportate da EmbASP: Desktop in tutte e tre le versioni e Android solo sulla versione Java. La classe DesktopHandler gestisce il DesktopService per l'esecuzione del solver su una piattaforma Desktop e, in modo simile, la classe An-

droidHandler gestisce l'AndroidService sulla piattaforma Android.

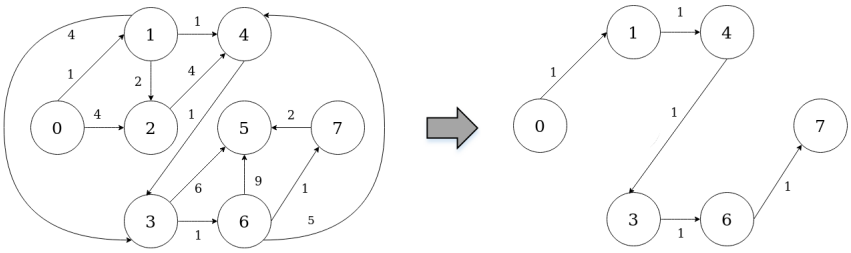
Nel modulo Languages sono presenti le estensioni delle classi InputProgram e Output, le quali hanno lo scopo di gestire l'input e l'output dei differenti solver. Per il linguaggio ASP l'InputProgram è esteso dalla classe ASPInputProgram e l'Output è esteso dalla classe AnswerSets, composto da un insieme di AnswerSet. Mentre l'InputProgram del linguaggio PDDL è esteso dalla classe PDDLInputProgram e l'Output dalla classe Plan, composto da un insieme di Action. L'InputProgram permette la creazione dell'input del programma logico in 3 modi: aggiungendo le informazioni sotto forma di stringhe, da essere poi passate direttamente al solver; creando uno o più file in cui sono contenute le informazioni e specificando gli indirizzi di tali file; oppure creare delle classi che rappresentano i predicati del programma logico e aggiungere all'input le loro istanze (oggetti). Questo procedimento è assistito dalla classe Mapper, specializzato nell'ASPMapper e nel PDDLMapper, e da ANTLR, uno strumento generatore di parser e traduttori che permette di definire grammatiche, di cui EmbASP fa uso. Le classi che intendono rappresentare predicati del programma logico devono essere arricchite con delle informazioni che il Mapper è in grado di elaborare, al fine di convertire gli oggetti in stringhe (in grado di essere passate al solver). Per effettuare ciò la versione Java usa le Java Annotation, una forma di meta-codice che marca le classi con delle informazioni che sono analizzate a tempo di esecuzione. Le classi che rappresentano un predicato specifico devono essere marcate con l'annotazione @Id(nome_del_predicato) e ogni parametro della classe deve essere annotato con @Param(posizione). Grazie alle Java Reflection le annotazioni vengono esaminate e gli oggetti vengono tradotti in stringe e passati al solver. Nella versione Python viene usata una classe astratta Predicate. Le classi che rappresentano i predicati devono estendere la classe Predicate. Devono contenere al loro interno un parametro di nome "predicate_name", contenente una stringa rappresentante il nome del predicato, e una lista contenente il nome dei parametri che il predicato ha. Queste informazione sono poi usate dal Mapper per eseguire correttamente la conversione delle classi in stringhe. Nella versione C#, in modo simile alla versione Java, le classi che rappresentano i predicati vengono marcate attraverso i C# Attributes. La classe deve essere marcata con [Id(nome_del_predicato)] e i parametri con [Param(posizione)] per poter permettere al Mapper, con il supporto della C# Reflection, di eseguire la traduzione.

Nel modulo Specialization sono contenute le specifiche estensioni delle classi Service e Output, per poter gestire l'esecuzione dei differenti solver. Nello specifico, in tutte e tre le versioni sono contenute le estensioni della classe DesktopService per i differenti solver (DLVDesktopService, DLV2DesktopService, DLVHEXDesktopService, ClingoDesktopService e SPDDesktopService) mentre solo nella versione Java sono presenti le estensioni dell' AndroidService e solo per i solver DLV, DLV2 e SPD (DLVAndroidService, DLV2AndroidService e SPDAndroidService). Sono contenute anche le estensioni della classe AnswerSets per il linguaggio ASP (DLVAnswerSets, DLV2AnswerSets, DLVHEXAnswerSets e ClingoAnswerSets) e della classe Plan per il linguaggio PDDL (SPDPlan).

4.3 Esempi

Di seguito sono mostrati due esempi del funzionamento di EmbASP. Nel primo esempio sarà risolto il problema del cammino minimo usando la programmazione ASP, usando la versione Java di EmbASP su una piattaforma Desktop e usando il solver DLV2. Il

problema del cammino minimo è un problema in cui, dato un grafo orientato e pesato, bisogna riuscire a trovare il cammino, da un nodo sorgente a un nodo di destinazione, che minimizzi la somma dei pesi negli archi inclusi in tale cammino. In questo esempio, dato il grafo riportato di seguito, si vuole trovare il cammino minimo dal nodo 0 al nodo 7.



Per effettuare ciò, sono state create due classi, rappresentati gli archi del grafo, che verranno poi usate come predicati del programma logico. La classe Edge rappresenta gli archi in input del programma logico, mentre la classe Path rappresenta gli archi inclusi nel cammino minimo.

```
@Id("edge") public class Edge {

  @Param(0) private int from;
  @Param(1) private int to;
  @Param(2) private int weight;

  public Edge(int from, int to, int weight) {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }

  [...]
}

@Id("path") public class Path {

  @Param(0) private int from;
  @Param(1) private int to;
  @Param(2) private int weight;

  public Path(int from, int to, int weight) {
    this.from = from;
    this.to = to;
    this.weight = weight;
```

```
    }

    [...]
}
```

A questo punto, supponendo di aver scaricato il solver DLV2 sul proprio computer e di averlo incluso nel progetto, possiamo sviluppare la nostra applicazione.

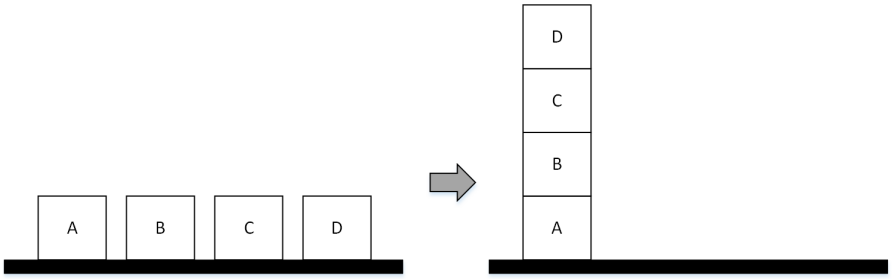
```
public class ShortestPath {

    // nodo sorgente e di destinazione
    private static int from = 0;
    private static int to = 7;

    // archi nel cammino minimo (ordinati)
    private static ArrayList<Integer> sortedPath;

    public static void main(String[] args) {
      try {
        Handler handler = new DesktopHandler(
                new DLV2DesktopService("executable/dlv2"));

        ASPMapper.getInstance().registerClass(Edge.class);
        ASPMapper.getInstance().registerClass(Path.class);

        InputProgram input = new ASPInputProgram();

        String rules = "from(" + from + "). to(" + to + ")."
          + "path(X,Y,W) | notPath(X,Y,W) :- from(X), edge(X,Y,W)."
          + "path(X,Y,W) | notPath(X,Y,W) :-
                      path(_,X,_), edge(X,Y,W), not to(X)."
          + "visited(X) :- path(_,X,_)."
          + ":- to(X), not visited(X)."
          + ":~ path(X,Y,W). [W@1 ,X,Y]";

        input.addProgram(rules);
        for(Edge edge : getEdges())
          input.addObjectInput(edge);

        handler.addProgram(input);

        AnswerSets answerSets = (AnswerSets) handler.startSync();

        for(AnswerSet answerSet : answerSets.getOptimalAnswerSets()) {
          // archi nel cammino minimo (non ordinati)
          ArrayList<Path> path = new ArrayList<Path>();
```

```
            int sum = 0; // peso totale del cammino

            for(Object obj : answerSet.getAtoms()) {
              if(obj instanceof Path) {
                path.add((Path)obj);
                sum += ((Path)obj).getWeight();
              }
            }
            join(from,path,sortedPath); // ordina gli archi
            print(sortedPath,sum); // mostra il cammino
          }
        } catch (Exception e) { e.printStackTrace(); }
    }

    private static ArrayList<Edge> getEdges() {
      ArrayList<Edge> edges = new ArrayList<Edge>();

      edges.add(new Edge(0,1,1));
      edges.add(new Edge(0,2,4));
      edges.add(new Edge(1,2,2));
      edges.add(new Edge(1,3,4));
      edges.add(new Edge(1,4,1));
      edges.add(new Edge(2,4,4));
      edges.add(new Edge(3,5,6));
      edges.add(new Edge(3,6,1));
      edges.add(new Edge(4,3,1));
      edges.add(new Edge(6,4,5));
      edges.add(new Edge(6,5,9));
      edges.add(new Edge(6,7,1));
      edges.add(new Edge(7,5,2));

      return edges;
    }

    [...]
}
```

L'applicazione contiene un Handler, inizializzato con un DesktopHandler, che usa come parametro un DLV2DesktopService con una stringa che rappresenta il percorso in cui si trova il solver DLV2. La classe ASPMapper registra le classi Edge e Path precedentemente create, per poter poi effettuare la traduzione delle loro istanze in stringhe, e viceversa. E' stato creato un InputProgram, inizializzato con un ASPInputProgram, e gli sono stati aggiunti una stringa e una lista di Edge, rappresentanti fatti, regole e vincoli del programma ASP. L'InputProgram è stato passato all'Handler, che ha poi avviato l'esecuzione del solver. L'output è stato raccolto nella classe AnswerSets, che poi è stata

esaminata. Sono stati raccolti i Path, rappresentanti gli archi nel cammino minimo, e la somma dei loro pesi. Infine sono stati ordinati e visualizzati.

Il secondo esempio riguarda il problema blocks-world, ed è stato usato il linguaggio PDDL, usando l'implementazione in Python di EmbASP e usando il solver SPD. Blocks-world è un problema in cui dato un insieme di blocchi disposti in una configurazione iniziale, bisogna trovare una serie di passaggi attraverso i quali disporre i blocchi nella configurazione desiderata. In questo esempio, data la configurazione iniziale a sinistra, ovvero 4 blocchi tutti separati tra loro, si vuole portarli nella configurazione finale a destra, ovvero uno sopra l'altro.



Per fare questo sono state create delle classi rappresentanti le azioni che posso essere eseguite.

```
class PickUp(Predicate):
    predicateName="pick-up"

    def __init__(self, block=None):
        super(PickUp, self).__init__([("block")])
        self.block = block

    [...]

class PutDown (Predicate):
    predicateName="put-down"

    def __init__(self, block=None):
        super(PutDown, self).__init__([("block")])
        self.block = block

    [...]

class Stack (Predicate):
    predicateName="stack"

    def __init__(self, block1=None, block2=None):
```

```
        super(Stack, self).__init__([("block1"), ("block2")])
        self.block1 = block1
        self.block2 = block2

    [...]

class Unstack (Predicate):
    predicateName="unstack"

    def __init__(self, block1=None, block2=None):
        super(Unstack, self).__init__([("block1"), ("block2")])
        self.block1 = block1
        self.block2 = block2

    [...]
```

A questo punto, supponendo che siano stati creati due file (domain.pddl e p01.pddl) contenenti le definizioni del dominio e del problema (il cui contenuto è discusso nell'esempio nel capito 2.2), si può iniziare a sviluppare l' applicazione.

```
handler = DesktopHandler(SPDDesktopService())

input_domain = PDDLInputProgram(PDDLProgramType.DOMAIN)
input_problem= PDDLInputProgram(PDDLProgramType.PROBLEM)

input_domain.add_files_path("../domain.pddl")
input_problem.add_files_path("../p01.pddl")

handler.add_program(input_domain)
handler.add_program(input_problem)

PDDLMapper.get_instance().register_class(PickUp)
PDDLMapper.get_instance().register_class(PutDown)
PDDLMapper.get_instance().register_class(Stack)
PDDLMapper.get_instance().register_class(Unstack)

output = handler.start_sync()

for obj in output.get_actions_objects():
    if isinstance(obj, PickUp) | isinstance(obj, PutDown) | isinstance(obj, Stack) |
isinstance(obj, Unstack) :
        print(obj)
```

Il file contiene un Handler, inizializzato con un DesktopHandler usando il parametro SPDDesktopService. Siccome PDDL richiede di definire separatamente il dominio e il problema, sono stati creati due PDDLInputProgram, uno di tipo domain e l'altro di tipo

problem. Ad essi sono stati specificati gli indirizzi dei percorsi dei file precedentemente creati, e poi sono stati passati all'Handler. La classe PDDLMapper registra le classi precedentemente create, per poter poi interagire con l'output del solver. L'handler ha avviato l'esecuzione del solver e l'output è stato ricevuto, esaminato e visualizzato.

5 Portare EmbASP in Produzione

Da fare

6 Conclusioni

Da fare