Università degli Studi di Bari Aldo Moro
A.A. 2012-2013

Ingegneria della Conoscenza e Sistemi Esperti

TURTLE

Una expert system shell ispirata a CLIPS

Claudio Greco	Daniele Negro	Marco Di Pietro
#545060	#528038	#477442

4 settembre 2014

Sommario

La presente documentazione descrive TURTLE, un tool per lo sviluppo di sistemi basati su conoscenza, come progetto d'esame per il corso di Ingegneria della Conoscenza e Sistemi Esperti. Questo documento illustra le caratteristiche del sistema, gli aspetti implementativi, nonché le scelte progettuali adottate al fine di ottenere un software a scopo didattico che realizzi un compromesso tra codice sorgente scalabile e riduzione della complessità legata al modello computazionale basato su regole di produzione, oltre che al linguaggio interpretato Python. Quest'ultimo obiettivo è anche la motivazione per cui il sistema prende il nome di TURTLE: un modo per scongiurare ogni preoccupazione legata alle performance in termini di velocità. Essendo la gestione di un sistema a produzioni un problema non banale, l'intento è stato quello di rendere meno degradanti possibile le prestazioni.

Indice

Capitolo 1

Introduzione

La conoscenza si colloca nella società come elemento fondamentale per un'economia avanzata. Il *knowledge management* è una disciplina manageriale che si propone di studiarla e gestirla sotto gli aspetti culturali, organizzativi e tecnologici al fine di controllarla, diffonderla e replicarla.

Nella maggior parte dei casi le perle di saggezza emergono dalla conoscenza tacita, legata all'esperienza umana che risiede esclusivamente in ogni essere. L'*expertise* rappresenta un vantaggio competitivo poiché permette, in pochi passi, di ottimizzare e migliorare processi.

L'acquisizione di nozioni derivate dall'esperienza è un notevole ostacolo per il knowledge management, motivo per cui l'ICT risulta essere un valido alleato per fornire i giusti strumenti atti alla gestione di tale sapere. In particolare, l'*ingegneria della conoscenza* applica metodologie e formalismi per la progettazione di *sistemi basati su conoscenza* allo scopo di acquisire, rappresentare, gestire, distribuire ed infine manutenere la conoscenza, con particolare attenzione e dedizione all'elaborazione dell'esperienza.

> *You may have a Ph.D. in Computer Science, you may be a wiz programmer, but you couldn't do their job unless you underwent the right training and somehow acquired their troubleshooting experience*[1].
>
> Peter Jackson

[1] Introduction to Expert Systems (1998), cap. 1, pag. 1

La caratteristica fondamentale dei sistemi basati su conoscenza è quella di attuare una procedura d'*inferenza* che, facendo uso della conoscenza codificata posseduta, renda possibile esibire un comportamento intelligente nel raggiungimento di un obiettivo in un determinato dominio di riferimento, utilizzando strategie di *Problem Solving*. Tale intelligenza viene enfatizzata dal modo in cui tali sistemi si comportano in situazioni di incertezza e di parziale conoscenza del problema.

1.1 Il problema della ricerca

Un qualsiasi problema, per essere risolto, necessita di una o più soluzioni possibili se ne esistono. Pertanto, un generico problema può essere formulato nei seguenti termini:

- Uno **stato di partenza**.
- Un **insieme di operazioni** che si possono applicare allo stato corrente per far evolvere (o regredire) un problema allo stato successivo (o precedente).
- Uno o più **stati finali** che rappresentano lo **spazio delle soluzioni** nonché gli obiettivi del problema se soddisfano un determinato *test di terminazione*.

La rappresentazione dello spazio degli stati di un problema è un grafo connesso. Un semplice e primordiale algoritmo *trial and error* potrebbe portare alla soluzione del problema. Tuttavia bisogna considerare due aspetti fondamentali nella ricerca di una soluzione:

- Lo spazio di ricerca può essere finito ma ampio, oppure infinito.
- Il passaggio da uno stato all'altro potrebbe presentare la ripetizione di stati già visitati (loop).

Si possono adottare delle strategie alternative come la ricerca *depth-first* e quella *breadth-first*.

Entrambe le ricerche però potrebbero portare a computazioni decisamente complesse in spazi di ricerca molto grandi essendo esaustive. Inoltre, la depth-first search potrebbe raggiungere presto un obiettivo se fosse guidata da un qualche tipo di euristica ma potrebbe anche non terminare se lo spazio di ricerca fosse infinito, anche nel caso in cui vi sia una soluzione.

Si evince, quindi, che il numero di stati generati in ogni fase può crescere esponenzialmente, causando il cosiddetto fenomeno denominato *esplosione combinatoria*. Per cui, tentare ad ogni step una ricerca per tutti i possibili stati di un problema corrisponderebbe ad un calcolo non indifferente equivalente a quello dato da un algoritmo di *"brute force"*.

I problemi che richiedono un tempo che cresce esponenzialmente sono intrattabili e spesso rientrano nella classe dei problemi *NP-completi*. Perciò, se un programma dovesse enumerare tutte le possibili mosse per il gioco degli scacchi, è evidente che non seguirebbe la strategia ideale per giocare una partita. Piuttosto, se un programma imitasse i grandi giocatori di scacchi, esibendo skill acquisite per selezionare strategie opportune e mosse vincenti, esso mostrerebbe un comportamento "intelligente".

Per ottenere una ricerca più efficiente, è stata introdotta la cosiddetta **ricerca euristica** o ricerca informata, che utilizza della conoscenza, tipicamente rappresentata da una funzione di valutazione che stima la distanza dall'obiettivo, per attraversare il grafo dello spazio degli stati del problema e, ad ogni passo, scegliere di procedere verso uno stato più vicino al goal da raggiungere.

Un'euristica può essere pensata come una regola empirica: a seconda dell'algoritmo o del processo di decisione scelto, il successo non è necessariamente garantito ma nella maggior parte dei casi la risoluzione del problema è facilitata. Sebbene la ricerca guidata da una funzione di valutazione euristica riduca notevolmente lo spazio di ricerca e permetta di trovare una soluzione migliore, talvolta questo approccio si rivela inadeguato per determinate applicazioni che richiederebbero comunque un tempo non ragionevole per giungere ad una soluzione (ad esempio nei task di pianificazione) oltre che costi elevati in termini di risorse.

Per affrontare le difficoltà di gestione della conoscenza descritte, si è cercato un approccio basato sull'utilizzo di regole di produzione, giungendo alla realizzazione di *sistemi esperti*, capaci di rappresentare esplicitamente in dettaglio sia la conoscenza di un dominio posseduta da esperti, sia le strategie che essi utilizzano per ragionare sulla loro conoscenza (know-how).

I sistemi esperti fanno ampio uso della cosiddetta **programmazione euristica**, la quale, in contrapposizione alla programmazione algoritmica (che si basa su procedure matematicamente dimostrabili), permette di definire programmi che risolvono problemi senza un algoritmo definito a priori, determinando al momento, passo per passo, la sequenza di operazioni da eseguire.

1.2 Rappresentare la conoscenza

La rappresentazione della conoscenza è legata al processo attraverso il quale l'informazione può essere memorizzata ed associata nel cervello umano dal punto di vista della logica. Nell'ambito dei sistemi esperti, rappresentare la conoscenza significa descrivere in modo formale una mole consistente di informazioni affinché siano *machine readable*. Descrivere formalmente l'informazione significa convertirla in un linguaggio non ambiguo che abbia una *sintassi* ed una *semantica*, allo scopo di governare forme di espressioni diverse, ciascuna con un significato ben definito. Quando la conoscenza è opportunamente codificata, può essere trattata con elaborazioni in cui simboli e strutture di simboli vengono utilizzate per rappresentare concetti e relazioni. Un esempio per rappresentare in forma standardizzata la conoscenza è la situazione in cui si hanno frasi aventi sintassi diverse ma stesso significato:

```
Mario è il padre di Gianni.

Il padre di Gianni è Mario.

Gianni ha come padre Mario.
```

CONDIZIONE ——→ AZIONE

Figura 1.1: Regola di produzione nella forma antecedente ⟶ conseguente.

Le frasi suddette hanno lo stesso significato, di conseguenza dovrebbero essere codificate in un unico modo. Un mapping in un'unica espressione per ogni frase potrebbe essere:

```
padre(Mario, Gianni).
```

Come si può notare, il nome della funzione rappresenta la relazione (padre-figlio) che lega i due oggetti tra le parentesi, il primo oggetto è il padre ed il secondo il figlio. Questo linguaggio di rappresentazione è soltanto un esempio; esistono diversi formalismi per codificare la conoscenza. Il formalismo preso in considerazione per lo sviluppo di TURTLE è quello delle *production rules* (regole di produzione) poiché costituisce un modello di calcolo molto diffuso nell'ambito dello sviluppo di sistemi esperti. Esso permette la rappresentazione dei più disparati domini, ha un forte potere espressivo per le euristiche, risulta essere molto flessibile nella definizione della notazione adottata per codificare l'esperienza e permette di *guidare il ragionamento* tramite la codifica di strategie di controllo.

1.3 Il modello computazionale

> *It is probably an axiom of artificial intelligence, and modern psychology, that intelligent behavior is rule-governed*[2].
>
> Peter Jackson

Le *production rules* sono un formalismo utilizzato in diversi contesti tra cui i sistemi esperti (Buchanan and Feigenbaum, 1978). In questo ambito vengono spesso chiamate *regole condizione-azione* perché codificano condizioni tra pattern di dati e azioni da eseguire nel caso in cui tali premesse siano soddisfatte.

[2]Introduction to Expert Systems (1998), cap. 5, pag. 76

Le regole di produzione sono state introdotte da Post nel 1943 con la definizione di ciò che venne chiamato *canonical system*, un sistema formale per la manipolazione di simboli. Un sistema a produzioni è composto dalle seguenti componenti:

- Un ***insieme di regole di produzione*** (altresì detto *production memory*).
- Un ***interprete di regole*** che decide quando e quali regole possono essere applicate.
- Un ***database globale*** (o *working memory*) che contiene la rappresentazione degli stati del problema, iniziali, intermedi ed obiettivo. Ogni elemento della working memory, di solito, prende il nome di ***fatto***. Una definizione più formale di cosa sia un fatto sarà illustrata a breve.

Nello scenario del funzionamento di un sistema a produzioni, la working memory viene esaminata e modificata dalle regole di produzione. Quest'ultime si attivano quando si verificano condizioni legate alla presenza di determinati fatti all'interno del database globale che soddisfino i pattern (dei fatti) presenti nella parte sinistra (LHS[3]) di una o più regole. L'interprete delle regole ad ogni ciclo verifica quali di esse sono attivate e ne seleziona una. Questo ciclo macchina prende il nome di **recognize-act cycle**. Quando una regola è attivata, una o più azioni, presenti nella parte destra della regola (RHS [4]), vengono eseguite. Le azioni possono apportare un cambiamento allo stato del problema consistente in modifiche ai fatti della working memory (aggiunte, aggiornamenti e rimozioni di fatti). Schematicamente, le regole in un sistema a produzioni sono nella forma:

$$C_1, ..., C_m \rightarrow A_1, ..., A_n$$

che si legge:

se le *condizioni* C_1 e ... e C_m sono vere,

[3]Left hand side.
[4]Right hand side.

allora esegui le azioni $A_1, ..., A_n$.

Come già accennato, l'*interprete di regole* esegue delle operazioni non banali ad ogni ciclo *recognize-act* (riconosci-agisci):

1. ***Matching***: consiste nella verifica delle condizioni nella parte sinistra di una regola, per ogni regola nell'insieme delle regole, rispetto ai fatti contenuti nella working memory. Al termine di questa verifica complessa, un sottoinsieme di regole attivabili viene restituito ed utilizzato nella fase successiva.

2. ***Risoluzione dei conflitti***: se è possibile attivare più di una regola, allora un algoritmo (*strategia di risoluzione dei conflitti*) seleziona la regola da applicare tra quelle presenti nel *conflict set*.

3. ***Applicazione di una regola***: eseguire le azioni nella parte destra della regola selezionata, quindi si ritorna al passo 1. Le azioni possono comprendere manipolazioni della working memory.

La fase di *matching* impone una notevole sequenza di controlli tra l'insieme delle regole ed il database globale essendo sostanzialmente un prodotto cartesiano tra due insiemi. Un esempio di matching semplice ma inefficiente, per una singola regola, sarebbe:

```
Considera la regola r_i
   ∀ pattern p_j in r_i
      ∀ fatto f_k nel db globale
         m ← match(f_k, p_j)
         se m è false allora restituisci false
restituisci true
```

Dall'esempio si evince che, all'aumentare della dimensione degli insiemi di fatti e regole il problema diventa intrattabile utilizzando scansioni lineari per ogni ciclo recognize-act. Inoltre, la funzione `match` specificata esegue ulteriori controlli componente a componente per pattern, per cui si nota come

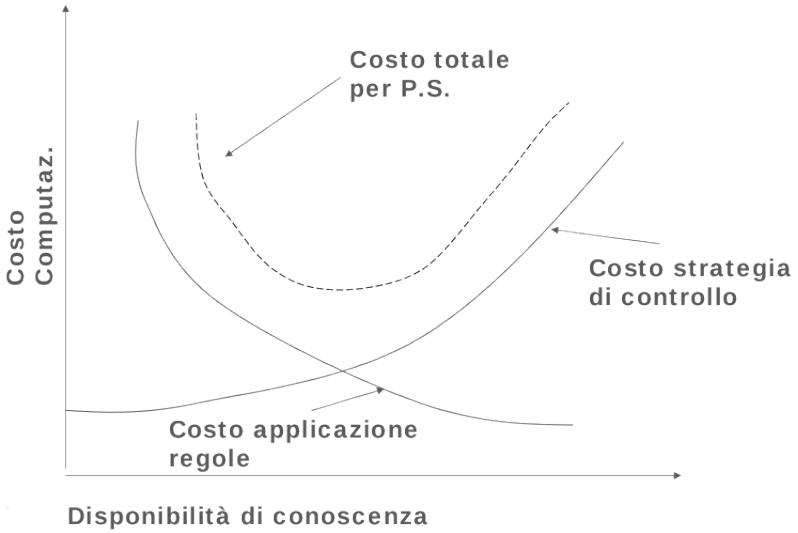

Figura 1.2: Costo di un production system.

il processo di *matching* non sia banale.
Per quel che riguarda la risoluzione dei conflitti, è evidente che, se si definisse un insieme di regole, in cui, per ogni ciclo ne sia applicabile una ed una sola, allora si avrebbe un flusso deterministico paragonabile alla programmazione imperativa. Ciò che invece risulta essere tipico dei sistemi a regole è proprio il non-determinismo. Quest'ultimo è ancor più presente quando i pattern non sono composti solo da costanti ma anche da variabili. Tuttavia, affidarsi al totale non-determinismo è spesso controproducente in termini di efficienza nella ricerca di soluzioni per problemi di una certa consistenza (pur disponendo di un ottimo algoritmo di matching), motivo per cui, oltre alla risoluzione dei conflitti fornita di default da un interprete di regole, è possibile definire strategie di controllo del ragionamento attraverso *meta-regole*, le quali modellano in modo più dettagliato relazioni ed oggetti di dominio. Da ciò si deduce che, una rappresentazione dettagliata e approfondita della conoscenza implica un costo più basso di attivazione delle regole. La figura 1.2 esplica esaustivamente il concetto [5].

Nello sviluppo del sistema TURTLE si è considerato come software di

[5]Principles of Artificial Intelligence (Nilsson, 1980).

riferimento il tool CLIPS (derivante dalla famiglia `OPS5`)[6]. Questo software fornisce diversi approcci per la rappresentazione della conoscenza per realizzare un sistema esperto. CLIPS è un sistema ***forward chaining*** (concatenazione in avanti) in cui il match avviene tra le parti sinistre delle regole e la working memory mentre le azioni sono specificate nelle parti destre. Di conseguenza, anche TURTLE lavora concatenando in avanti. In contrapposizione al ***backward chaining***, che ha un approccio *top-down*, il forward chaining è associato ad un approccio *bottom-up*. Sebbene la concatenazione di simboli all'indietro sia indispensabile per problemi top-down (partendo da possibili goal), nulla vieta di risolvere il problema implementando il ragionamento all'indietro tramite forward chaining.

Essendo CLIPS un sistema molto vasto, in questo documento tratteremo solo le caratteristiche da cui si è attinto per realizzare TURTLE. Gli elementi fondamentali di entrambi i sistemi, ed in generale di tutti i sistemi a regole di produzione, sono i **fatti**, le **regole** ed il **meccanismo di matching**. Per quel che concerne i fatti, la struttura del fatto preso in considerazione è l'**ordered fact** (fatto ordinato), definito formalmente come una ennupla $(\mu, \sigma_1, ..., \sigma_n)$, dove μ è il nome del fatto mentre ciascun σ_i rappresenta un valore indicizzato per posizione.

```
(padre Mario Gianni)
(padre Antonio Nicola)
```

Figura 1.3: Una lista di fatti ordinati in CLIPS compatibile con TURTLE. Relazione padre-figlio.

1.4 I sistemi esperti

Un sistema esperto è un programma, basato su conoscenza, che tenta di riprodurre il comportamento di un esperto umano in uno specifico dominio. In particolare fornisce le risposte o i consigli che fornirebbe l'esperto umano ed è in grado di giustificare la propria risposta. I sistemi esperti consentono

[6]C Language Production system - `http://clipsrules.sourceforge.net/`.

```
(defrule <nome-regola>
    (condizione_1)
    ...
    (condizione_m)
    =>
    (azione_1)
    ...
    (azione_n)
)
```

Figura 1.4: Struttura della definizione di una regola in CLIPS compatibile con TURTLE. Si evince la sintassi LISP-like.

di risolvere problemi, la cui soluzione richiede una considerevole esperienza umana, ragionando euristicamente su una rappresentazione parziale della realtà del problema.

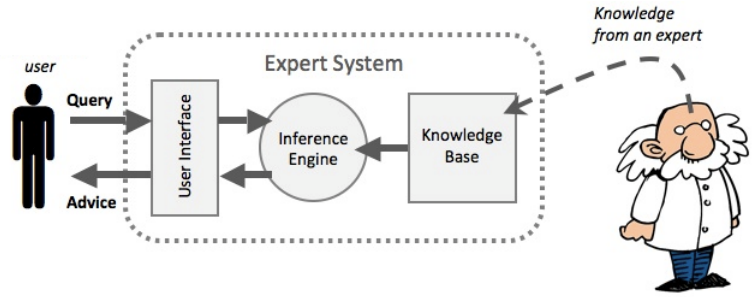

Figura 1.5: Come funziona un sistema esperto.

Un sistema esperto è costituito da un'interfaccia, che ha il compito di rendere la comunicazione tra utente e sistema più naturale possibile, da una *base di conoscenza*, che memorizza la conoscenza estratta dall'esperto (tale conoscenza consiste, essenzialmente, di fatti e di regole) e da un *motore inferenziale*, parte attiva del sistema, che utilizza la base di conoscenza per inferire nuovi fatti e produrre soluzioni, ovvero simula il processo di ragionamento tramite il quale è possibile trarre delle deduzioni logiche partendo dall'esperienza presente nella base di conoscenza, ampliandola.

Un sistema esperto, inoltre, incapsula un modulo per la spiegazione della soluzione che ***fornisce indicazioni sulle modalità di ragionamento*** e di ***giustificazione delle scelte intraprese*** per giungere ad una soluzione.

Capitolo 2

TURTLE

TURTLE è un tool per lo sviluppo di sistemi basati sulla conoscenza codificata mediante fatti e regole di produzione. Il sistema si ispira a CLIPS per ciò che concerne la sintassi dei programmi accettati, la modalità di interazione con l'utente e cerca di replicare gli aspetti peculiari dell'algoritmo di matching RETE. L'attenzione di questo capitolo è rivolta alla descrizione del parser, della rappresentazione dei fatti e delle regole, ad una descrizione dell'algoritmo di matching, nonché della procedura di risoluzione dei conflitti ed infine, della shell d'interazione.

2.1 Il parser

Il parser è il modulo di TURTLE che si occupa del caricamento e della verifica sintattica di un file di testo contenente i costrutti accettati per la rappresentazione di fatti, regole e variabili globali. La grammatica di riferimento utilizzata è un sottoinsieme della grammatica di CLIPS, la cui versione versione integrale è riportata in forma BNF sul suo manuale di riferimento[1].

L'implementazione del parser del sistema sfrutta i servizi offerti dalla libreria **pyparsing**[2], la quale permette la costruzione di una grammatica BNF estesa mediante la composizione di particolari classi, ciascuna delle quali rappresenta uno specifico costrutto del linguaggio. La classe im-

[1] `http://clipsrules.sourceforge.net/documentation/v630/bpg.pdf`
[2] `http://pyparsing.wikispaces.com/`

plementata per il parsing costruisce delle liste di coppie ordinate del tipo `(nome costrutto, AST)`, dove AST è un *Abstract Syntax Tree* tipizzato costituito da liste annidate contenenti i costrutti ed i termini riconosciuti per quel particolare costrutto. È possibile, inoltre, inserire commenti di riga tra un costrutto e l'altro all'interno del file specificando come primo carattere il simbolo `";"`, oppure commentare uno o più costrutti utilizzando le sequenze di simboli `"/*"` e `"*/"` rispettivamente per iniziare e terminare un commento multiriga. Nell'appendice A di questo documento è presente la grammatica BNF estesa implementata dal sistema.

2.2 Rappresentazione dei fatti

La scelta progettuale per la rappresentazione dei fatti è ricaduta sui fatti ordinati, costituiti da un nome e da una sequenza di valori. L'ordine dei valori è determinante ai fini del riconoscimento. Per garantire la futura espandibilità del sistema è stata implementata una interfaccia *Fact* dalla quale deriva la classe specifica *OrderedFact*, che rappresenta un fatto ordinato. Si potrebbero in questo modo supportare i fatti non ordinati in future versioni del sistema. La classe *OrderedFact* comprende il nome del fatto ed una lista di valori associati. La classe *Builder* si occupa della lettura dei fatti ottenuti dal parser e dell'inserimento, previa valutazione di eventuali variabili globali ed espressioni costanti, nella lista dei fatti individuati, i quali verranno interpretati dalla classe *Network* ai fini della fase di **match** prima dell'inserimento nella *Working Memory*.

La *Working Memory* è stata realizzata utilizzando due dizionari: il primo associa ai fatti un valore booleano, che indica se un fatto con tale nome è già stato inserito, e il secondo indicizza i WME a partire dal loro identificativo numerico univoco, in modo da permetterne un facile ritrovamento ed eliminazione. L'inserimento di un fatto nella *Working Memory* determina la costruzione del WME corrispondente, che verrà restituito dopo tale aggiunta. Non si permette l'inserimento di fatti aventi lo stesso nome. L'ausilio dei due dizionari permette l'inserimento, il ritrovamento e la rimozione dei fatti in un tempo costante.

2.3 Rappresentazione delle regole

Per la rappresentazione delle regole si è deciso di implementare una classe *Rule* costituita dal nome della regola, da una lista contenente la parte sinistra, da una lista contenente la parte destra e da un valore numerico di priorità identificato da *salience*. La classe *Builder* si occupa della lettura delle regole ottenute dal parser e dell'inserimento, previa valutazione di eventuali variabili globali ed espressioni costanti, nella lista delle regole individuate, le quali verranno interpretate dalla classe *Network* ai fini della fase di **build** prima dell'inserimento nella *Production Memory*.

La *Production Memory* è stata realizzata utilizzando un dizionario indicizzato rispetto al nome della regola. In caso di inserimento di una regola avente lo stesso nome di un'altra regola già presente, si sostituisce tale regola con quella che si intende aggiungere. L'ausilio del dizionario permette il ritrovamento di una regola e la verifica di duplicati in un tempo costante.

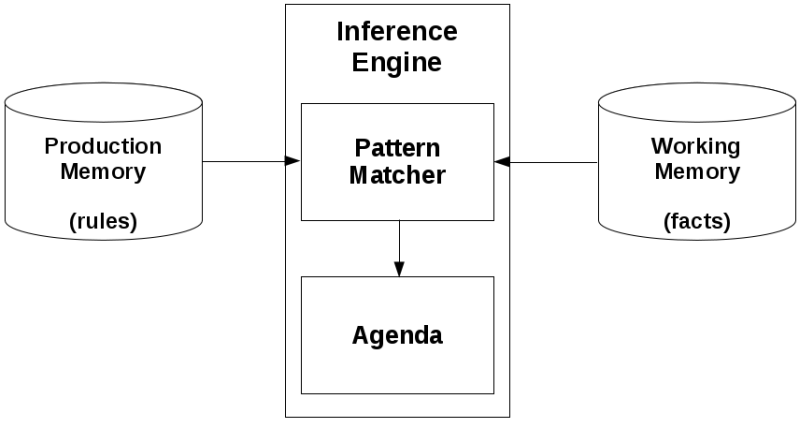

Figura 2.1: Architettura di un inference engine.

2.4 L'algoritmo di matching

Il modulo di matching del sistema si basa sull'implementazione dell'algoritmo RETE. Per la sua efficienza, rappresenta un punto d'inizio per ricerche, miglioramenti e discussioni nell'ambito dello sviluppo di un motore inferen-

ziale. Il concetto alla base di RETE è quello della scomposizione atomica di tutte le parti sinistre delle regole e della costruzione di un grafo per la verifica di un singolo fatto in una prima fase e successivamente di più fatti congiunti da uno o più predicati. I punti di forza di questo algoritmo sono:

- L'**utilizzo di memorie** (alfa e beta) contenenti riscontri parziali, le quali permettono di ridurre il numero di controlli effettuando un match solo su un nuovo fatto asserito, senza verificare nuovamente i fatti già presenti nella working memory.
- La **condivisione dei nodi** tra regole di produzione che hanno stesse condizioni in comune sia nell'alfa network che nella beta. In TURTLE, attualmente, la condivisione dei nodi avviene solamente nell'alfa network; ciò comporta una duplicazione di nodi nella beta network.

RETE presenta anche alcuni svantaggi che naturalmente coinvolgono anche l'algoritmo di matching realizzato per TURTLE:

- Si verificano **sprechi di memoria** dovuti alla duplicazione di riferimenti ai token (nonché ai wme) nelle beta memory.
- L'**eliminazione** di uno o più fatti rappresenta un'operazione non banale all'interno del grafo poiché andrebbero rimossi tutti i riferimenti ed i token generati in cui tali fatti compaiono (operazione denominata retract).
- Spesso si verificano **propagazioni nulle** tra i nodi beta che non avranno mai un'attivazione destra (o solo pochi di essi andranno avanti). Tale problema viene definito "*utility problem*"[Min90].
- Le **performance** possono diventare drastiche se le condizioni nelle regole sono definite in un ordine sparso, con test ridondanti ed eccessive ritrattazioni.

Nella definizione di matching rientra esclusivamente l'algoritmo che viene eseguito quando si verificano attivazioni destre e sinistre. Tuttavia, per

permettere tali operazioni e renderle meno costose, esiste una fase di costruzione e definizione delle strutture, pronte per essere attraversate in fase di matching. Per cui, è necessario distinguere la fase di **build** da quella di **match**: la prima viene eseguita subito dopo il parsing e l'interpretazione del file, costruendo la rete (alfa e beta), distribuendo i test nei punti opportuni e collegando i pattern di una medesima produzione tra di loro; la seconda, invece, attraversa i suddetti nodi, esegue i test nei nodi che prevedono determinati controlli ed infine costruisce e propaga dei record di fatti denominati *token*.

2.4.1 Alfa Network

La parte alfa della rete è composta da un nodo radice (**root node**), da nodi alfa (**alfa node**) e da nodi di memoria (**alfa memory**).

Fase di costruzione

La **costruzione** dell'*alfa network* avviene in avanti partendo da un nodo radice fino a raggiungere i nodi delle memorie alfa.

Dopo aver costruito un *nodo radice*, si collegano ad esso i *nodi alfa* che rappresentano il nome di ciascun pattern. Ad ogni diramazione corrisponde la scomposizione di un pattern. Se due o più pattern aventi lo stesso nome presentano gli stessi campi a partire dal primo, allora condividono tutti i nodi in comune fino a quanto non presentano campi diversi. I nodi terminali, cioè i nodi foglia, di ciascuna diramazione dell'alfa network sono dei *nodi alfa memory*, nei quali vengono memorizzati i *WME* riconosciuti dalla rete alfa.

L'alfa network è costruita mediante la chiamata di un metodo di **build**, presente nelle classi che rappresentano i nodi di tale rete. A ciascun nodo è delegata la costruzione del nodo successivo e la chiamata del metodo di build del nodo appena costruito. I nodi rappresentano i propri figli mediante dizionari di nodi indicizzati rispetto all'etichetta che ne rappresenta il campo corrispondente. La scelta del dizionario per la rappresentazione dei figli di un nodo è dettata dalla necessità di dover individuare immediatamente, durante la fase di build, la necessità di condividere o no un nodo e, durante la fase di

match, il nodo corrispondente al successivo valore di un fatto. Il modello di riferimento è stato quello della *Data Network with Hashing*[3].

Il metodo di build del *root node* riceve in input un pattern e costruisce un figlio di tipo *alfa node* o ne condivide uno già esistente, a seconda che il nodo abbia o no un figlio con l'etichetta corrispondente al primo elemento di tale pattern; in seguito tale metodo richiama il metodo di build del nodo appena costruito.

Il metodo di build della classe *alfa node* riceve in input un pattern e un dizionario, inizialmente vuoto, delle variabili incontrate durante la scansione del pattern. Il metodo in questione memorizza nel nodo corrente un valore di profondità, che indica la posizione, incrementata di una unità, del campo del pattern che si sta considerando, in modo da facilitare la successiva fase di match, dato che si stabilisce una corrispondenza uno-ad-uno tra campo del pattern e campo di un fatto. Il valore di profondità è uguale a 0 per il primo campo successivo al nome del pattern, ovvero per il primo livello di nodi alfa costruiti. Il metodo di build dei nodi alfa, in modo analogo a quanto detto per il nodo radice, costruisce un figlio, dato da un altro nodo alfa, o ne condivide uno esistente, a seconda che il nodo abbia o no un figlio con l'etichetta corrispondente al campo che si sta considerando; in seguito si richiama il metodo di build del nodo appena costruito.

La rappresentazione dei nodi alfa prevede, oltre al dizionario di figli indicizzati per campo e al valore che indica la profondità di tali nodi, anche l'etichetta dei nodi in questione, un insieme di riferimenti ai figli che corrispondono per eventuali variabili, fondamentale per evitare l'iterazione su nodi che rappresentano costanti durante il match di variabili, e un riferimento all'ultima variabile, se presente, con il nome analogo all'eventuale nome di variabile del nodo corrente, utile ai fini del controllo di uguaglianza in caso di variabili duplicate in uno stesso pattern.

Nel caso in cui si sia giunti alla costruzione dell'ultimo nodo dell'alfa network per il pattern corrente, allora si costruisce un nodo di tipo alfa memory, collegato al nodo corrente, e si richiama il metodo di build di tale nodo, il quale restituisce un riferimento a se stesso, che verrà propagato lungo

[3] `http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf`

le chiamate precedenti, finché non viene restituito a ritroso fino alla chiamata del metodo di build del nodo radice. Ciò permette la memorizzazione del riferimento di ciascuna memoria alfa dell'alfa network, ai fini del collegamento della beta network all'alfa network.

Fase di matching

La fase di **match** dell'*alfa network* è effettuata mediante la chiamata di un metodo di **match** presente nelle classi che rappresentano i nodi di tale rete. Il metodo di match riceve in input un WME e verifica se, nel dizionario dei figli del nodo corrente, vi è un nodo con etichetta uguale al campo corrente del WME, individuato mediante l'attributo locale di profondità costruito in fase di build: in tal caso si richiama il metodo di match del figlio corrispondente. Se il nodo corrente è stato etichettato come duplicato di una variabile precedente, allora si sfrutta tale riferimento per il controllo dei duplicati. Nel caso in cui l'insieme di riferimenti ad eventuali figli che rappresentano variabili, allora si richiama il metodo di match di tali nodi. Se è stata raggiunta la fine del pattern, e quindi se la fase di match per il WME corrente abbia avuto esito positivo fino al termine dell'alfa network, allora si richiama il metodo di **attivazione della beta network** presente nella memoria alfa associata al nodo terminale.

Il metodo di attivazione della beta network, presente ne nodi che rappresentano le memorie alfa, prevede la memorizzazione del WME ricevuto in input nella memoria interna del nodo costituita da un dizionario ordinato di WME indicizzate rispetto al loro identificativo numerico (tale struttura dati consente una facile rimozione delle WME durante eventuali operazioni di ritrattazione). Vengono inoltre memorizzate le istanze delle variabili identificate all'interno del pattern che ha superato la fase di match. Si richiama a quel punto il metodo di attivazione del **match** della **beta network**.

2.4.2 Beta Network

La parte beta della rete è composta da nodi di join (**join node** e **dummy join node**), nodi di memoria (**beta memory**) e da nodi di tipo production

(**pnode**).

Fase di costruzione

La costruzione della *beta network* avviene a ritroso partendo da un *production node* fino a raggiungere i nodi terminali rappresentati dalle memorie della rete alfa già generate.

Partendo da una regola, si scandisce ciascun pattern (test esclusi), si genera un nodo di join associando a quest'ultimo, come input destro, la memoria alfa del pattern; si crea un nodo di memoria beta per l'input sinistro e si ripetono scansione e costruzione sul pattern successivo fino al termine della scansione della parte sinistra di una regola. L'ultimo pattern letto è associato ad una memoria alfa tramite un nodo dummy join.

Nel caso in cui una regola sia composta esclusivamente da un singolo pattern, viene stabilito un link diretto al *production node*, il quale, nella fattispecie, si occuperà di verificare ciascun WME in arrivo dalla memoria alfa associata ed eventualmente provvederà a far scattare l'attivazione della regola di riferimento.

Durante la costruzione della rete, se un pattern è nella forma di un *assigned pattern ce*, allora si memorizza nel nodo di join il rifermento alla variabile che conterrà l'istanza di un pattern.

Inoltre, si memorizzano i test che coinvolgono le variabili di ciascun pattern all'interno del nodo di join di competenza: in fase di matching, in seguito al binding delle variabili, verranno eseguiti i test per verificarne il contenuto.

I test e le variabili si possono trovare anche in un production node qualora esista un collegamento diretto ad una memoria alfa.

L'implementazione delle memorie beta è stata realizzata utilizzando dizionari ordinati, i quali permettono un ritrovamento in un tempo ridotto rispetto ad una lista, mantenendo l'ordine di arrivo dei token. Ciò favorisce una riduzione dei tempi in fase di *retract*. Per ciò che concerne le variabili, invece, esse sono memorizzate in dizionari nella forma etichetta-valore.

Fase di matching

Quando un WME arriva in una memoria alfa, esso è pronto per essere propagato attraverso la beta network.

Un nodo *beta memory* contiene istanze parziali di regole di produzione, cioè combinazioni di WME che soddisfano in parte una o più regole, cosiddetti **token**.

Un nodo *join node* ha *due input*: uno *sinistro* a cui è associata una memoria beta ed uno *destro*, a cui è connessa una memoria alfa. Ogni nodo di join si occupa sia di effettuare il binding delle variabili, in comune e non, tra condizioni diverse, che di effettuare i test sugli input ricevuti. L'input sinistro è rappresentato da un token in arrivo dalla memoria beta mentre quello destro è dato da un WME proveniente dalla memoria alfa.

Il binding delle variabili in comune consiste nel verificare che, tutte quelle aventi lo stesso nome, sia nel token che nel WME considerati, abbiano anche lo stesso valore.

Quando un token giunge in un nodo join si parla di **attivazione sinistra** mentre l'arrivo di un WME fa scattare un'**attivazione destra**; le attivazioni non possono verificarsi contemporaneamente ed il processo di matching non procede finché non abbiano luogo entrambe: un'attivazione deve attendere il verificarsi dell'altra per poter proseguire la fase di matching.

Se esistono input validi da sinistra e da destra, allora il nodo join si occupa di generare un nuovo elemento unendo i due input in un nuovo token da inviare alla beta memory figlia. Un caso speciale è rappresentato dal nodo di tipo dummy join che possiede un solo input al quale è associata una memoria alfa (input destro): quando un WME supera il matching nell'alfa network, viene incapsulato in un nuovo token dal dummy join ed il matching prosegue. Il matching in un nodo di join è descritto negli algoritmi 1 e 2.

Le memorie beta ed alfa permettono di riconsiderare rispettivamente token e WME respinti precedentemente, quando certe condizioni sono soddisfatte, con il vantaggio di evitare la ripetizione della fase di matching per ognuno di essi.

Se un token giunge in un **production node**, allora tutte le condizioni

della regola rappresentata dal nodo sono soddisfatte e scatta l'attivazione di quest'ultima; l'insieme delle azioni, presenti nella parte destra della regola, viene aggiunto al conflict set. Il nodo che rappresenta una produzione viene raggiunto in ultima istanza, per cui rappresenta un nodo terminale. Nelle figure 2.3 e 2.2 si possono osservare esempi di matching.

Ogni volta che avviene una propagazione da un nodo di join, vengono generati un nuovo token ed un nuovo dizionario di variabili, unendo le variabili del token considerato con quelle del WME considerato.

Come già accennato, la fase di ritrattazione di fatti dalla Working Memory implica la cancellazione di tutte le occorrenze dei WME relativi dalla rete, nonché di tutti token in cui tali WME sono presenti. Un token o un WME, quindi, viene eliminato solo quando è eseguita un'azione di **retract**, altrimenti resta nelle memorie dove occorre, in attesa di partecipare ad un nuovo match.

```
Dati: token
Risultato: new token
se esistono WME nella memoria alfa allora
    determina le variabili in comune tra le memorie alfa e beta;
    per ciascun WME fai
        confronta le istanze delle variabili nel WME rispetto al token;
        se il matching va a buon fine allora
            fondi le variabili delle memorie alfa e beta;
            esegui i test;
            se i test sono andati a buon fine allora
                recupera i pattern assignment per il token in input;
                costruisci un nuovo token Token(token in input, WME
                corrente);
                propaga ai figli del nodo il nuovo token generato, le
                variabili ed i pattern assignment;
            fine
        fine
    fine
fine
```

Algoritmo 1: Beta network - Attivazione sinistra di un nodo join.

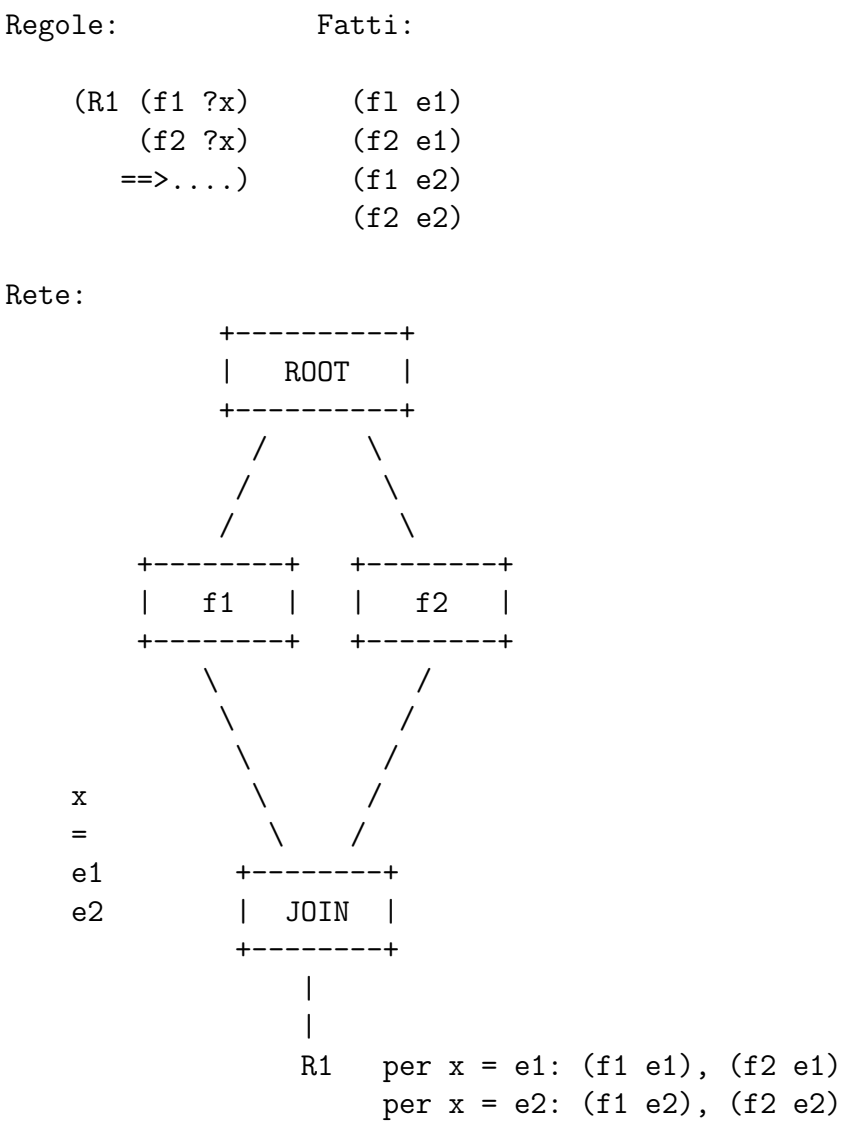

Figura 2.2: Esempio di matching.

2.5 Risoluzione dei conflitti

Al termine della fase di matching, viene generata un insieme di regole attivabili le cui condizioni sono soddisfatte dai fatti all'interno della Working Memory. Tale insieme prende il nome di *Conflict Set*. Dal momento che il sistema è ispirato a CLIPS, il conflict set prende il nome di *Agenda*. Il fine

Dati: WME
Risultato: new token
se *esistono token nella memoria beta* **allora**
    determina le variabili in comune tra le memorie alfa e beta;
    per ciascun *token* **fai**
        confronta le istanze delle variabili nel token rispetto al WME;
        se *il matching va a buon fine* **allora**
            fondi le variabili delle memorie alfa e beta;
            esegui i test;
            se *i test sono andati a buon fine* **allora**
                recupera i pattern assignment per il token corrente;
                costruisci un nuovo token Token(token corrente, WME in input);
                propaga ai figli del nodo il nuovo token generato, le variabili ed i pattern assignment;
            fine
        fine
    fine
fine

Algoritmo 2: Beta network - Attivazione destra di un nodo join.

dell'insieme determinato è quello di risolvere i conflitti tra le regole attivabili selezionandone una ed eseguendone le azioni presenti nella parte destra. La scelta della regola da applicare è effettuata mediante strategie di risoluzione dei conflitti, le quali ordinano le regole applicabili considerandone le proprietà intrinseche e il loro valore di priorità, che prende il nome di *salience*. Dopo l'ordinamento delle regole viene applicata la prima della lista.

L'agenda implementata in TURTLE è composta da un dizionario che associa le salience delle regole attivabili con le liste delle regole aventi tale salience e da un dizionario che associa a ciascun token il numero di regole che esso attiva. L'agenda, inoltre, tiene traccia della strategia scelta e ne permette il cambiamento.

In TURTLE sono state realizzate le seguenti strategie:

- **Depth**: È la strategia, come in CLIPS, di **default**. Tale strategia ordina le regole attivabili in base al loro tempo di attivazione: le regole attivate dal fatto più recente vengono posizionate al di sopra delle regole

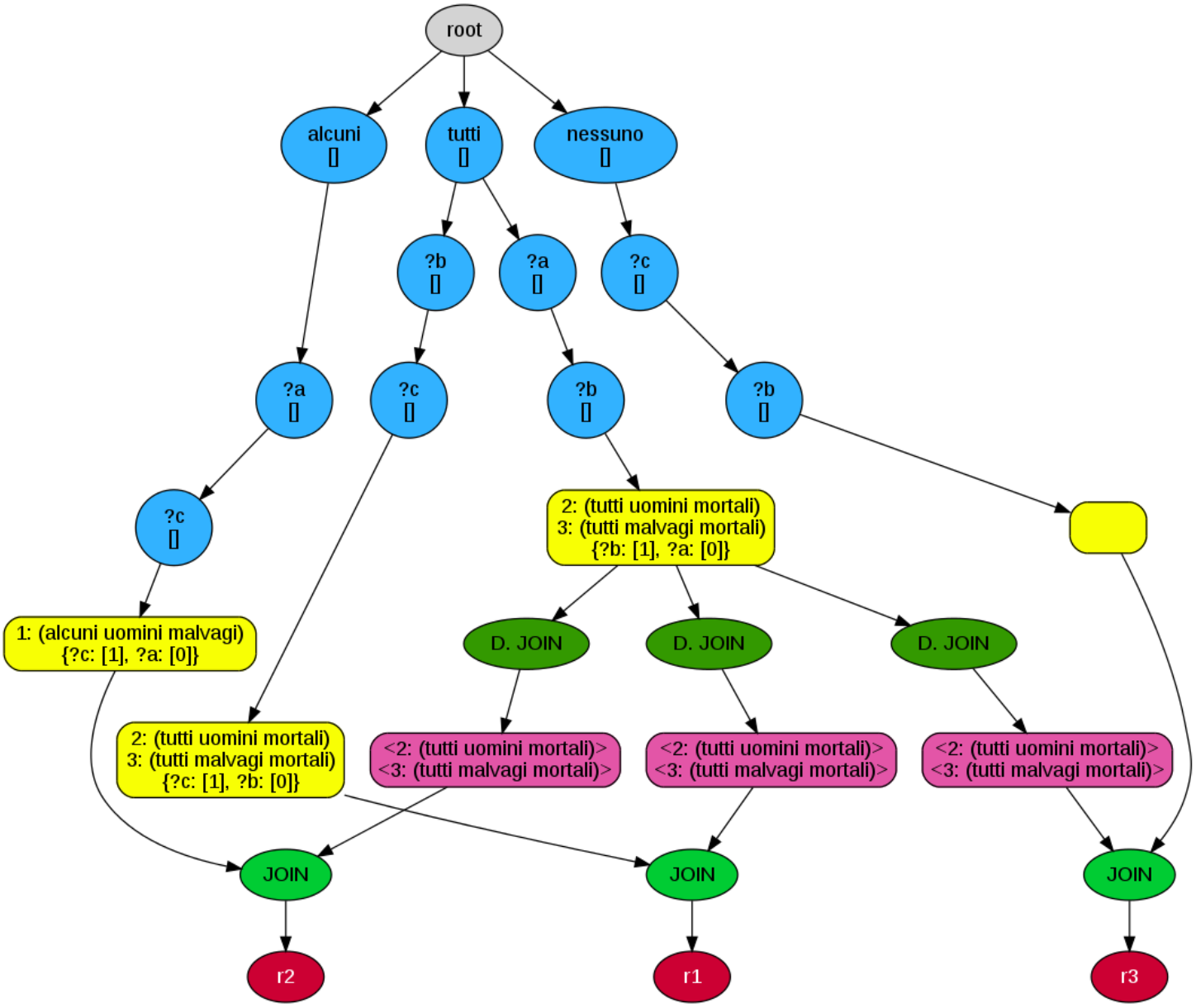

Figura 2.3: Matching beta network per il programma Sillogismi.clp presente nell'appendice di questo documento.

aventi la stessa salience attivate dai fatti meno recenti. La strategia in questione è denominata depth poiché è una depth-first-search.

- **Breadth**: Le regole attivate dal fatto più recente vengono posizionate al di sotto delle regole aventi la stessa salience attivate dai fatti meno recenti. La strategia in questione è denominata breadth poiché è una breadth-first-search, in cui l'ordine delle attivazioni è gestito come se fosse una coda.

- **Random**: Le regole attivabili vengono mescolate in un ordine casuale.
- **Simplicity**: Tra tutte le regole con la stessa salience, le nuove attivazioni vengono inserite al di sopra di quelle corrispondenti a regole con uguale o maggiore specificità, definita in base ai seguenti criteri di analisi per la loro parte sinistra:
  - **+1** per ciascun pattern.
  - **+1** per ciascuna variabile univoca.
  - **+1** per ciascuna funzione o predicato.
- **Complexity**: Tra tutte le regole con la stessa salience, le nuove attivazioni vengono inserite al di sopra di quelle corrispondenti a regole con uguale o minore specificità.
- **LEX**: È una implementazione Naive della strategia LEX di CLIPS. Le regole attivate dai fatti più recenti vengono posizionate al di sopra delle regole aventi la stessa salience attivate dai fatti meno recenti. Rispetto alla strategia depth la priorità di una regola è data da un ordinamento lessicografico dei fatti che la attivano piuttosto che dall'individuazione del fatto più recente.
- **MEA**: È una implementazione Naive della strategia MEA di CLIPS. Le regole in cui il primo fatto dei token che le attivano è più recente vengono posizionate al di sopra di quelle in cui il primo fatto dei token che le attivano è meno recente.

2.6 La shell d'interazione

Per caricare un programma, eseguirlo, ed interagire con esso, è necessario avviare TURTLE da riga di comando[4]. Il file da passare in input all'interprete Python è **Turtle.py** (**python -B Turtle.py**)[5].

Dopo aver avviato il sistema, un prompt sarà a disposizione per ricevere i comandi necessari al caricamento e all'esecuzione di un programma a regole, solitamente contenuto in un file di testo con estensione ".clp".

```
                 __
     .,-;-;-,. /'_\  TURTLE Pre-Release
   _/_/_/_|_\_\) /
 '-<_><_><_><_>=/\   An expert system shell inspired by CLIPS syntax
   `/_/====/_/-'\_\  Running on Python 2.7.3 32bit
    ''     ''    ''
  Usage: help to see online help
         <Tab> to show commands
         <Up-arrow>, <Down-arrow> to scroll through the history
         <Ctrl-c> to break a running program
         <Ctrl-d>, quit or exit to leave


TURTLE>
```

Figura 2.4: Prompt della shell.

Nella figura 2.4 viene mostrato come si presenta il prompt dei comandi all'avvio del sistema. L'interfaccia testuale propone un banner informativo con le scorciatoie da tastiera ed i comandi utili per interagire, i quali offrono diverse comodità come la possibilità di richiamare comandi già digitati in precedenza, il completamento automatico e l'help online.

Quest'ultimo fornisce spiegazioni sull'utilizzo di tutti i comandi oltre che le liste di funzioni e predicati disponibili ed utilizzabili all'interno di fatti e

[4]Sono necessarie alcune dipendenze, specificate in un file di testo nella directory del progetto.

[5]Il parametro "-B" serve ad evitare la creazione del file .pyc contenente il bytecode; quest'ultimo permette di avviare rapidamente il sistema, ma non offre alcun vantaggio in fase di esecuzione.

regole. E' possibile eseguire un programma impostando un limite d'esecuzione oltre il quale viene fermato il ciclo riconosci-agisci, lasciando in agenda tutte le attivazioni non ancora eseguite.

Per fornire compatibilità sintattica con CLIPS, è possibile racchiudere opzionalmente ogni comando tra parentesi tonde; per cui, ad esempio, il comando **help** è equivalente al comando **(help)**.

La shell dispone del completamento automatico del testo premendo il tasto TAB. Se quest'ultimo viene premuto senza aver digitato alcun carattere, verrà mostrato l'intero elenco dei comandi che è possibile impartire alla shell.

Si possono definire fatti, regole e variabili globali direttamente da shell oltre che caricare uno o più file. Al contrario di CLIPS [6], tutti gli elementi definiti dalla shell tramite *assert* o *deffacts* (oltre quelli caricati da file), dopo un *reset*, saranno mantenuti in memoria. Attualmente la command line non accetta comandi multi-riga.

Se si intende ripulire l'intero ambiente è necessario il comando *clear*, come avviene in CLIPS.

[6]CLIPS, dopo un reset ripristina solo tutti i fatti che sono stati definiti tramite il costrutto deffacts, da shell o da file.

Capitolo 3

Dettagli implementativi

L'implementazione del sistema è interamente realizzata in **Python**[1], un linguaggio di grande supporto all'implementazione immediata di idee, data la sua notevole **espressività**. Quest'ultima si è rivelata un vantaggio per esprimere e condividere concetti sotto forma di codice chiaro e compatto. Tuttavia, poiché quest'ultimo è interamente interpretato, il problema che si riscontra maggiormente è in alcuni bug che spesso passano inosservati, se non definendo opportuni test grazie al supporto di moduli come il debugger **pdb** [2].

3.1 Il sistema dei tipi

I tipi di dati realizzati fanno riferimento a quelli proposti da CLIPS (nonché dal linguaggio LISP). Utilizzando l'ereditarietà, si è definito un tipo di base denominato **BaseType** dal quale derivano tipi che a loro volta sono superclassi e costituiscono la seguente gerarchia:

- **BooleanType**: rappresenta un valore booleano e si indica con i simboli `TRUE` e `FASLE`.
- **NumberType**: superclasse per la definizione di tipi numerici.

[1] La versione utilizzata è la 2.7.

[2] E' possibile mandare in esecuzione il sistema in modalità debug utilizzando come parametro aggiuntivo **-m pdb** (`http://docs.python.org/2/library/pdb.html`).

  - **IntegerType**: rappresenta un valore intero, positivo o negativo. Il valore intero massimo specificabile è legato all'implementazione in Python[3].
  - **FloatType** : rappresenta un valore floating point, positivo o negativo. Il valore intero massimo specificabile è legato all'implementazione in Python [4].

- **LexemeType**: superclasse per la definizione di tipi alfanumerici.
  - **SymbolType**: rappresenta il tipo maggiormente utilizzato, composto da caratteri alfanumerici e dai simboli "`-`" e "`_`".
  - **StringType**: rappresenta il tipo di dato stringa racchiusa tra doppi apici "`"`" e contenente qualsiasi carattere stampabile. Questo tipo si differenzia dal SymbolType per il numero di caratteri utilizzabili e soprattutto per la capacità di poter utilizzare gli spazi.

- **VariableType**: superclasse per la definizione dei tipi di variabili. Le variabili trattate non prevedono **wildcard**.
  - **SinglefieldVariableType**: rappresenta una variabile single field che interessa un singolo campo di un fatto o di un pattern. Essa si dichiara all'interno della parte sinistra di una regola e, se necessario, anche nella parte destra. Alcuni esempi di variabili single field sono: `?a`, `?var`, `?a1`, `?var1`.
  - **GlobalVariableType**: rappresenta una variabile globale. Le variabili globali sono condivise durante tutta l'esecuzione di un programma. Hanno una sintassi differente da quelle locali; alcuni esempi sono: `?*data-nascita*`, `?*x*`, `?*y1*`.

- **FunctionType**: superclasse per la definizione dei tipi di funzioni disponibili.

[3]Il massimo intero specificabile, per Python v2.x, è definito in *sys.maxint* mentre il minimo equivale a quest'ultimo più uno.

[4]Per maggiori informazioni consultare *sys.float_info*.

- **FunctionCallType**: rappresenta una chiamata ad una funzione o ad un predicato utilizzabili sia nelle due parti delle regole che nella definizione di fatti. E' possibile quindi definire un campo di un fatto come il risultato di una determinata funzione che lavori su numeri, stringhe o valori booleani.
- **SpecialFunctionCallType**: rappresenta una chiamata ad una funzione speciale. Le funzioni speciali permettono di interagire e manipolare lo stato del sistema piuttosto che elaborare solo dati come invece fanno le funzioni. Alcuni esempi sono: `assert`, `retract`, `printout`. Si utilizzano nella RHS di una regola eccetto la funzione speciale `test` che viene utilizzata nella LHS per verificare dei predicati sulle variabili.

3.2 Variabili, funzioni e valutazione

Come già osservato, il sistema supporta l'utilizzo di variabili di differente tipo. Nella versione attuale del sistema è possibile utilizzare le variabili *globali* e *single field*. Esse possono essere utilizzate sia nei fatti che nelle regole, oltre che nelle funzioni.

Quando un fatto viene definito all'interno del costrutto **deffacts** è possibile utilizzare esclusivamente variabili globali poiché la shell d'interazione non prevede la possibilità di definire variabili locali da riga di comando. In questo costrutto è possibile utilizzare funzioni e predicati che coinvolgano o meno variabili globali; la valutazione delle variabili, di funzioni e di predicati, avviene prima dell'inserimento dei fatti all'interno della working memory. Se una variabile globale non è stata definita o viene utilizzata in una funzione che non accetta parametri del tipo della variabile, il sistema interrompe l'operazione corrente ed un'eccezione viene sollevata.

Nella definizione di un pattern di un fatto, nella parte sinistra della regola, è possibile specificare variabili locali (single field) e globali. Tali variabili concorreranno al match durante il *recognize-act cycle*; non sono ancora presenti pattern conditional element di tipo not e or. Tuttavia, si consideri che,

per le variabili, è possibile definire dei test opportuni (sempre nella LHS di una stessa regola) per verificare uno o più predicati.

Se una regola viene attivata, le sue azioni potrebbero consistere nella stampa su *stdout* di contenuti elaborati, nell'asserzione di nuovi fatti o nella ritrattazione di essi. In tali casi è possibile utilizzare variabili globali e locali[5], funzioni e predicati.

Per tenere traccia delle variabili globali (o condivise) durante l'esecuzione di un programma, è stato realizzato un **environment** al quale accedono tutte le componenti interessate e che viene condiviso durante tutto il ciclo di esecuzione di un programma, dal parsing del file fino all'esecuzione dell'ultima regola applicabile. Perciò, quando si rende necessaria la valutazione di una variabile, viene consultato l'environment per ritrovarne il valore corrente.

La visita e la valutazione di variabili, funzioni e predicati avviene tramite la classe **evaluator**, che si occupa di visitare ciascun nodo in base al suo tipo e di valutarne il contenuto. L'evaluator si compone di un metodo di default che restituisce il nodo visitato nel caso in cui non sia stato definito un comportamento per quel tipo di nodo, e di metodi definiti per ciascun tipo che si occupano di elaborarne il contenuto ove possibile. Infine, è presente il metodo *evaluate* che cerca di valutare il contenuto del nodo richiamando se stesso nel caso in cui vi siano annidamenti (ad esempio nel caso di nodi che contengono funzioni).

Le funzioni ed i predicati possono essere utilizzati quando si definiscono dei fatti, nei test presenti nella parte sinistra della regola e nella parte destra di quest'ultima.

3.3 Strutture dati per il conflict set

La risoluzione dei conflitti implica l'utilizzo di una strategia che riordini l'insieme in base a determinati criteri. In TURTLE, sono stati adottati

[5]Il controllo della presenza di una variabile locale nella LHS non viene effettuato in fase di parsing per non introdurre ulteriori controlli. Tale inconsistenza si noterà solo in fase d'esecuzione. Si lascia al buon senso e alla responsabilità dello sviluppatore l'attenzione verso questi effetti collaterali.

```
(deffacts
   (soggetto1 eta 30)
   (soggetto2 eta 20)
   ...)

(defrule maggiore
   (?s1 eta ?x)
   (?s2 eta ?y)
   (test (> ?x ?y))
   =>
   (assert (?s1 maggiore ?s2))
   (printout ?s1 "è più grande di" ?s2)
   (printout "totale:" (+ ?x ?y)))
```

Figura 3.1: Utilizzo di variabili e funzioni in una regola.

diversi *container* dati per adattarsi al tipo di strategia utilizzato ed eseguire attivazioni in un tempo costante. Per le strategie più complesse è stato definito un tipo di dato **KeyHeapq** allo scopo di riordinare le attivazioni sfruttando una coda con priorità e specificando, a seconda del caso, la chiave per effettuare tale ordinamento. I container utilizzati sono:

- **Lista** per la strategia **Depth**: permette l'inserimento e la rimozione di un elemento come se fosse una pila[6]
- **Deque** per la strategia **Breadth**.
- **Lista** per la strategia **Random**.
- **KeyHeapq** per la strategia **Simplicity** con chiave la **complessità** di una regola.
- **KeyHeapq** per la strategia **Complexity** con chiave la **complessità** negativa di una regola.

[6]In Python una lista si può utilizzare come fosse una pila attraverso i metodi *append* e *pop* rispettivamente per inserire e rimuovere elementi dalla coda in un tempo costante.

- **KeyHeapq** per la strategia **LEX** con chiave gli **indici dei fatti** che soddisfano una regola.
- **KeyHeapq** per la strategia **MEA** con chiave il primo **indice di un fatto all'interno di un token** che soddisfa una regola.

3.4 Confronto con CLIPS

TURTLE, allo stato attuale, supporta solo un piccolo sottoinsieme di feature ereditate da CLIPS, con qualche variazione sintattica. Le caratteristiche principali sono:

- Supporto dei fatti ordinati (senza template).
- Presenza di variabili globali definite attraverso il costrutto **defglobal** ed utilizzabili nella parte destra di una regola, come parametro di una funzione e come campo nella definizione di un nuovo fatto. Rispetto a CLIPS, si è scelto di poter utilizzare le variabili globali anche nella parte sinistra di una regola.
- Utilizzo del costrutto **deffacts** per asserire uno o più fatti con la possibilità di utilizzare funzioni e variabili globali.
- Utilizzo del costrutto **defrule** per definire una regola[7].
- Utilizzo del costrutto **declare** in defrule per definire le proprietà di una regola. Attualmente l'unica proprietà supportata è la **salience**.
- Presenza di variabili locali **single field**.
- Possibilità di utilizzare la funzione speciale **test** nella parte sinistra di una regola per verificare uno o più predicati su una o più variabili locali.
- Utilizzo della fuzione speciale **assert** nella parte destra di una regola per asserire uno o più fatti contenenti costanti, variabili globali e variabili locali già presenti nella parte sinistra.

[7]Una regola è considerata valida solo se ha almeno un pattern nella parte sinistra. Ne consegue che non sono ammesse attivazioni di tipo wildcard.

- Utilizzo della funzione speciale **retract** per eliminare uno o più fatti.
- Presenza della funzione speciale **bind** per modificare una variabile globalmente o localmente a seconda del tipo.
- Possibilità di variare il criterio di ordinamento del conflict set (agenda) tramite la funzione speciale **strategy**, sia tramite shell che in fase d'esecuzione, specificando la strategia da adottare tra le azioni nella parte destra di una regola.
- Possibilità di stampare contenuti su standard output tramite la funzione speciale **printout**.

Consultare l'help online presente nella shell[8] per conoscere meglio tali feature.

3.5 Estensibilità

Durante lo sviluppo del sistema, si è cercato di rendere il codice scalabile per sviluppi futuri. L'intento di realizzare un sistema di base che fosse predisposto alla realizzazione di nuove feature è stato al centro del processo di progettazione.

Tra le estensioni che si potrebbero introdurre in futuro, vi sono due punti cruciali che necessitano una certa elasticità per un facile aggiornamento: l'introduzione di nuovi nodi nell'algoritmo di matching e l'inserimento di nuove funzioni utilizzabili con i tipi di dati presenti.

Nel primo caso si è realizzata una superclasse *Node* che è possibile estendere con le caratteristiche del nodo che si intende implementare, mentre, per il secondo caso è stata realizzata una classe denominata *FunctionMapper*, la quale si occupa di caricare le funzioni disponibili leggendo dei file *.py* presenti all'interno della directory in cui specificare nuove funzioni. Tali file estendono la classe *Module*, definita per permettere all'utente di inserire nuovi file

[8] digitare **help *nome_comando*** per ottenere maggiori informazioni.

.py contenenti nuove funzioni e nuovi predicati definiti in base alle proprie esigenze[9].

[9]Per comprendere meglio come realizzare nuove funzioni e nuovi predicati, consultare i file *Functions.py* e *Predicates.py*.

Capitolo 4

Conclusioni

Allo stato attuale il sistema non supporta diverse componenti che CLIPS fornisce. Di seguito vengono proposti alcuni sviluppi possibili per il futuro sia a livello di feature che di modifiche architetturali al fine di ottimizzare il sistema. E' stato effettuato qualche test per verificare le prestazioni del sistema. Ad esempio, è stato riscontrato, su una stessa macchina, che CLIPS esegue 22000 attivazioni al secondo contro le 800 eseguite da TURTLE; stando ad una valutazione preliminare, TURTLE è risultato essere circa 30-35 volte più lento di CLIPS[1].

4.1 Sviluppi futuri

Per semplificare e rendere più flessibili sintassi e semantica si potrebbero introdurre le variabili **multifield** ed il supporto alle **wildcard** (? e $?).

Due conditional element molto utili per evitare duplicazioni di regole e test sono l'**OR CE**, il **NOT CE** ed un'analisi per scomporre gli **AND CE**[2].

Per ciò che concerne la definizione dei fatti, un metodo che fornirebbe maggiore dettaglio nella formalizzazione è quello dei **template**; in tal caso si

[1]Test effettuati su una macchina con processore Intel P8600 2.4GHz. Va considerato anche l'overhead generato da Python rispetto al linguaggio C (con il quale è stato realizzato CLIPS).

[2]Come già specificato, da non confondere con i predicati *and*, *or* e *not* utilizzabili nei test.

parlerebbe di *unordered facts* poiché i fatti in questione sarebbero indicizzati per attributo.

Attualmente, in TURTLE, la rete costruita può essere aggiornata, dopo la costruzione, aggiungendo nuove regole tramite shell o file. Non è possibile rimuovere regole se non ricostruendo l'intera rete.

Per evitare riscritture di regole potrebbe tornare utile l'utilizzo del costrutto **if-then-else** per avere, all'interno di una regola, una parte destra più articolata.

Il costrutto **deffunction** di CLIPS fornirebbe un livello di astrazione migliore. Allo stato attuale, è possibile definire le proprie funzioni solamente in Python.

Infine, per fornire tutti gli elementi della logica proposizionale, dovrebbero essere integrati i quantificatori **forall** (per ogni) ed **exists** (esiste).

A livello architetturale, sarebbe possibile velocizzare la fase di retract (e di conseguenza l'intero sistema) attraverso la condivisione di memorie beta imputabili agli stessi gruppi di pattern e relativi test. Inoltre, ottimizzazioni nella fase di scomposizione di una regola apporterebbero migliorie in termini di tempo durante la fase di matching distribuendo maggiore carico di lavoro alla fase di costruzione.

Per permettere lo sviluppo incrementale di un programma a regole, risulterebbe molto utile la possibilità di rimuovere regole ed aggiungerne di nuove senza dover ricostruire interamente la rete; inoltre, introdurre il salvataggio di snapshot dello stato attuale del mondo (fatti e regole, sia di partenza che aggiunte o rimosse in fase di run) renderebbe più agile la ripresa di una computazione dopo una terminazione forzata o involontaria del sistema.

Un'implementazione interessante potrebbe essere l'inserimento dell'algoritmo di ricerca informata **A*** al fine di poter utilizzare un programma *skeleton* da modellare in base al problema da risolvere, specificando, ad esempio, una funzione euristica, una funzione costo di passo o gli stati del problema.

4.2 Ringraziamenti

La realizzazione di questo sistema è stata possibile grazie ai numerosi individui che hanno contribuito al mondo open-source, rilasciando componenti software che si sono rivelate fondamentali per TURTLE.

Grazie anche a coloro che hanno reso disponibili tesi e pubblicazioni inerenti l'algoritmo RETE, documenti fondamentali senza i quali difficilmente avremmo potuto procedere nello sviluppo di un algoritmo di matching RETE-like.

Infine, ma non per ultimo, un sentito ringraziamento alla *Professoressa Floriana Esposito* per averci trasmesso tenacia, determinazione, motivazione e per aver suscitato in noi studenti, durante il corso di *Ingegneria della Conoscenza e Sistemi Esperti*, un interesse non indifferente verso questa branca dell'intelligenza artificiale.

Appendice A

Grammatica del linguaggio

BNF estesa

⟨*printables*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | ! | # | $ | % | & | (|) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [| \ |] | ^ | _ | { | } | ~ | ´ | ¨

⟨*nums*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*alphas*⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨*alphanums*⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*sign*⟩ ::= `+` | `-`

⟨*unsigned int*⟩ ::= ⟨*nums*⟩+

⟨*integer*⟩ ::= [⟨*sign*⟩]⟨*unsigned int*⟩

⟨*exponent*⟩ ::= { '`e`' | '`E`' }⟨*integer*⟩

⟨*float*⟩ ::= ⟨*integer*⟩'.'⟨*unsigned int*⟩[⟨*exponent*⟩]
| ⟨*integer*⟩⟨*exponent*⟩
| ⟨*integer*⟩'.'[⟨*exponent*⟩]
| [⟨*sign*⟩]'.'⟨*unsigned int*⟩[⟨*exponent*⟩]

⟨*boolean*⟩ ::= `TRUE` | `FALSE`

⟨*string*⟩ ::= '`"`' ⟨*printables*⟩+ '`"`'

⟨*printout string*⟩ ::= ⟨*printables*⟩+

⟨*comment*⟩ ::= ⟨*string*⟩

⟨*symbol*⟩ ::= ⟨*alphas*⟩{⟨*alphanums*⟩| '`_`' | '`-`' }*

⟨*constant*⟩ ::= ⟨*boolean*⟩
| ⟨*symbol*⟩
| ⟨*string*⟩
| ⟨*float*⟩
| ⟨*integer*⟩

⟨*deffacts name*⟩ ::= ⟨*symbol*⟩

⟨*variable symbol*⟩ ::= ⟨*alphas*⟩+⟨*alphanums*⟩*

⟨*global variable*⟩ ::= '`?*`'⟨*vairable symbol*⟩'`*`'

⟨*singlefield variable*⟩ ::= '`?`'⟨*vairable symbol*⟩

⟨*variable*⟩ ::= ⟨*singlefield variable*⟩
| ⟨*global variable*⟩

⟨*special function names*⟩ ::= '`assert`' | '`retract`' | '`bind`' | '`printout`'
| '`strategy`'

⟨*function name*⟩ ::= ⟨*printables*⟩+
-{⟨*special function names*⟩ | '(' | ')' }*

⟨*expression*⟩ ::= ⟨*costant*⟩
| ⟨*variable*⟩
| ⟨*function call*⟩

⟨*function call*⟩ ::= '(' ⟨*function name*⟩⟨*expression*⟩* ')'

⟨*fact field*⟩ ::= ⟨*global variable*⟩
| ⟨*constant*⟩
| ⟨*function call*⟩

⟨*ordered fact pattern*⟩ ::= '(' ⟨*symbol*⟩⟨*fact field*⟩* ')'

⟨*fact pattern*⟩ ::= ⟨*ordered fact pattern*⟩

⟨*deffacts construct*⟩ ::= '(' '`deffacts`' ⟨*deffacts name*⟩[⟨*comment*⟩]
⟨*fact pattern*⟩* ')'

⟨*rule name*⟩ ::= ⟨*symbol*⟩

⟨*integer expression*⟩ ::= ⟨*expression*⟩

⟨*rule property*⟩ ::= '(' '`salience`' ⟨*integer expression*⟩ ')'

⟨*declaration*⟩ ::= '(' '`declare`' ⟨*rule property*⟩ ')'

⟨*rhs term*⟩ ::= ⟨*constant*⟩
| ⟨*variable*⟩
| ⟨*function call*⟩

⟨*lhs term*⟩ ::= ⟨*constant*⟩
| ⟨*singlefield variable*⟩

⟨*rhs constraint*⟩ ::= ⟨*rhs term*⟩

⟨*lhs constraint*⟩ ::= ⟨*lhs term*⟩

⟨*ordered pattern ce*⟩ ::= '(' ⟨*symbol*⟩⟨*lhs constraint*⟩* ')'

⟨*pattern ce*⟩ ::= ⟨*ordered pattern ce*⟩

⟨*assigned pattern ce*⟩ ::= ⟨*singlefield variable*⟩ '`<-`' ⟨*pattern ce*⟩

⟨*test ce*⟩ ::= '(' '`test`' ⟨*function call*⟩ ')'

⟨*conditional element*⟩ ::= ⟨*boolean*⟩
| ⟨*test ce*⟩
| ⟨*pattern ce*⟩
| ⟨*assigned pattern ce*⟩

⟨*conditional elements*⟩ ::= ⟨*conditional element*⟩*

⟨*rhs pattern*⟩ ::= ⟨*ordered rhs pattern*⟩

⟨*rhs patterns*⟩ ::= ⟨*rhs pattern*⟩+

⟨*rhs function call*⟩ ::= '(' '`assert`' ⟨*rhs patterns*⟩')'
| '(' '`retract`' {⟨*singlefield variable*⟩|⟨*unsigned int*⟩}+')'
| '(' '`bind`' ⟨*variable*⟩⟨*expression*⟩')'
| '(' '`printout`' {⟨*prinout string*⟩|⟨*expression*⟩}+')'
| '(' '`strategy` {`depth`|`breadth`|`random`|`complexity`|`simplicity`|`lex`|`mea`}')'

⟨*action*⟩ ::= ⟨*rhs function call*⟩

⟨*defrule construct*⟩ ::= '(' '`defrule`' ⟨*rulename*⟩[⟨*comment*⟩][⟨*declaration*⟩]
⟨*conditional elements*⟩ '`=>`' ⟨*action*⟩* ')'

⟨*global assignment*⟩ ::= ⟨*global variable*⟩ '`=`' ⟨*expression*⟩

⟨*defglobal construct*⟩ ::= '(' '`defglobal`' ⟨*global assignment*⟩* ')'

⟨*construct*⟩ ::= ⟨*deffacts construct*⟩
| ⟨*defrule construct*⟩
| ⟨*defglobal construct*⟩

⟨*line comment*⟩ ::= {';'}+ ⟨*any character until EOL*⟩

⟨*multiline comment*⟩ ::= {'`/*`'}+ ⟨*any character*⟩ + {'`*/`'}

⟨*program*⟩ ::= {⟨*construct*⟩|⟨*line comment*⟩|⟨*multiline comment*⟩}*

Appendice B

Funzioni e Predicati

B.1 Funzioni

B.1.1 NumberType

Ciascun termine delle seguenti funzioni deve essere numerico (`IntegerType` o `FloatType`) oppure una variabile di tipo numerico che verrà sostituita automaticamente se istanziata correttamente.

Addition

Sintassi: (+ x_1 x_2 ... x_n)

Somma i termini $x_1, ..., x_n$. Il passo base è `IntgerType(0)`.

Subtraction

Sintassi: (− x_1 x_2 ... x_n)

Sottrae i termini $x_1, ..., x_n$. Il passo base è x_n.

Multiplication

Sintassi: (∗ x_1 x_2 ... x_n)

Moltiplica i termini $x_1, ..., x_n$. Il passo base è `IntegerType(1)`.

Division

Sintassi: (/ x_1 x_2 ... x_n)

Divide i termini $x_1, ..., x_n$. Il passo base è x_1.

Module

Sintassi: (% x_1 x_2 ... x_n)

Calcola il resto della divisione tra i termini $x_1, ..., x_n$. Il passo base è x_1.

Power

Sintassi: (** x_1 x_2 ... x_n)

Calcola la potenza di x_1 con esponente il prodotto dei termini $x_2, ..., x_n$. Il passo base è x_1.

Abs

Sintassi: (*abs* x)

Calcola il valore assoluto di x. E' un wrapper per la funzione `abs` di Python.

Minimum

Sintassi: (*min* x_1 x_2 ... x_n)

Trova il minimo tra i termini $x_1, ..., x_n$. E' un wrapper per la `min` di Python.

Maximum

Sintassi: (*max* x_1 x_2 ... x_n)

Trova il massimo tra i termini $x_1, ..., x_n$. E' un wrapper per la funzione `max` di Python.

Randint

Sintassi: (*randint* x_1 x_2)

Restituisce un numero intero casuale compreso tra x_1 e x_2 inclusi. Il primo termine non deve essere necessariamente più piccolo del secondo.

B.1.2 LexemeType: StringType

Strcat

Sintassi: (*strcat* x_1 x_2 ... x_n)

Concatena le stringhe $x_1, ..., x_n$ restituendo un nuovo `StringType`.

Substr

Sintassi: (*substr* string start end)

Restituisce una sottostringa compresa tra gli indici di posizione start ed end (escluso) di una stringa passata in input. La funzione si comporta come lo *slicing* di Python, in cui l'intervallo considerato inizia dalla posizione start e termina nella posizione end-1. Se entrambi gli indici superano il massimo range specificabile, viene restituita l'intera stringa in input. Se uno solo degli indici supera il range di posizioni possibili si hanno due casi: se `start` è out of range allora viene restituita una sottostringa data dagli indici `0` ed `end` (`[0:end]`); se `end` è out of range viene restituita una sottostringa data dagli indici `start` e `ultima posizione stringa in input` (`[start:]`).

Strlen

Sintassi: (*strlen* string)

Calcola la lunghezza della stringa in input ed utilizza un `IntegerType` come valore di ritorno.

Strindex

Sintassi: (*strindex* string substring)

Cerca un carattere o una sottostringa all'interno di una stringa e, in caso di successo, restituisce la posizione iniziale della sequenza di caratteri ricercata. In caso contrario viene restituito l'intero negativo `-1`. Il risultato viene memorizzato in un `IntegerType` utilizzato come valore di ritorno.

B.1.3 LexemeType: SymbolType

Symcat

Sintassi: (*symcat* $\mathrm{x_1}$ $\mathrm{x_2}$ ... $\mathrm{x_n}$)

Concatena i simboli $x_1, ..., x_n$ restituendo un nuovo `SymbolType`.

B.2 Predicati

B.2.1 NumberType e LexemeType

Ciascun termine dei seguenti predicati deve essere numerico (`IntegerType` o `FloatType`), una lexeme (`StringType` o `SymbolType`), `BooleanType` oppure una variabile di tipo compatibile che verrà sostituita automaticamente se istanziata correttamente.

Equal

Sintassi: (*eq* x_1 x_2 ... x_n)

Confronta i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono tutti uguali, altrimenti `FALSE`.

Not equal

Sintassi: (*neq* x_1 x_2 ... x_n)

Confronta i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se almeno uno dei termini è diverso dagli altri, altrimenti `FALSE`.

B.2.2 NumberType e StringType

Ciascun termine dei seguenti predicati deve essere numerico (`IntegerType` o `FloatType`), una stringa (`StringType`), `BooleanType` oppure una variabile di tipo compatibile che verrà sostituita automaticamente se istanziata correttamente. Eccezioni sono i predicati logici `and`, `or` e `not` che accettano come parametri esclusivamente dei valori booleani e/o altri predicati.

Equal

Sintassi: (*eq* x_1 x_2 ... x_n)

Confronta i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono tutti uguali, altrimenti `FALSE`.

Not equal

Sintassi: (*neq* x_1 x_2 ... x_n)

Confronta i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se almeno uno dei termini è diverso dagli altri, altrimenti `FALSE`.

Less than

Sintassi: (< x_1 x_2 ... x_n)

Verifica l'esistenza di una relazione d'ordine stretto `<` tra i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono in ordine crescente, altrimenti `FALSE`.

Less or equal

Sintassi: (<= x_1 x_2 ... x_n)

Verifica l'esistenza di una relazione d'ordine `<=` tra i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono in ordine crescente, altrimenti `FALSE`.

Greater than

Sintassi: (> x_1 x_2 ... x_n)

Verifica l'esistenza di una relazione d'ordine stretto `>` tra i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono in ordine decrescente, altrimenti `FALSE`.

Greater or equal

Sintassi: (>= x_1 x_2 ... x_n)

Verifica l'esistenza di una relazione d'ordine `>=` tra i termini $x_1, ..., x_n$. Restituisce il valore `TRUE` se i termini sono in ordine decrescente, altrimenti `FALSE`.

Logical and

Sintassi: (*and* x_1 x_2 ... x_n)

Esegue un `and` logico tra i termini $x_1, ..., x_n$[1]. Restituisce il valore `TRUE` se ogni termine ha un valore `TRUE`, altrimenti `FALSE`. I termini, di solito, sono altri predicati.

Logical or

Sintassi: (*or* x_1 x_2 ... x_n)

Esegue un `or` logico tra i termini $x_1, ..., x_n$. Restituisce il valore `FALSE` se ogni termine ha un valore `FALSE`, altrimenti `TRUE`. I termini, di solito, sono altri predicati.

Logical not

Sintassi: (*not* x)

Predicato unario che nega il termine x. Restituisce il valore `TRUE` se il termine è `FALSE`, altrimenti `FALSE`. Il termine, di solito, è un altro predicato.

B.2.3 Predicati Conditional Element

Sono predicati utilizzati nella parte sinistra di una regola per verificare una serie di condizioni.

Test

Sintassi: (*test* x)

Predicato unario che testa una condizione x. Restituisce il valore `TRUE` se la condizione verificata è vera, altrimenti `FALSE`. E' utilizzato come wrapper di predicati su costanti e variabili nella `LHS` di una regola ed è un conditional element.

[1] I predicati *and*, *or* e *not* sono differenti dai predicati *and ce*, *or ce* e *not ce*. Mentre i primi lavorano sui suddetti predicati, questi ultimi riguardano esclusivamente i *pattern ce* per verificare in fase di matching condizioni di esistenza di fatti nella working memory.

Appendice C

Esempi

Sillogismi

```
;;; Sillogismi

;;; FATTI
(deffacts facts
    (alcuni uomini malvagi)
    (tutti uomini mortali)
)

;;; REGOLE

(defrule r1 "Tutti A sono B, Tutti B sono C => Tutti A sono
    C"
  (tutti ?a ?b)
  (tutti ?b ?c)
  =>
  (assert (tutti ?a ?c))
  (printout "Tutti" ?a "sono" ?b)
  (printout "Tutti" ?b "sono" ?c)
  (printout "==>")
  (printout "Tutti" ?a "sono" ?c)
)

(defrule r2 "Tutti A sono B, Alcuni A sono C => Tutti C
   sono B"
```

```
  (tutti ?a ?b)
  (alcuni ?a ?c)
  =>
  (assert (tutti ?c ?b))
  (printout "Tutti" ?a "sono" ?b)
  (printout "Alcuni" ?a "sono" ?c)
  (printout "==>")
  (printout "Tutti" ?c "sono" ?b)
)

(defrule r3 "Tutti A sono B, Nessun C e' B => Nessun C e' A
   "
  (tutti ?a ?b)
  (nessuno ?c ?b)
  =>
  (assert (nessuno ?c ?a))
  (printout "Tutti" ?a "sono" ?b)
  (printout "Nessun" ?c "e'" ?b)
  (printout "==>")
  (printout "Nessun" ?c "e'" ?a)
)
```

Il problema dei missionari e dei cannibali

```
;;; Missionaries and cannibals
;;;
;;; C         cannibals
;;; M         missionaries with M >= C
;;; B         boat
;;; LB        left bank
;;; RB        right bank
;;;
;;; example:
;;; initial state MMMCCCB
;;; goal    state BMMMCCC
;;;
```

```
;;; constraints: not(C>M) on each bank, boat maximum of two
    passengers
;;;
;;; valid rules:
;;; LB      RB
;;; C  -->
;;; CC -->
;;; M  -->
;;; MM -->
;;; MC -->


;;;;;;;;;;;;;;;;;;
;;; DEFGLOBALS ;;;
;;;;;;;;;;;;;;;;;;

(defglobal
  ?*mc* = 150
)


;;;;;;;;;;;;;
;;; FACTS ;;;
;;;;;;;;;;;;;

(deffacts states "stati del problema: configuration LB or
   RB, M, C, B"
  (configuration LB ?*mc* ?*mc* 1)
  (configuration RB 0 0 0)
)
;;;; GOAL --> RB ?*mc* ?*mc* 1


;;;;;;;;;;;;;
;;; RULES ;;;
;;;;;;;;;;;;;

(defrule CtoRB "B C -->"
  ?f<-(configuration LB ?a ?b 1)
```

```
  ?g<-(configuration RB ?c ?d 0)
  (test(>= ?b 1))
  (test (or (>= (- ?c ?d) 1) (and (eq ?c 0)(<= ?d 1))))
  =>
  (printout "B(C) --> RB")
  (retract ?f)
  (retract ?g)
  (bind ?x (- ?b 1))
  (bind ?y (+ ?d 1))
  (assert (configuration LB ?a ?x 0 ))
  (assert (configuration RB ?c ?y 1 ))
  (printout " Bank LB: Missionaries=" ?a " Cannibals="?x)
  (printout " Bank RB: Missionaries=" ?c " Cannibals="?y)
)


(defrule CCtoRB "B CC -->"
  (declare (salience 1000))
  ?f<-(configuration LB ?a ?b 1)
  ?g<-(configuration RB ?c ?d 0)
  (test (>= ?b 2))
  (test (or (>= (- ?c ?d) 1)
            (and (eq ?c 0)(eq ?d 0))))
  =>
  (printout "B(C,C) --> RB")
  (retract ?f)
  (retract ?g)
  (bind ?x (- ?b 2))
  (bind ?y (+ ?d 2))
  (assert (configuration LB ?a ?x 0 ))
  (assert (configuration RB ?c ?y 1 ))
  (printout " Bank LB: Missionaries=" ?a " Cannibals="?x)
  (printout " Bank RB: Missionaries=" ?c " Cannibals="?y)
)


(defrule MtoRB "B M -->"
  ?f<-(configuration LB ?a ?b 1)
  ?g<-(configuration RB ?c ?d 0)
```

```
  (test(>= ?a 1))
  (test (or (<= ?b 1) (>= (- ?a ?b) 1)))
  (test (or (eq ?d 0) (>= (- ?c ?d) -1)))
  =>
  (printout "B(M) --> RB")
  (retract ?f)
  (retract ?g)
  (bind ?x (- ?a 1))
  (bind ?y (+ ?c 1))
  (assert (configuration LB ?x ?b 0 ))
  (assert (configuration RB ?y ?d 1 ))
  (printout " Bank LB: Missionaries=" ?x " Cannibals="?b)
  (printout " Bank RB: Missionaries=" ?y " Cannibals="?d)
)


(defrule MMtoRB "B MM -->"
  (declare (salience 2000))
  ?f<-(configuration LB ?a ?b 1)
  ?g<-(configuration RB ?c ?d 0)
  (test(>=  ?a 2))
  (test (or (<= ?b 2)(>= (- ?a ?b) 2)))
  (test (or (eq ?d 0) (>= (- ?c ?d) -2)))
  =>
  (printout "B(M,M) --> RB")
  (retract ?f)
  (retract ?g)
  (bind ?x (- ?a 2))
  (bind ?y (+ ?c 2))
  (assert (configuration LB ?x ?b 0))
  (assert (configuration RB ?y ?d 1))
  (printout " Bank LB: Missionaries=" ?x " Cannibals="?b)
  (printout " Bank RB: Missionaries=" ?y " Cannibals="?d)
)


(defrule MCtoRB "B MC -->"
  (declare (salience 10))
  ?f<-(configuration LB ?a ?b 1)
```

```
  ?g<-(configuration RB ?c ?d 0)
  (test(> ?a 0))
  (test(> ?b 0))
  (test(eq ?a ?b))
  =>
  (printout "B(M,C) --> RB")
  (retract ?f)
  (retract ?g)
  (bind ?z (- ?a 1))
  (bind ?k (- ?b 1))
  (bind ?x (+ ?c 1))
  (bind ?y (+ ?d 1))
  (assert (configuration LB ?z ?k 0))
  (assert (configuration RB ?x ?y 1))
  (printout " Bank LB: Missionaries=" ?z " Cannibals="?k)
  (printout " Bank RB: Missionaries=" ?x " Cannibals="?y)
)


(defrule backEmpty " <-- B "
  ?g<-(configuration RB ?c ?d 1)
  ?f<-(configuration LB ?a ?b 0)
  (test(or (neq ?a 0) (neq ?b 0)))
  =>
  (printout "LB <-- B()")
  (retract ?g)
  (retract ?f)
  (assert(configuration RB ?c ?d  0))
  (assert(configuration LB ?a ?b  1))
)


(defrule stop
  ?f<-(configuration LB 0 0 0)
  ?g<-(configuration RB ?*mc* ?*mc* 1)
  =>
  (printout "GOAL!")
  (retract ?f)
  (retract ?g)
```

```
  (assert (configuration RB ?*mc* ?*mc* 1))
  (printout " Bank LB: Missionaries=0" " Cannibals=0")
  (printout " Bank RB: Missionaries=" ?*mc* " Cannibals="
     ?*mc*)
  (printout "END")
)
```

Relazioni di parentela

```
;;; Relazioni di parentela

(deffacts famiglia "alcune relazioni date"
  (padre P Q)
  (padre P V)
  (madre K Q)
  (madre K V)
  (sesso V maschio)
  (sesso Q femmina)

  (padre A Q)
  (padre A B)
  (madre C Q)
  (madre C B)
  (sesso B maschio)

  (padre D J)
  (padre D E)
  (madre F J)
  (madre F E)
  (sesso E maschio)

  (padre G J)
  (padre G H)
  (madre I J)
  (madre I H)
  (sesso H maschio)

  (padre L Q)
```

```
  (padre L M)
  (madre N Q)
  (madre N M)
  (sesso M maschio)

  (padre R P)
  (sesso R maschio)
)


(defrule fratello
  (sesso ?x maschio)
  (padre ?y ?x)
  (madre ?k ?x)
  (padre ?y ?z)
  (madre ?k ?z)
  (test (neq ?z ?x))
  =>
  (assert (fratello ?z ?x))
)

(defrule sorella
  (sesso ?x femmina)
  (padre ?y ?x)
  (madre ?k ?x)
  (padre ?y ?z)
  (madre ?k ?z)
  (test (neq ?z ?x))
  =>
  (assert (sorella ?z ?x))
)

(defrule nonno
  (sesso ?x maschio)
  (padre ?x ?y)
  (padre ?y ?z)
  =>
  (assert (nonno ?x ?z))
)
```

```
(defrule nonna
  (sesso ?x femmina)
  (padre ?x ?y)
  (padre ?y ?z)
  =>
  (assert (nonno ?x ?z))
)

(defrule zio
  (sesso ?x maschio)
  (fratello ?x ?y)
  (padre ?y ?z)
  =>
  (assert (zio ?x ?z))
)

(defrule zio1
  (sesso ?x maschio)
  (fratello ?x ?y)
  (madre ?y ?z)
  =>
  (assert (zio ?x ?z))
)

(defrule zia
  (sesso ?x femmina)
  (sorella ?x ?y)
  (madre ?y ?z)
  =>
  (assert (zia ?x ?z))
)

(defrule zia1
  (sesso ?x femmina)
  (sorella ?x ?y)
  (padre ?y ?z)
  =>
  (assert (zia ?x ?z))
```

```
)

(defrule stampa_fratello
  (fratello ?y ?x)
  =>
  (printout  ?y "ha un fratello che si chiama" ?x)
)

(defrule stampa_sorella
  (sorella ?y ?x)
  =>
  (printout  ?y "ha una sorella che si chiama" ?x)
)

(defrule stampa_nonno
  (nonno ?x ?y)
  =>
  (printout  ?x "e' nonno di" ?y)
)

(defrule stampa_nonna
  (nonna ?x ?y)
  =>
  (printout  ?x "e' nonna di" ?y)
)

(defrule stampa_zio
  (zio ?x ?y)
  =>
  (printout  ?x "e' zio di" ?y)
)

(defrule stampa_zia
  (zia ?x ?y)
  =>
  (printout  ?x "e' zia di" ?y)
)
```

Interazioni in una rete sociale

```
;Una rete sociale (social network) consiste di un gruppo
     di persone connesse tra loro da diversi legami
   sociali. Si pensi ad esempio ad una rete
   dipartimentale in cui esiste una relazione fra due
   individui se si scambiano email. Le relazioni sono
   bidirezionali. Come ipotesi semplificativa si
   consideri una rete sociale consistente di k individui
   connessi tra loro da una sola relazione r.

;Rappresentare una semplice rete sociale ed individuare il
   numero dei seguenti motif:

; M1) A e' collegato con B, e B e' collegato con C
; M2) A e' collegato con B, B e' collegato con C, e C e'
   collegato con A
; M3) A, B e C sono collegati tra loro in modo
   bidirezionale
; M4) A e' collegato con B, e C e' collegato con B



(deffacts network "Una rete di relazioni r tra k individui"
  (r mario anna)
  (r anna mario)
  (r pippo franco)
  (r franco gianni)
  (r gianni pippo)
  (r gianni franco)
  (r pippo gianni)
  (r franco pippo)
)

(defglobal
  ?*m1* = 0
  ?*m2* = 0
  ?*m3* = 0
  ?*m4* = 0
```

```
)

(defrule M1
  (r ?a ?b)
  (r ?b ?c)
  (test (neq ?a ?b ?c))
  =>
  (bind ?*m1* (+ ?*m1* 1))
)

(defrule M2
  (r ?a ?b)
  (r ?b ?c)
  (r ?c ?a)
  (test (neq ?a ?b ?c))
  =>
  (bind ?*m2* (+ ?*m2* 1))
)

(defrule M3
  (r ?a ?b)
  (r ?b ?a)
  (r ?b ?c)
  (r ?c ?b)
  (r ?c ?a)
  (r ?a ?c)
  (test (neq ?a ?b ?c))
  =>
  (bind ?*m3* (+ ?*m3* 1))
)

(defrule M4
  (r ?a ?b)
  (r ?c ?b)
  (test (neq ?a ?b ?c))
  =>
  (bind ?*m4* (+ ?*m4* 1))
)
```

Il mondo dei blocchi (con piano senza limitazioni)

```
; Si vuole implementare un sistema a regole per la
; risoluzione di un problema del mondo dei blocchi.
; Supponiamo di avere la seguente configurazione iniziale
;
;  A  D
;  B  E
;  C  F
;
; e di voler mettere il blocco C su blocco F.
; *********************************************************

(defrule sposta "Sposta un blocco libero su un altro blocco
    libero"
   ?obiettivo <- (obiettivo sposta ?blocco1 sopra-a ?
      blocco2)
   (blocco ?blocco1)
   (blocco ?blocco2)
   (sopra niente sotto ?blocco1)
   ?pila1 <- (sopra ?blocco1 sotto ?blocco3)
   ?pila2 <- (sopra niente sotto ?blocco2)
   =>
   (retract ?obiettivo)
   (retract ?pila1)
   (retract ?pila2)
   (assert (sopra ?blocco1 sotto ?blocco2))
   (assert (sopra niente sotto ?blocco3))
   (printout "Sposta " ?blocco1 " sopra a " ?blocco2 "."))

(defrule sposta-sul-piano "Sposta un blocco libero sul
   piano"
   ?obiettivo <- (obiettivo sposta ?blocco1 sopra-a piano)
   (blocco ?blocco1)
   (sopra niente sotto ?blocco1)
   ?pila <- (sopra ?blocco1 sotto ?blocco2)
   =>
   (retract ?obiettivo)
   (retract  ?pila)
   (assert (sopra ?blocco1 sotto piano))
```

```
   (assert (sopra niente sotto ?blocco2))
   (printout "Sposta " ?blocco1 " sul piano."))

(defrule libera-blocco-partenza
        "Stabilisce l'obiettivo di liberare il blocco di
           partenza"
   (obiettivo sposta ?blocco1 sopra-a ?wild)
   (blocco ?blocco1)
   (sopra ?blocco2 sotto ?blocco1)
   (blocco ?blocco2)
   =>
   (assert (obiettivo sposta ?blocco2 sopra-a piano)))

(defrule libera-blocco-arrivo
         "Stabilisce l'obiettivo di liberare il blocco d'
            arrivo"
   (obiettivo sposta ?wildcard sopra-a ?blocco1)
   (blocco ?blocco1)
   (sopra ?blocco2 sotto ?blocco1)
   (blocco ?blocco2)
   =>
   (assert (obiettivo sposta ?blocco2 sopra-a piano)))

(deffacts stato-iniziale "A/B/C, D/E/F, e vogliamo mettere
   C su F."
   (blocco A)
   (blocco B)
   (blocco C)
   (blocco D)
   (blocco E)
   (blocco F)
   (sopra niente sotto A)
   (sopra A sotto B)
   (sopra B sotto C)
   (sopra C sotto piano)
   (sopra niente sotto D)
   (sopra D sotto E)
   (sopra E sotto F)
   (sopra F sotto piano)
```

```
   (obiettivo sposta C sopra-a F))
```

Il gioco dell'otto senza euristica

```
(deffacts stato-iniziale
        (obiettivo non-raggiunto)

        (cella 1 1 2)
        (cella 1 2 8)
        (cella 1 3 1)
        (cella 2 1 0)
        (cella 2 2 4)
        (cella 2 3 3)
        (cella 3 1 7)
        (cella 3 2 6)
        (cella 3 3 5)
)

(defrule sposta-vuoto-sinistra
        (obiettivo non-raggiunto)
        ?c1 <- (cella ?x ?y 0)
        (test (> ?y 1))
        ?c2 <- (cella ?x ?k ?v)
        (test (eq ?k (- ?y 1)))
        =>

        (retract ?c1 ?c2)
        (assert (cella ?x ?k 0))
        (assert (cella ?x ?y ?v))
)

(defrule sposta-vuoto-destra
        (obiettivo non-raggiunto)
        ?c1 <- (cella ?x ?y 0)
        (test (< ?y 3))
        ?c2 <- (cella ?x ?k ?v)
        (test (eq ?k (+ ?y 1)))
        =>
```

```
        (retract ?c1 ?c2)
        (assert (cella ?x ?k 0))
        (assert (cella ?x ?y ?v))
)

(defrule sposta-vuoto-su
        (obiettivo non-raggiunto)
        ?c1 <- (cella ?x ?y 0)
        (test (> ?x 1))
        ?c2 <- (cella ?k ?y ?v)
        (test (eq ?k (- ?x 1)))
        =>

        (retract ?c1 ?c2)
        (assert (cella ?k ?y 0))
        (assert (cella ?x ?y ?v))
)

(defrule sposta-vuoto-giu
        (obiettivo non-raggiunto)
        ?c1 <- (cella ?x ?y 0)
        (test (< ?x 3))
        ?c2 <- (cella ?k ?y ?v)
        (test (eq ?k (+ ?x 1)))
        =>

        (retract ?c1 ?c2)
        (assert (cella ?k ?y 0))
        (assert (cella ?x ?y ?v))
)

(defrule termina-gioco
        (declare (salience 10000))
        ?ob <- (obiettivo non-raggiunto)
        (cella 1 1 1)
        (cella 1 2 2)
        (cella 1 3 3)
        (cella 2 1 8)
```

```
        (cella 2 2 0)
        (cella 2 3 4)
        (cella 3 1 7)
        (cella 3 2 6)
        (cella 3 3 5)
        =>
        (printout "Obiettivo raggiunto!")
        (retract ?ob)
)
```

Appendice D

Note di installazione

E' obbligatoria l'installazione del pacchetto `pyparsing` versione 1.5.7.

Per sfruttare interamente TURTLE, sono necessari pacchetti aggiuntivi, installabili tramite i tool `pip` o `easy_install`:

- `colorama`
- `termcolor`
- `networkx`
- `pydot`

E' necessaria, inoltre, l'installazione del tool Graphviz per visualizzare graficamente la rete.

Microsoft Windows®

Per installare le dipendenze è consigliabile l'installazione di setuptools eseguendo con python il file al seguente link: `https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py`.

Al termine dell'installazione sarà possibile utilizzare il tool `easy_install` per poter aggiungere i pacchetti richiesti da TURTLE.

Graphviz è disponibile all'indirizzo `http://graphviz.org/Download_windows.php`.

GNU/Linux e Unix®

Di solito è già presente il tool `pip` per installare i pacchetti aggiuntivi. In caso contrario, è possibile effettuare l'installazione di `pip` o `easy_install` utilizzando il package manager del sistema.

Graphviz è disponibile per il download diretto all'indirizzo `http://graphviz.org/Download..php` oppure l'installazione può essere effettuata attraverso il package manager del sistema.

Bibliografia

[aNP09] Russell Stuart Norvig Peter. *Artificial Intelligence: A Modern Approach*, volume 1. Prentice Hall, 2009.

[Doo95] Robert B Doorenbos. *Production matching for large learning systems*. PhD thesis, University of Southern California, 1995.

[For82] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[Ing] Giorgio Ingargiola. CIS587: The RETE Algorithm. `http://www.cis.temple.edu/~giorgio/cis587/readings/rete.html`.

[Jac98] Peter Jackson. *Introduction to Expert Systems International Computer Science Series*. Addison-Wesley Pub Co, ISBN, 1998.

[Min90] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2):363–391, 1990.

Made with LaTeX