

Sistemi ad Intelligenza Distribuita un'introduzione

Paolo Montalto

Introduzione.

Prima di tutto bisogna specificare cosa significa, per un sistema artificiale, essere ad intelligenza distribuita. Inoltre cosa significa per un sistema artificiale essere intelligente? Mostrare intelligenza od averla?

Se facciamo le stesse domande per gli umani, come rispondiamo?

Significati di Intelligenza Artificiale.

Riportiamo alcune opinioni di esperti espresse su libri di IA :

- Lo sforzo di far pensare i calcolatori...renderle macchine con una mente (Haugeland 1985)
- Lo studio di facoltà mentali mediante lo studio di modelli computazionali (Charniak McDermott 1985)
- L'arte di creare macchine che realizzano funzioni che richiedono intelligenza quando fatte dagli uomini (Kurzweil 1990)
- Lo studio delle computazioni che rendono possibile percepire, ragionare e agire (Winston 1992)
- un campo di studio che cerca di spiegare e emulare il comportamento intelligente in termini di processi di computazione (Schalkoff 1990)
- Lo studio di come si fa a far fare ai calcolatori cose nelle quali attualmente , le persone riescono meglio (Rich e Knight 1991)
- Quella branca del computer science che si occupa del rendere automatico il comportamento intelligente (Luger e Stubblefield 1993)

Parte delle risposte parlano di mente , pensiero, ragionamento (quindi un sistema ha intelligenza), altre parlano di comportamento intelligente (un sistema mostra intelligenza).

Le risposte degli esperti sembrano proporre 4 goals conflittuali per l'IA:

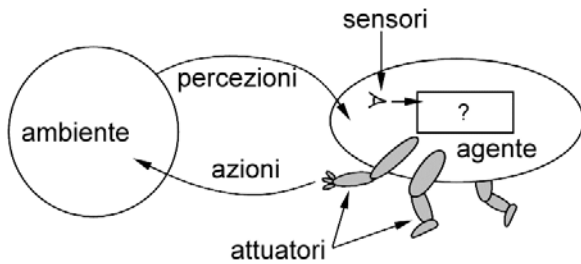
- Occuparsi di Sistemi che pensano come gli esseri umani, di Sistemi che pensano razionalmente, di Sistemi che agiscono come gli esseri umani, di Sistemi che agiscono razionalmente (qui un sistema razionale è quello che fa la cosa "giusta").
- Come vedete o ci si riferisce a esseri umani, le teorie sulla cui intelligenza e performance sono ancora iniziali, oppure ci si riferisce ad un concetto ideale di intelligenza chiamato da Norvig e Russel razionalità'.

Approcci all'Intelligenza Artificiale.

Anche dalla storia della AI emergono i seguenti Approcci:

- Approccio Cognitivo ovvero pensare come gli umani:
 - Andare a studiare la mente ed emularla a certi livelli.
 - Difficoltà :non c'è una teoria precisa della mente .
- Approccio Logico:
 - Sarà affrontato in modo esteso in seguito perché non solo chiarisce punti fondanti come "corrette inferenze e rappresentazione del mondo" dell' IA tradizionale ma rimane anche, per gli stessi motivi, anche nell' IA distribuita.
- Approccio ad Agente Razionale

Approccio ad agente razionale.



Un Agente Intelligente è un sistema che percepisce un ambiente ed agisce “razionalmente”, cioè persegue i suoi goals dato quello che percepisce e le cose che “ crede “.

Agire “razionalmente “ non significa sempre essere guidato da corrette inferenze : si pensi per esempio ad azioni riflesse .

I vantaggi di seguire l’approccio ad Agente Razionale sono che: comprende l’Approccio Logico, mantiene il riferimento all’ uomo e alle sue scienze come fatto di sollecitazione intellettuale , si può rendere abbastanza rigoroso, diventa piu’ propriamente un campo dell’informatica cioè produce tecnologie informatica.

L’ Agente Razionale è un particolare Agente Intelligente, che a sua volta è un particolare Agente dove l’ Agente è la nozione fondante il moderno paradigma di computazione ad Agenti di cui parleremo nella seconda parte del corso.

Autonomia.

È la proprietà principe nelle definizioni di ogni tipo di agente , ed è anche la proprietà che è stata inseguita fin dagli albori dell’IA essendo considerata una delle caratteristiche principali di noi umani.

Sia per gli umani che per gli artefatti, essere Autonomi è di difficile definizione.

Definizioni:

- 1 Un Sistema è tanto più autonomo quanto più il suo comportamento è determinato dalla sua esperienza piuttosto che sulla base di assunzioni built-in.
- 2 Un Sistema è autonomo se non è sotto l’ immediato controllo di un umano .
- 3 L’autonomia è una questione di architettura (cioè come è fatto) e di capacità d’interazione.

Esempi di Agenti non Intelligenti :

- Termostato :
 - Percepisce solo il troppo caldo e il troppo freddo.
 - Azioni :
 - 1 Se troppo caldo: Spegne la caldaia .
 - 2 Se troppo freddo: Accende la caldaia .
- Demone di Sistema Operativo che avvisa se ci sono messaggi non letti:
 - “Percepisce “la posta in arrivo
 - Azioni :
 - 1 Se c’è posta in arrivo avvisa che ci sono messaggi non letti.
 - 2 altrimenti avvisa che tutti i messaggi sono stati letti.

Agente Razionale Ideale.

Definizione: Un Agente (cioè un sistema autonomo in grado di percepire ed agire) è un Agente Razionale Ideale se ad ogni sequenza di percezioni associa , sulla base di quello che ha percepito e di qualsiasi conoscenza abbia, l’azione che massimizza la “ misura di performance” scelta per l’agente.

Essere razionale dipende quindi da:

- la misura di performance scelta, che definisce il grado di successo
- Tutto quello che è stato percepito dall’ agente
- Cosa l’ agente conosce
- Le azioni che un Agente può fare

Ad esempio: supponiamo che un Agente traversi la strada a occhi chiusi e venga travolto da una macchina: poiché la sua sequenza di percezioni non contiene “macchina in transito” è razionale secondo la definizione? A prima vista si, ma dove sono le sottigliezze che fanno rispondere di no?

Sistemi ad Intelligenza Distribuita

I trucchi sono nella opportuna misura di performance e nelle azioni associate con la situazione data. Allora come si definisce la misura di performance?

Procedura per definire la misura di performance.

Noi (in genere umani), come osservatori esterni, stabiliamo uno standard che definisca “avere successo in un certo ambiente” e usiamo questo, insieme alle capacità dei sensori e degli effettori, per definire la misura di performance. (come si può vedere, riemerge l’uomo come giudice dei suoi artefatti)

Nel caso precedente, l’azione di Guardare a sinistra e a destra prima di attraversare, ammesso che l’agente abbia queste capacità sensorie e sia in grado di fare le azioni dette, massimizza la performance nella situazione data. “Essere un agente razionale ideale” risente in ultima analisi, del giudizio degli umani ed è quindi anche sociale, culturale, ...

Vincoli oggettivi posti dalla definizione di Agente.

Esempi:

1. Facciamo un viaggio che cambia il nostro fuso orario : l’orologio non si aggiorna automaticamente: che tipo di agente può essere un comune orologio? E’ razionale?
2. Il clock del mio computer si è aggiornato automaticamente per il ripristino dell’ora solare, può essere pensato come un agente razionale?

Tipologie di Agenti.

L’Intelligenza Artificiale si occupa di definire il software, compresa l’architettura di esso, che realizza l’associazione fra percezioni ed azioni che definisce l’Agente. L’agente comunque si differenzia anche per tutte quelle parti affiancate al programma che permettono ad esso di funzionare e di essere eseguito .

Il computer, l’hardware in grado di processare immagini, filtrare segnali audio etc, software che permette ai dati prodotti dai sensori di essere disponibili per il programma, permettono al programma di essere eseguito e fanno in modo che gli effettori generino le azioni corrispondenti alle scelte risultato del programma.

Tipologie di ambiente.

L’Agente è sempre collocato in un ambiente: origine delle sue percezioni e “ luogo delle sue azioni. L’ambiente influenza la progettazione dell’Agente.

Gli ambienti possono essere classificati in base alle difficoltà che offrono e quindi alla struttura dell’ agente che richiedono, in base al grado di:

- 1 accessibilità
- 2 determinismo
- 3 cambiamento
- 4 episodicità
- 5 separabilità e distinzione delle percezioni e delle azioni

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

Se l’ambiente è “completamente accessibile” l’agente **può non mantenere storia dell’ evoluzione di esso.**

Se è “deterministico”, cioè il suo “prossimo stato è determinato dallo stato corrente e dall’ azione dell’Agente, ed è completamente accessibile **l’Agente può non gestire l’ incertezza.**

Sistemi ad Intelligenza Distribuita

Se è “episodico”, l’esperienza dell’ Agente può essere separata in “episodi” , cioè unità del ciclo percezione-azione indipendenti: Ogni episodio non è correlato al precedente, o al futuro: **Non c’è bisogno di pensare in avanti o indietro.**

Se è statico, l’ambiente non cambia mentre l’ agente sta scegliendo la sua azione (se diminuisce la performance dell’agente l’ ambiente è detto semidinamico), **allora è inutile che l’ Agente interagisca con l’Ambiente, senza aver prima agito in esso, oppure che l’ Agente si preoccupi per il passare del tempo.**

Se è discreto, le percezioni e le azioni sono ben definite (notare che essere discreto o continuo dipende dal livello di granularità scelta , e quindi è in relazione a come è stato definito l’ Agente).

Tipologie di Agenti Razionali Ideali.

Ci sono diverse tipologie di agenti, ma c’è una schematizzazione base che se ne può dare:

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent’s memory of the world

  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```

Ovviamente la funzione Update-Memory avrà complessità dipendente dal bisogno di “analizzare le passate percezioni” dell’ agente.

Passiamo ora ad analizzare le varie tipologie di agente.

Table driven agent

```
function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
         table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Il programma presuppone che siano note tutte le possibili sequenze di percezioni, le quali sono le chiavi per arrivare alle azioni. Questa tipologia di agente ha ovviamente degli svantaggi:

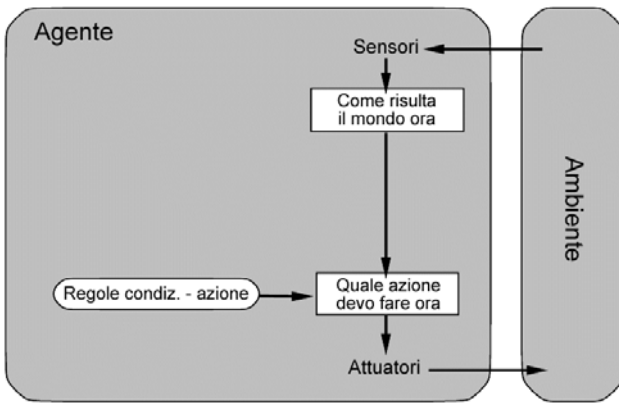
- non è proponibile quando la tavola diventa di dimensioni proibitive(ad esempio negli scacchi: 35^{100}), lunga da costruire per il progettista;
- l’Agente ha tutto built-in, eventuali meccanismi di “learning” richiederebbero tempi molto lunghi per modificare opportunamente tutte le entry.

Per illustrare i vari tipi di agente si può prendere ad esempio il “guidatore di Taxi”:

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

Se ad esempio di pensa al tassista come puramente reattivo bisogna ipotizzare una cosa di questo genere: la macchina davanti frena, cioè le luci relative ai freni si accendono, il tassista comincia a frenare.

Agente Reattivo(Reflex Agent)



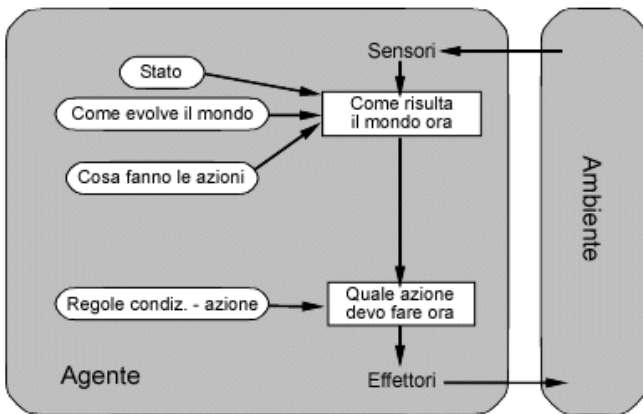
Le connessioni percezione-azione sono espresse mediante regole dette regole condition(o situation)-action. Interpretazione: Se la parte sinistra è soddisfatta dalla stato corrente allora fai l'azione ad essa connessa.

Ovviamente in questo caso lo stato coincide con la percezione corrente.

Esistono situazioni in cui le percezioni devono essere aiutata da un qualche stato interno e da conoscenza su come il mondo evolve e sugli effetti delle azioni: pensare al tassista che cambia corsia.

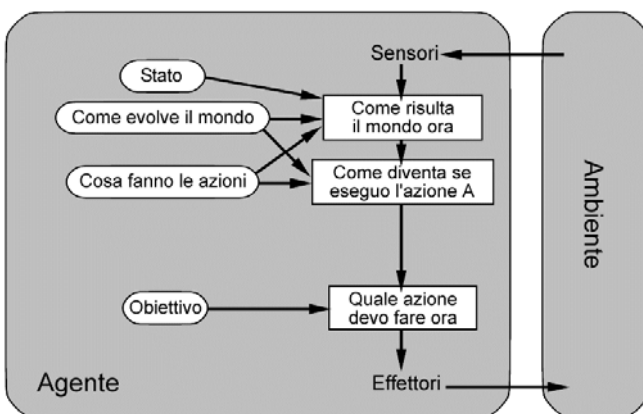
Agente Reattivo con Stato Interno

Non cambia la natura delle regole ma lo stato dell'agente tiene conto dello stato precedente, dell' attuale percezione, di come il mondo evolve indipendentemente dall'agente e da come le proprie azioni influenzano l'evoluzione del mondo.



Agenti che perseguono goals

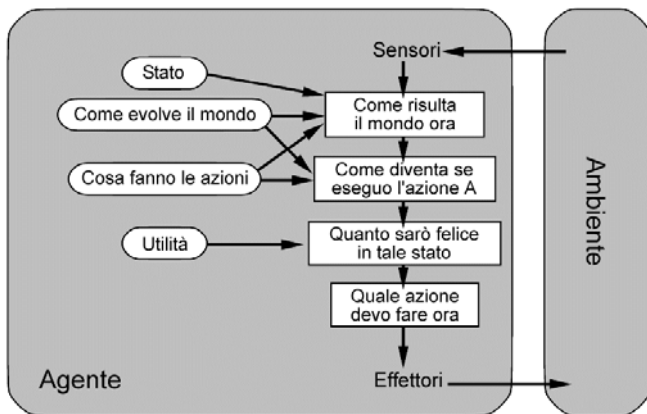
Si può anche pensare che il tassista non voglia solo guidare per la città senza meta, ma che persegua un goal, ad esempio arrivato ad un incrocio dove può girare a sinistra, a destra o continuare dritto (azioni tutte possibili!), la scelta su quale azione intraprendere dipende da dove deve andare, da come è possibile arrivarci in macchina (Search, Planning), da cosa succede se fa una scelta o un'altra etc.



Questi agenti sono caratterizzati da un aumento di flessibilità: gestione dell'imprevisto, della produzione di altri goals, varietà delle risposte etc.

Si può, ancora, pensare che il tassista, oltre ad avere un goal, persegua un qualche suo utile: va bene sia girare a destra che a sinistra, ma girando a sinistra è possibile vedere fino a che ora è aperto un negozio che lo interessa, la strada è meno trafficata ed è meno dissestata etc. (scelta guidata da un qualche criterio di utilità).

Utility-based Agents



L' utilità si traduce, in IA, solitamente, in una funzione: dagli stati (o da sequenze di stati)ai reali, a volte però può essere trasformata in una serie di goals aggiuntivi, in tal caso l' agente è equivalente ad un agente goal-based.

Simulazione dell'ambiente

Un modo di testare un sistema è quello di simulare una classe di ambienti e di farlo interagire con questa simulazione, rispettando la natura dell'agente e facendo evolvere l'ambiente per effetto dell'agente, di altri agenti ed eventualmente da processi non Agenti.

```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
```

```
inputs: state, the initial state of the environment  
UPDATE-FN, function to modify the environment  
agents, a set of agents  
termination, a predicate to test when we are done
```

```
repeat
```

```
  for each agent in agents do
```

```
    PERCEPT[agent] ← GET-PERCEPT(agent, state)
```

```
  end
```

```
  for each agent in agents do
```

```
    ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
```

```
  end
```

```
  state ← UPDATE-FN(actions, agents, state)
```

```
until termination(state)
```

```
function RUN-EVAL-ENVIRONMENT(state, UPDATE-FN, agents,
                             termination, PERFORMANCE-FN) returns scores
  local variables: scores, a vector the same size as agents, all 0

  repeat
    for each agent in agents do
      PERCEPT[agent] ← GET-PERCEPT(agent, state)
    end
    for each agent in agents do
      ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
    end
    state ← UPDATE-FN(actions, agents, state)
    scores ← PERFORMANCE-FN(scores, agents, state)
  until termination(state)
  return scores                                     /* change */
```

Problem-Solving Agents

Sono particolari Agenti, goal-based, il cui compito è risolvere problemi. Rappresentano uno step intermedio rispetto agli Agenti Knowledge-based. Ci permettono di definire un “Problema”, la sua “Soluzione” e la ricerca della soluzione. Formulazione di un problema nel contesto Problem-solving.

Nel contesto dei problem-solving Agents, un problema coincide con la formulazione di un problema, il cui primo passo è individuare gli elementi che ci permettono di formulare un problema e la sua soluzione, e il secondo passo è individuare linguaggi e strutture necessari per rappresentarli.

Un problema viene scomposto nei seguenti elementi:

- Ambiente in cui l’Agente è inserito, in particolare “ambiente percepito”
- “Goal che si vuole raggiungere” cioè “quale ambiente “ soddisfa il goal
- azioni a disposizione dell’ agente
- definizione di soluzione e algoritmi di ricerca

Questo può avvenire solo sotto le seguenti ipotesi:

- L’ ambiente o mondo può essere in ogni istante rappresentato da uno stato.
- Un goal è un insieme di stati del mondo: esattamente quegli stati del mondo in cui il goal è soddisfatto.
- Le azioni possono essere viste come realizzanti transizioni fra stato e stato.

Risolvere un problema significherà trovare sequenze di azioni che se eseguite trasformano lo stato iniziale nello stato finale (problema : rimangono eseguibili una volta pensate?)

Formulazione del problema

Formulare un problema significa decidere gli stati, le azioni da prendere in considerazione, in accordo con la formulazione del goal.

In che cosa differisce e in che cosa è simile all’ usuale processo di costruire un programma?

- Livello del Linguaggio, differenza fra dare un algoritmo (o programma) e dare le primitive di un linguaggio con cui costruire algoritmi.
- Dare algoritmo per un problema ovvero dare un algoritmo per “costruire“ algoritmi per problemi particolari.

Trovare la soluzione è compito del processo di ricerca : Search process.

Una soluzione può assumere diverse forme: sequenze lineari di azioni, albero...

Si possono, in generale, individuare 4 tipologie di problemi: a stato singolo, a stato multiplo, dipendenti da situazioni imprevedibili a priori, problemi tipo “esplorazione in ambiente ignoto senza informazioni sul risultato delle proprie azioni”. Le classi più tradizionali sono le prime due.

Problemi a stato singolo

Sistemi ad Intelligenza Distribuita

I problemi a stato singolo sono caratterizzati dalla presenza di sensori in grado di definire completamente lo stato dell'ambiente (ambiente completamente accessibile) e dalla conoscenza completa sugli effetti delle azioni, quindi l'agente può calcolare esattamente lo stato dell'ambiente dopo che una qualsiasi sequenza di azioni viene eseguita. Più formalmente, un "single state problem" è caratterizzato da:

- Uno stato iniziale;
- Un insieme di Operatori o alternativamente una funzione Successore (un "operatore" è una descrizione di una azione in termine dello stato raggiungibile, eseguendo una specifica azione, a partire da uno specifico stato. Una "funzione Successore", associa ad uno stato l'insieme degli stati raggiungibili da questo, qualora venissero eseguite tutte le azioni possibili a partire da esso);
- Uno "spazio degli stati" relativo al problema, cioè l'insieme di tutti gli stati raggiungibili a partire da un determinato stato iniziale. Ogni sequenza di azioni che eseguite a partire da s conducono ad s' , è detta "Cammino" o "path" congiungente uno stato s e uno stato s' appartenenti allo spazio degli stati;
- Un Goal: è alternativamente uno stato, un insieme di stati, una proprietà definita sugli stati;
- Goal test: il test che un agente può applicare per sapere se uno stato appartiene al goal;
- Una funzione "costo" che assegna un costo ad ogni path;

Una soluzione è un path fra lo stato iniziale e uno stato che soddisfa il goal test.

Problemi a stato multiplo

L'agente ha un accesso limitato allo stato dell'ambiente (ad esempio per sensori insufficienti o ambiente parzialmente accessibile) e conosce esattamente tutti gli effetti delle sue azioni, ma non conoscendo lo stato in cui si trova è costretto a ragionare su insiemi di stati.

Essere a stato singolo o a stato multiplo dipende dall'accessibilità dell'ambiente: è possibile quindi dare una versione single-state e una versione multiple-state di uno stesso problema (ad esempio per l'Agente Aspirapolvere).

La definizione di un "multiple state problem" è simile alla definizione dei problemi a single state, sostituendo insiemi di stati a stato. La funzione costo e il goal test restano inalterati, ci sono poi:

- Un insieme di stati iniziale;
- Operatori che specificano per ogni azione l'insieme degli stati raggiungibili a partire da un determinato stato (un operatore applicato ad un insieme di stati S , da come risultato l'insieme unione degli insiemi ottenuti applicando l'operatore ad ognuno degli stati di S);

Un cammino connette insiemi di stati e una soluzione di un problema è un cammino che porta a un insieme di stati, ognuno dei quali è uno stato goal.

Nel "Problem Solving" la capacità di definire stati ed azioni è un'arte, cioè la capacità di definire cosa includere e cosa escludere nella nostra descrizione degli stati e quali azioni prendere in considerazione e come descriverle: come descrivere l'ambiente in cui sono applicabili e l'ambiente che producono.

La capacità coinvolta è la capacità di astrazione: cioè togliere quei dettagli che non servono, quei dettagli che qualora introdotti identificano una soluzione che è in una corrispondenza biunivoca con la soluzione astratta.

Problemi caratterizzati da "imprevedibilità"

In tali problemi la conoscenza dell'agente è incompleta e scorretta: si ricordi che la conoscenza dell'ambiente da parte dell'agente nel contesto del problem solving è determinata sia da caratteristiche dei sensori dell'agente sia da caratteristiche proprie dell'ambiente in cui l'agente è inserito.

Rispetto ai problemi precedentemente introdotti emergono due nuovi elementi:

- Le "Azioni percettive" sono introdotte durante la fase di esecuzione quindi previste nella soluzione;
- La soluzione è di fatto un'albero, non una sequenza, dove in generale, ogni ramo dell'albero è relativo al verificarsi di qualche imprevisto.

La struttura dell'agente solitamente è tale che l'agente può agire prima di aver trovato l'intera soluzione, cioè, l'agente invece di considerare tutto l'albero delle soluzioni, cioè di pensare risposte a tutte le possibili situazioni che possono avvenire, comincia ad eseguire soluzioni parziali e durante l'esecuzione vede cosa succede per continuare a risolvere il problema con l'informazione acquisita.

Problemi tipo "Esplorazione"

Sistemi ad Intelligenza Distribuita

A differenza che nei precedenti problemi, in questo caso l'agente non conosce tutto l'effetto delle sue azioni, cioè non è a priori noto lo spazio degli stati, (il caso più complesso, previsto da Hewitt, è quello in cui il set delle azioni disponibili si arricchisce man mano che si arriva alla soluzione).

Ricerca della soluzione: problemi a stato singolo e multiplo.

Trovare la soluzione è compito dei processi di ricerca della soluzione. Un algoritmo di ricerca ha in input lo stato iniziale, gli operatori, il goal-test, e un'eventuale funzione costo per cammini nello spazio degli stati. L'output è una soluzione, eventualmente, il path-soluzione di minor costo (ottimale).

Uno degli elementi fondamentali del processo di ricerca della soluzione è la costruzione di un albero di ricerca che, *in ogni passo* del processo, presenti:

- una **radice** corrispondente allo stato iniziale;
- dei **nodi** corrispondenti a stati nello spazio degli stati;
- delle **foglie** corrispondenti a nodi a cui ancora non sono stati applicati operatori, o a nodi a cui sono stati applicati operatori ma hanno dato come risultato l'insieme vuoto;

C'è una sostanziale differenza fra albero di ricerca e spazio degli stati: il primo è formato da "nodi" il secondo da stati, inoltre nel primo più nodi (potenzialmente infiniti) sono in corrispondenza con uno stesso stato dello spazio degli stati.

Dato un problema, i processi di ricerca si differenziano per la strategia che implementano, la quale definisce una modalità di costruzione dell'albero di ricerca: solo l'algoritmo di backtrack definisce una strategia che sviluppa non un albero ma una lista lineare formata da nodi che mantengono traccia degli operatori applicati fino a quel momento.

Informalmente ogni processo di ricerca (ad eccezione del Backtrack) può essere visto essenzialmente come la ripetizione di:

- *se non hai nodi da sviluppare allora return Fallimento;*
- *Scegli il nodo da sviluppare secondo la strategia: se contiene il goal return la soluzione;*
- *espandi il nodo e aggiungi i figli all' albero di ricerca.*

Nell'algoritmo appena esposto sviluppare un nodo significa: applicare tutti gli operatori applicabili (questo meccanismo sviluppa un albero).

Il Goal-Test viene fatto non quando il nodo-goal viene generato ma nel momento in cui esso viene scelto per l'espansione.

Tipicamente la struttura dati per un nodo è definita nel seguente modo:

Un nodo è composto da 5 componenti:

- Lo stato a cui corrisponde
- Il nodo che lo ha generato.
- L'operatore che lo ha generato.
- Il suo depth.
- Il costo del path dalla root ad esso.

Di seguito è riportato lo schema di un generico algoritmo di ricerca:

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Notare che nell'algoritmo si usa il termine "Queue" per sottolineare che, per l'espansione si prende sempre il primo elemento della lista di operatori.

Le caratteristiche di una strategia di ricerca sono essenzialmente le seguenti:

- Completezza: garantisce di trovare una soluzione se ne esiste una (Ricordare che il problema di cui parliamo è la particolare formulazione del problema che abbiamo scelto!);

- Time Complexity: quanto tempo occorre a trovare la soluzione.
- Space Complexity: quanta memoria richiede portare avanti la ricerca;
- Ottimalità: la soluzione trovata è la “ migliore” secondo un qualche criterio? Nelle strategie di ricerca solitamente il criterio è implementato in una funzione “costo”.

Spesso questi requisiti sono conflittuali fra loro (ad esempio Completezza e Complessità, Ottimalità e complessità, etc.) Occorre fare una distinzione fra strategie “senza informazione” ovvero “ strategie cieche”, e strategie “che usano informazione” ovvero Strategie euristiche. Il termine “strategia euristica” attualmente sta ad indicare qualsiasi elemento che migliora il comportamento medio di un sistema di IA, non è detto che lo faccia nei casi peggiori.

Negli algoritmi di ricerca il termine “euristica” significa “ funzione di valutazione”, cioè una funzione che da una stima del “costo di una soluzione”.

Le strategie di ricerca “cieche” sono 6 (Breadth-First, Uniform-cost, Depth-first, Depth-limited, Iterative Deepening, Bidirectional) e differiscono per l’ordine in cui i nodi ancora da espandere vengono espansi, a questo proposito risulta interessante notare che l’unica parte dell’algoritmo dato a dipendere dalla particolare strategia è la funzione Queuing-Fn(...).

Breadth-First: i nodi appena generati vengono aggiunti alla fine della coda preesistente. Questa strategia è completa, poiché se la soluzione esiste, prima o poi viene trovata. Space Complexity e Time complexity sono esponenziali nel fattore di ramificazione dell’albero di ricerca. L’ottimalità non definibile, nel senso che non è definita una funzione costo, quello che sicuramente si può dire è che la soluzione che fornisce questo algoritmo, se esiste, è il cammino più corto dallo stato iniziale al goal.

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Come si può evincere dai dati sovrastanti, strategie di ricerca in cui la Complessità di tempo varia come il fattore di ramificazione dell’albero “b” elevato al depth dell’albero “d” ($T(n)=b^d$) non sono applicabili, per lo meno nell’ambito degli attuali computer e degli attuali modelli di computazione.

Uniform cost Strategy: questa strategia usa una funzione costo $g(n)$ definita su tutti i nodi: i nodi appena generati sono inseriti nella coda in ordine crescente del valore di g . L’implementazione di una strategia del genere, in alcuni casi, può ridurre la complessità (ma non è detto, ad esempio il Breadth-First può essere immaginato come Uniform cost: $g(n)=depth(n)$). E’ ottimale se si suppone che: $g(\text{successore}(n)) \geq g(n)$, tale condizione è rispettata da funzioni $g(n)$ che rappresentino il costo del path dalla radice a n , nell’ipotesi che l’applicazione di un qualsiasi operatore non abbia costo negativo.

Depth-First Strategy: i nodi appena generati vengono inseriti all’inizio della coda, sono cioè i primi che saranno espansi (LIFO: Last In First Out). Questa strategia non è completa (potrebbe continuare ad espandere un ramo infinito) e, se trova una soluzione, non è detto sia la meno costosa (si pensi ad una funzione di costo associata alla profondità (depth) del nodo-goal). La memoria richiesta è modesta: lineare del depth: infatti ogni cammino completamente espanso che non ha portato a soluzioni, viene cancellato. Il tempo di esecuzione è, nel caso peggiore, pari a quello di breadth-first ($T(n)=b^d$).

Depth-limited search: è una variante di depth-first caratterizzata da un limite massimo per il depth, raggiunto il quale elimina il cammino in considerazione e torna ad espandere il primo nodo non espanso. E’ simile in tutto e per tutto a depth-first.

Iterative deepening search: ad ogni stadio è una depth limited search, solo che tenta tutti i possibili valori per il limite di depth. Combina le buone qualità del breadth e del depth first e il fatto di dover espandere più volte gli stessi nodi è solo una piccola correzione all’andamento esponenziale delle espansioni.

Ricerca Bidirezionale: sviluppa contemporaneamente la ricerca a partire dal nodo corrispondente allo stato iniziale e a partire da uno stato in cui il goal-test è soddisfatto. Nel caso di questo secondo tipo di ricerca è necessario conoscere, per ogni nodo, i suoi predecessori, cioè tutti quei nodi applicando ai quali gli operatori, si ottiene il nodo di partenza.

Sistemi ad Intelligenza Distribuita

Si noti che calcolare i predecessori, ammesso che sia possibile, può risultare difficile per alcuni problemi. Inoltre, possono esserci più goals; in questo caso se sono esplicitati come un insieme di stati, il problema diventa un multistate problem, altrimenti bisogna ideare le possibili descrizioni degli stati che potrebbero generare l'insieme dei goals. In ogni caso, bisogna avere un modo efficiente per testare se un nodo generato in una direzione è stato già generato nell'altra, e bisogna decidere per ogni direzione la strategia più adeguata.

Riportiamo di seguito una tabella riepilogativa delle strategie esposte:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Di fatto gli algoritmi fin qui esposti trasformano in albero qualcosa che da un punto di vista degli stati corrispondenti ai nodi è più correttamente un grafo.

Graph – Search

Da scrivere

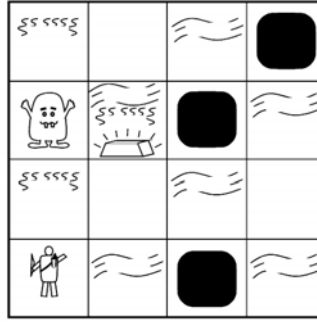
Conoscenza, ragionamento e Rappresentazione

Nel problem solving è data enfasi agli algoritmi di ricerca mentre la parte di rappresentazione è confinata allo stato che qualche modo ha a che fare con l'ambiente percepito, rilevante per il particolare problema, le azioni sono rappresentate come link labellati fra stati e la scelta di quale azione fare è determinata al più da una funzione di valutazione. La formalizzazione scelta per rappresentare l'attività stessa del risolvere il problema, rende non soddisfacenti delle estensioni plausibili dell'agente stesso: Per esempio se chiedessimo al problem solving Agent di spiegarci perché ha dato una certa soluzione o perché, in una certa situazione, ha scelto una certa strada esso ci parlerebbe al più del valore di g . Anche relativamente all'attività di problem solving, l'agente risolvente di problemi non ha per esempio la possibilità di tener conto dell'esperienza acquisita per eventualmente riutilizzarla, oppure la capacità di inferire dalle sue percezioni, proprietà, non ancora percepite dell'ambiente in cui è inserito etc... Se poi si ritorna alla problematica generale dell'Artificial Intelligence e si vuole progettare e realizzare un nostro collaboratore, dotato di autonomia, capace di risolvere più problemi, capace di apprendere etc, allora è chiaro che dobbiamo immaginare estensioni radicali dell'agente risolvente di problemi. Si pensi come esempio ad un agente del tipo appena detto, e supponiamo che esso sia a casa e abbia come goal quello di comprare del latte: è facile pensare che comunque definiamo gli stati, ad essi sono applicabili moltissime delle molteplici azioni che è capace di fare: "andare a dormire", "leggere un libro", "riordinare la casa", "andare al mercato" etc... Cioè "search" è impossibile.

Riportiamo di seguito un semplice esempio di gioco il Wumpus game: uno dei primi giochi per computer basato su un agente che esplora una caverna fatta di stanze connesse da porte: nelle stanze ci possono essere pozzi, in una stanza c'è sicuramente una belva Wumpus e in una stanza ci può essere una massa di oro. Se l'agente cade in un pozzo viene intrappolato e muore, se va nella stanza in cui c'è Wumpus viene divorato. L'agente ha una sola freccia che una volta tirata percorre la direzione di fronte all'agente finché sbatte su una parete o colpisce il Wumpus o viene persa. Se il Wumpus viene colpito emette un urlo che si sente ovunque. L'Agente percepisce solo localmente. Se percepisce un fetore c'è il Wumpus in una delle stanza direttamente adiacenti, se c'è una brezza allora c'è un pozzo in una delle stanze direttamente adiacenti, se è avvertito uno scintillio in una delle stanze direttamente adiacenti c'è l'oro. Il goal è uscire dalla caverna possibilmente con l'oro.

L'agente comincia sempre l'esplorazione dalla stanza in basso a sinistra e le azioni sono: gira di 90 gradi a destra o a sinistra, vai avanti, tira la freccia, risalì la grotta (possibile solo se l'agente si trova all'ingresso).

Domande: Come potremmo trattarlo con il problem solving tradizionale? Se per esempio chiedo all'agente <<dove sei?>>, come può costruirmi la risposta non percependo la sua posizione? Come può (riferimento alla fig. sottostante) capire che il pozzo è esattamente nella casella (1,3)? E così via.



Sistemi di Produzione

Introdotti da Post nel 1943 come meccanismo di computazione del tutto generale (*in ultima analisi come schema di programmazione, modo di programmare, ecc.*) sono stati poi ripresi e modificati in Intelligenza Artificiale.

Tutti i sistemi di produzione hanno in comune la caratteristica che le cose da rappresentare devono essere immerse in uno schema consistente di 3 parti:

1. Base di dati (o Global Data Base o contesto o ambiente)
2. Regole di Produzione
3. Interprete (o sistema di controllo)

Restrizione: L'effetto dell'applicazione di una regola da parte dell' interprete alla BD ha come unico effetto quello di modificare la Base di Dati.

La Base di dati contiene tutta la conoscenza sull'ambiente

La caratteristica importante di un sistema di produzione è che le regole non sono assimilabili a programmi, non si chiamano reciprocamente ma interagiscono solo tramite la base di dati.

Inizialmente i sistemi di produzione sono stati caratterizzati dal loro uso come modi di rappresentazione della conoscenza in 2 campi di ricerca:

1. Modelli di funzionamento psicologico (Newell) cioè come "architettura di sistemi cognitivi".
2. Nella costruzione di sistemi esperti basati sulla conoscenza(es.: MYCIN, DENDRAL, molti altri sistemi esperti ecc.)

Nel primo caso si fa l'ipotesi che molti processi cognitivi siano niente altro che "processare informazioni" e che quindi le regole di produzione offrono un modo chiaro e formale per esprimere atti di elaborazione simbolica di base che possono formare i primitivi del fenomeno cognitivo complesso.

Nel secondo caso, le regole di produzione offrono una rappresentazione della conoscenza che permette un facile accesso che è facilmente modificabile e quindi utile per sistemi designati per realizzare approcci incrementali alla competenza.

Ripetiamo che la struttura del GDB, delle regole e dell'interprete può variare a piacere, con l'unica caratteristica che le regole non interagiscono fra di loro direttamente ma tramite il GDB. Cioè ogni regola applicabile al GDB, qualora sia applicata, eventualmente la modifica. Questa modifica è accessibile a tutte le regole e quindi può rendere applicabile qualche regola che prima non era applicabile. Questa è l'interazione fra regole in un sistema di produzione.

Se interpretiamo le regole come pezzi di codice di un programma questa chiamata indiretta è analoga ad una programmazione che usi unicamente **Side Effect**.

L'effetto primario di questa interazione limitata ed indiretta è la forte modularità dei sistemi di produzione contemporanea alla difficoltà con cui un sistema di produzione si segue l'evoluzione del GDB: infatti l'evoluzione non è data nella conoscenza dichiarativa. I sistemi di produzione quindi sottolineano l'importanza di pezzi di conoscenza indipendenti mentre rendono secondario il flusso del sistema. E' facile vedere che linguaggi procedurali, viceversa, sottolineano il flusso dei sistemi e organizzano il programma intorno ad esso.

Sistemi ad Intelligenza Distribuita

Una seconda conseguenza della forma limitata di comunicazione fra regole, caratteristica dei sistemi di produzione è che è difficile in questa struttura esprimere concetti che richiedono strutture più ampie di una singola regola. Allora laddove l'interesse è sul comportamento globale del sistema (concetti interrelati o principi generali) i sistemi di produzione sono meno trasparenti della formulazione in termini procedurali.

Tutto quello che riguarda la soluzione sta nell'interprete.

Vediamo ora di spiegare con più chiarezza cosa sono la GDB, le regole e l'interprete.

Global Data Base

In generale si può definire come una collezione di simboli. Poiché esso è inteso a rappresentare lo stato del mondo, la sua complessità e la rilevanza delle scelte, sarà in connessione ai problemi che rappresentiamo con esso. A volte sarà strutturato come una rete, un grafo, una stringa lineare, un insieme di relazioni ecc.

Regole

Usualmente le regole sono viste come composte da un lato sinistro e un lato destro:

L H S → R H S (if “condition” then “action”)

Il lato sinistro (nei sistemi di produzione in cui la struttura di controllo (interprete) agisce **Forward** (cosiddetti “data driven”) (intuitivamente significa che i problemi vengono risolti a partire da quello che si sa, in contrapposizione ai sistemi di produzione “goal directed” o **Backward**, in cui i problemi vengono risolti a partire dal risultato che si vuole ottenere)) rappresenta il lato che viene confrontato con la Base di dati (processo di **Pattern Matching**), il lato destro rappresenta l'azione che viene fatta sulla base di dati in corrispondenza all'applicazione della regola (**Processo di modifica della Base di dati**).

Le restrizioni che abbiamo discusso per la struttura generale di un sistema di produzione impongono che:

- La valutazione del LHS avvenga per pure operazioni di match e di rintracciamento.
- Le operazioni di match e di rintracciamento devono solo osservare lo stato del GDB senza modificarla.
- L'operazione di match vista come predicato Match(GDB,LHS) deve dare solo indicazioni di successo o di fallimento. Sono al più permesse operazioni di legame di variabili e segmenti.
- Le azioni che corrispondono all'esecuzione del RHS devono essere concettualmente primitive rispetto al dominio rappresentato.

Queste restrizioni nel rappresentare comportamenti complessi nelle regole hanno due conseguenze:

1. E' possibile per il sistema, in modo omogeneo, “leggere” agli utenti le proprie regole e quindi formulare una qualche spiegazione e giustificazione del proprio comportamento.
2. Può forzare o complicare la struttura di controllo del sistema forzando ad introdurre in questi meccanismi che equivalgono ad effetti combinati dell'uso di più regole. Ciò naturalmente rende più opaco il comportamento dell'intero sistema.

La struttura del GDB in generale è una collezione di simboli. Come questi simboli sono scritti e strutturati influenza direttamente l'algoritmo di pattern matching e le strutture nelle regole. In generale il GDB è inteso a rappresentare lo stato del mondo, ma l'interpretazione e le proprietà di esso dipendono strettamente dai problemi che si stanno rappresentando.

Sistema di controllo (interprete)

In generale il comportamento di un sistema di controllo può essere separato in due fasi:

1. Riconoscimento
2. Azione

La fase di riconoscimento può essere suddivisa in:

1. Scelta

2. Soluzione dei conflitti

Nella scelta una o più o tutte le regole applicabili sono individuate e passate all'algoritmo di soluzione dei conflitti, che sceglie una di esse. Gli algoritmi di scelta possono essere classificati in base ai loro criteri di scanning del GDB:

- alcuni appena trovata una regola in cui LHS è valutata positivamente smettono la ricerca svuotando il compito dell'algoritmo di soluzione dei conflitti
- altri selezionano tutte le regole in cui LHS valuta positivamente e passano questo insieme all'algoritmo di soluzione dei conflitti
- altri ancora, in particolare i sistemi che lavorano in Backward, scelgono tutte le regole applicabili, cioè le regole in cui RHS "dice" (ancora pattern matching) qualcosa circa il goal da realizzare, mandano in esecuzione la prima di queste regole, l'LHS di questa regola diventa il nuovo goal e il processo va in ricorsione su questo. Al termine del processo di ricorsione e su un risultato favorevole alcuni sistemi di controllo rompono il processo, altri lo continuano sulle rimanenti regole scelte in precedenza (il Prolog, visto come sistema di produzione, ha un interprete che può funzionare in entrambi i modi).
- Altri sistemi di controllo possono nella scelta e nella soluzione dei conflitti usare delle metaregole.
- Esempio di Metaregola: Se una regola porta ad un ragionamento circolare non va scelta.

Gli algoritmi di soluzione dei conflitti (qualora non siano fagocitati dagli algoritmi di scelta) possono usare criteri diversi per la scelta di una delle regole dell'insieme ad essi inviato:

1. L'insieme delle regole è ordinato (ordine completo): viene scelta la regola con maggiore (minore) priorità.
2. Gli elementi del GDB sono ordinati: viene scelta la regola in cui LHS ha il match favorevole con gli elementi nel GDB a più alta priorità.
3. La regola più specifica (cioè il match non ha richiesto sostituzioni complicate) viene scelta.
4. Esiste una struttura gerarchica (reti delle regole): si segue la gerarchia.
5. Viene scelta la regola usata più di recente o la regola il cui LHS ha esito favorevole con l'elemento del GDB usato per il match più di recente (mantengo fisso l'elemento del GDB).
6. Una qualsiasi delle regole applicabili viene scelta.

Qualcosa a proposito dell'**euristica** - Definire precisamente cosa si intende per euristica in AI è molto complicato nel senso che si potrebbe includere in essa tutte le parti che costituiscono un sistema di produzione. Infatti informalmente, per conoscenza euristica s'intende quello che si sa su una determinata classe di problemi e quello che può influenzare il sistema di controllo. Poiché la classe dei problemi può essere arbitrario e poiché la *conoscenza* penetra nella definizione di ogni costituente di un sistema di produzione, si capisce la difficoltà di caratterizzare il termine euristica.

Parliamo quindi di *ricerca euristica* più che di euristica. Per ricerca euristica s'intende essenzialmente guidare(e/o definire) il sistema di controllo mediante una qualche conoscenza.

Non è che in questo modo le cose siano definitivamente più chiare, ma perlomeno sappiamo dove andare a guardare e possiamo considerare ogni realizzazione di un sistema di controllo come "dicente" qualcosa sull'euristica. Allora, per esempio, un sistema di controllo breadth-first, usa una conoscenza euristica del tipo: se sistematicamente provi tutte le possibilità caratterizzate dalla stessa complessità e vai per complessità crescenti e se esiste una soluzione, allora troverai la soluzione avente complessità minima. Dove la "complessità" è la lunghezza del cammino fra il GDB iniziale e il GDB preso in considerazione.

Il discorso sulla ricerca euristica diventa quindi di nuovo un discorso sui sistemi di controllo ovvero un discorso sulle modalità dell'evoluzione di un sistema di produzione. Parleremo di euristica specifica quando l'evoluzione è guidata da conoscenza specifica del problema particolare, parleremo di euristica generale man mano che la conoscenza utilizzata per guidare l'evoluzione di un sistema di produzione si allontana dal problema specifico.

I modi principali in cui l'euristica "specificata" viene rappresentata sono:

1. Mediante una funzione definita da GDB $\rightarrow N$
2. Realizzando funzioni del tipo $f: (GDB, R) \rightarrow (GDB, R')$ con $R' \subseteq R$ e $R =$ insieme delle regole applicabili.
3. Strategie di ricerca.
4. Analisi delle relazioni fra lo stato attuale del processo di evoluzione e lo stato desiderato (in genere "goal")

Vedendo i modi in cui viene realizzata la ricerca euristica potremo dire che per euristica s'intendono le tecniche che migliorano l'efficienza di un processo di ricerca (evoluzione del sistema di produzione) sacrificando eventualmente esigenze di "completezza" cioè esigenze di trovare in generale evoluzioni caratterizzate da certe proprietà (es: soluzione di problemi).

Agenti basati sulla conoscenza

Ogni sistema di A.I. si riferisce nel suo comportamento ad un insieme di "conoscenze".

Storicamente (in A.I.) si è pensato che un sistema software possieda conoscenza se:

- a) La possiede implicitamente: Cioè la conoscenza è diffusa nelle procedure che la usano, la conoscenza necessaria per una certa attività è nella procedura che realizza quell'attività.
- b) La conoscenza si manifesta nell'esecuzione del programma che la definisce ed essa, o parti di essa, sono difficilmente estraibili, modificabili, utilizzabili in altri contesti, da altri programmi. (Brooks è una versione moderna e connessa alla robotica di questa interpretazione).

La conoscenza è esplicitamente contenuta, cioè è una parte a se stante nel programma che realizza il sistema. Tale conoscenza è inoltre detta dichiarativa se è espressa attraverso "sentenze" in un qualche linguaggio, le quali sono interpretate come dicenti qualcosa sul dominio di interesse di un particolare sistema. Inoltre, la conoscenza dichiarativa si presta di più ad essere modificata, interrogata, è resa disponibile a più usi, in particolare a usi non previsti quando è stata definita, a usi di indagine su se stessa etc. Al contrario la conoscenza procedurale è utilizzabile più efficientemente.

Lo schema generale di un agente Knowledge Based è il seguente:

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
         t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Inizialmente la conoscenza era vista essenzialmente come insieme di dati (cioè elementi passivi), poi si è visto che in realtà non era così, poiché della conoscenza fanno parte tutta una serie di nozioni ed euristiche inerenti la soluzione di un problema, in ogni caso risulta molto efficiente rappresentare la conoscenza mediante il calcolo dei predicati.

Ci sono due modi di utilizzare il calcolo dei predicati nella rappresentazione della conoscenza:

- 1) Come linguaggio: per rappresentare la conoscenza;
- 2) Come linguaggio + meccanismo inferenziale: per rappresentare la conoscenza ed il suo uso.

A questo punto si pongono le seguenti due domande:

- a) Predicati, funzioni, individui, connettivi logici primitivi sono in grado di rappresentare tutto quello che utilizziamo nel linguaggio naturale?
- b) La struttura di una frase ben formata nel calcolo dei predicati rende conto della complessità della frase in linguaggio naturale?

Bibliografia

[1] Russel, Norvig: "Intelligenza Artificiale: un approccio moderno", Prentice Hall, 1998.

[2] Appunti del corso di Sistemi ad Intelligenza distribuita 01-02 della Prof.ssa Eliana Minicozzi, Università degli studi di Napoli Federico II.