UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

ANSWER SET PROGRAMMING IN AMBIENTE DI GIOCO INTERATTIVO

Relatore: Prof. Mario ORNAGHI
Correlatore: Prof. Alessandro PROVETTI

Tesi di Laurea di:
Luca Padovani
Matr. Nr. 545460

ANNO ACCADEMICO 2002-2003

a Elena, Diego, Carlo.

Indice

Indice v

Abstract viii

Ringraziamenti xii

1 Introduzione 1
- 1.1 Il ragionamento per default e le sue applicazioni 1
- 1.2 L'efficienza del ragionamento 1
- 1.3 Benchmarking: randomizzazione e istanze tipiche 1

2 L'Answer Set Programming 2
- 2.1 Sintassi 2
- 2.2 Semantica 5
  - 2.2.1 Non monotonia 6
  - 2.2.2 Programmi con negazione esplicita 7
- 2.3 Answer Set Programming 8
- 2.4 Esempi 8
- 2.5 Complessità del calcolo 12
- 2.6 Risolutori 13
  - 2.6.1 SMODELS 14
  - 2.6.2 LPARSE 16
  - 2.6.3 CSMODELS 18
  - 2.6.4 CMODELS 19
  - 2.6.5 DLV 21

3 Quake 3 Arena 23
- 3.1 Introduzione 23
- 3.2 Architettura 25

3.3 Collision Detection 27
3.3.1 Binary Space Partitioning 27
3.4 I BOT 28
3.4.1 Modello cognitivo 29
3.4.2 Architettura a layer 30
3.4.3 Area awareness system 31
3.4.4 Routing 32
3.4.5 Obiettivi 33
3.4.6 Combattimento 34
3.4.7 AI Network 36

4 Architettura di Qsmodels 40
4.1 Introduzione 41
4.2 L'architettura 42
4.3 Ciclo di esecuzione 43
4.3.1 Sensing 44
4.3.2 Emergenze 46
4.3.3 Richiesta del piano 48
4.3.4 Validità del piano 49
4.3.5 Esecuzione del piano 50
4.3.6 Intelligenza di basso livello 51
4.4 Stati del programma 52
4.5 QsmodelsServer 54
4.6 Comunicazione 54
4.7 Conclusioni 55

5 Il Pianificatore automatico 56
5.1 Modello Cognitivo 57
5.1.1 Il mondo 57
5.1.2 L'agente 58
5.1.3 Il nemico 60
5.2 Struttura 61
5.3 Il preambolo 61
5.4 I Fluenti 64
5.5 Le Azioni 66
5.5.1 La Dichiarazione 66
5.5.2 Le Precondizioni 68
5.5.3 Le Conseguenze 69
5.6 La generazione del piano 69

6 L'implementazione di Qsmodels **73**
6.1 L'interazione con Quake 3 Arena 74
6.1.1 Le aree di memoria 75
6.1.2 Le funzioni di sistema 77
6.1.3 Le funzioni di utilità 78

A Codice del pianificatore **79**

Glossario **91**

Bibliografia **92**

Abstract

In questa tesi si vuole dimostrare la possibilità di utilizzare l'*Answer Set Programming* [2] (*ASP*) in un ambiente interattivo integrandolo nel videogioco *Quake 3 Arena*; in tal modo potremo utilizzare la grafica e l'interazione di questo popolare gioco per sperimentare teorie logiche dell'azione espresse attraverso l'*Answer Set Programming.*

Quake 3 Arena (*Q3A*) è stato realizzato dalla *id Software* e porzioni rilevanti del codice sono ora pubbliche. *Q3A* è un gioco "in prima persona": l'obiettivo è uccidere i propri avversari utilizzando armi e bonus che si trovano all'interno dell'ambiente di gioco. Le "intelligenze artificiali" presenti in *Q3A*, chiamate *BOT*, si comportano seguendo metodi di ragionamento reattivi, così come la maggior parte dei videogiochi attualmente in commercio: fissato il goal di lungo termine reagiscono agli eventi esterni secondo schemi prefissati. Questo comportamento svela il metodo usato per realizzarli: *macchine a stati finiti* (*FSM*) realizzate da programmatori esperti di videogiochi.

Lo schema da noi proposto segue il ciclo proposto da [**?**]

1. *Osservazione*

2. *Scelta del goal*

3. *Pianificazione*

4. *Esecuzione del piano*

L'architettura prevede l'utilizzo di due layer: uno di alto livello responsabile della pianificazione a medio e lungo termine ed uno di basso livello (reattivo) responsabile

dell'esecuzione del piano e della gestione delle situazioni d'emergenza. L'intelligenza di alto livello è realizzata tramite un pianificatore automatico. La generazione del piano è ottenuta tramite l'interpretazione di un programma *Answer Set Programming.*

L'idea di base dell'*Answer Set Programming,* a partire dal lavoro Gelfond e Lifschitz [2], è di rappresentare un problema computazionale con un programma logico i cui *Answer Set* (insiemi delle risposte) rappresentino le soluzioni dello stesso; per trovare queste soluzioni si utilizza un motore inferenziale quale `smodels`, sviluppato dalla Helskinki University of Technology o `DLV`, sviluppato congiuntamente dalla Vienna University of Technology e dall'Università della Calabria. Questi risolutori sono oggi in grado di interpretare programmi Answer Set con migliaia di atomi e regole in pochi secondi.

Nel nostro progetto, `smodels` (che viene applicato in coppia con il grounder `lparse`, anch'esso sviluppato ad Helsinki) calcola gli Answer Sets di un programma logico contenente la descrizione *astratta* dei piani possibili, seguendo lo schema di codifica di Baral [?]. Il modello (Answer Set) calcolato viene passato al layer sottostante che provvede ad eseguirlo. Le circostanze d'emergenza vengono trattate reattivamente ed al livello basso dell'architettura.

Il modello presentato mostra notevoli vantaggi rispetto agli approcci normalmente usati nelle `AI` dei videogiochi: maggiore semplicità realizzativa, maggior livello di astrazione; quindi un comportamento più *intelligente*. La maggiore semplicità realizzativa è frutto dalla separazione netta che si può operare tra le regole del planner vero e proprio e le regole che governano il mondo: quest'ultima parte può essere scritta anche da non esperti di `AI`.

Questa tesi ha come ulteriore scopo quello di testare il risolutore `smodels` in ambiti vicini alle applicazioni ed al real-time: abbiamo dimostrato che con taluni accorgimenti d'utilizzo e nella realizzazione del pianificatore è possibile utilizzare `smodels` e `lparse` in contesti interattivi; ciononostante, si è visto che maggiore potenza di calcolo è necessaria per un'esecuzione completamente *real-time*; l'utilizzo pratico *immediato* che se ne può avere è invece la realizzazione di prototipi.

Il Software sviluppato in questa tesi permetterà la realizzazione di laboratori virtuali di Intelligenza artificiale; l'utilizzo di un videogioco come base d'azione rende possibile decidere il livello di astrazione che si vuole dare al modello del mondo fisico ed al tempo stesso permette di avere un feedback visivo dei risultati della propria ricerca.

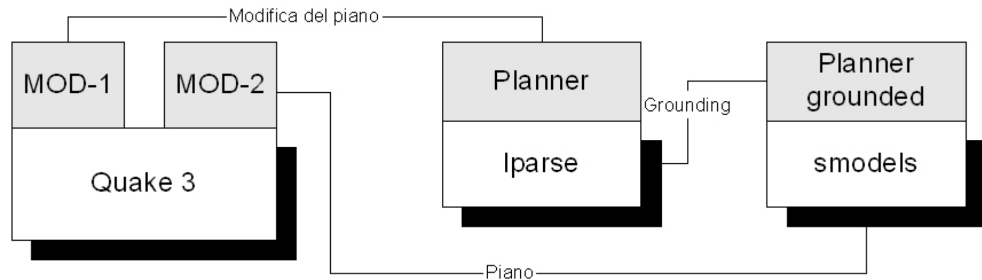

Figura 1: Schema di esecuzione

La parte ad alto livello che abbiamo sviluppato, realizza sostanzialmente un'automa per *Q3A* che, a partire da una conoscenza del mondo data (assimilabile a quella di un giocatore medio dopo le prime tre o quattro partite), tenta di sconfiggere i propri avversari, adottando strategie che gli permettano di affrontare il nemico sempre in condizioni ottimali. Nella prosecuzione di questo lavoro di tesi si valuterà la possibilità di aggiungere un modulo di *learning* che permetta di apprendere e sfruttare alcune regolarità nella strategia dell'avversario.

Il software che è stato qui sviluppato può essere diviso in tre parti: i) programma Answer Set `planner`, realizzato per `smodels`, ii) l'interfacciamento a *Q3A* realizzato in *C++* che si occupa del *sensing* e dell'esecuzione del piano, iii) l'*AI* di basso livello realizzata anche questa in *C++*.

Il ciclo di esecuzione è visibile in Figura 1, prevede una prima fase di *sensing* effettuata dal modulo `MOD-1` ispezionando aree di memoria del processo *Q3A*; le informazioni estratte (posizione nel mondo, stato di salute, bonus, ecc.) vengono convertite in *fluenti* da inserire all'interno del *planner*. Il piano viene generato usando

`smodels`; quindi `MOD-2` si occupa di estrarre il piano dall'Answer Set calcolato il risultato e di mandarlo in esecuzione effettuando delle *trap* (muovi, spara, raccogli, ecc.) a *Q3A*.

Ringraziamenti

Pare che sia finalmente giunto alla fine, non mi resta che ringraziare tutte quelle persone che in questo ultimo anno, abbondante a dire il vero, hanno contribuito in vari modi alla realizzazione di questo lavoro di tesi.

Ringrazio il Prof. Alessandro Provetti per l'estrema disponibilità dimostratami in ogni fase del progetto, dall'ideazione del progetto alla redazione di questa tesi.

Un ringraziamento particolare va ovviamente alla mia famiglia che in tutti questi anni mi ha sempre sostenuto nei vari alti e bassi che si sono susseguiti.

Gli amici da ringraziare sono tanti, mi rivolgo soprattutto a quelli che con la loro insistente domanda: "Quand'è che ti laurei?!?" mi facevano *discretamente* capire che era il caso di portare a termine questo progetto.

Ringrazio infine il Prof. Gianni Degli Antoni per aver tollerato alcuni miei errori di gioventù.

Milano, 2 febbraio 2004

Luca Padovani

Capitolo 1

Introduzione

L'obiettivo che questo lavoro di tesi si propone è duplice: da una parte, vogliamo creare e rendere disponibili nuovi programmi benchmark per la valutazione delle prestazioni di diversi risolutori Answer Set, dall'altra intendiamo misurare tali prestazioni per ottenere una visione d'insieme della qualità dei risolutori attualmente disponibili.

1.1 Il ragionamento per default e le sue applicazioni

1.2 L'efficienza del ragionamento

1.3 Benchmarking: randomizzazione e istanze tipiche

Capitolo 2

L'Answer Set Programming

L'Answer Set Programming è una forma di programmazione logica basata sulla semantica dei modelli stabili introdotta nel 1988 da [2], successivamente estesa e rinominata in semantica Answer Set [3]. L'espressività della sintassi e l'efficienza dei risolutori hanno permesso all'Answer Set Programming di occupare un posto sempre più rilevante nel campo dell'Intelligenza Artificiale; in particolare, essa viene sempre più frequentemente impiegata per codificare problemi di planning, diagnosi e, più in generale, problemi di rappresentazione della conoscenza.

Di seguito presentiamo una panoramica dell'Answer Set Programming, descrivendone la sintassi, la semantica e le maggiori implementazioni esistenti.

2.1 Sintassi

Sia $\mathcal{L}$ un linguaggio di costanti, predicati costanti, termini e atomi, dove termini e atomi siano costruiti come nel corrispondente linguaggio della logica del primo ordine.

Considereremo esclusivamente teorie (programmi answer set) *pseudoproposizionali*, cioè teorie che ammettono la sostituzione iterativa di variabili con simboli di

costante. Cioè, $\mathcal{L}$ sarà, a differenza della logica classica e della programmazione logica standard, privo di simboli di funzione.

Una *regola* è un'espressione della forma:

$$\rho : A_0 \leftarrow A_1, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n \tag{2.1.1}$$

dove $A_1, \ldots, A_m \in \mathcal{L}$ sono atomi e *not* è un connettivo chiamato *negazione da fallimento.*

Per una data regola ρ, definiamo

- $head(\rho) = A_0$,
- $pos(\rho) = A_1, \ldots, A_m$,
- $neg(\rho) = A_{m+1}, \ldots, A_n$,
- $body(\rho) = pos(\rho) \cup neg(\rho)$.

La testa delle regole non è mai vuota, mentre, se $body(\rho) = \emptyset$, allora ρ è un *fatto.*

Un programma logico è una collezione di regole. Una regola con variabili può essere interpretata come l'espressione sintetica dell'insieme di tutte le sue istanziazioni. Normalmente, il linguaggio $\mathcal{L}$ rimarrà implicitamente definito dal programma stesso; l'insieme di tutti gli atomi *ground*, cioè privi di variabili, che possiamo formare con il linguaggio $\mathcal{L}$ associato al programma Π sarà denotato con B_Π (la base di Herbrand di Π).

Le *query* ed i *vincoli* sono espressioni con la stessa struttura delle regole ma con la testa vuota ($head(\rho) = \emptyset$).

Intuitivamente, un vincolo come

```
:- body.
```

impone che `body` non possa essere vero in alcun modello stabile del programma cui il vincolo appartiene. L'effetto dell'applicazione di un vincolo su un programma, in altre parole, è l'invalidazione di alcuni degli answer set del programma di partenza. Il vincolo sopra riportato corrisponde alla regola:

```
x :- not x, body.
```

dove `x` è un atomo *fresco*, cioè non ancora apparso nel programma. Affinché la regola sopra riportata non derivi una contraddizione, è necessario che `body` risulti falso. Per questo motivo, il vincolo `:- body` può essere interpretato come "necessità che `body` sia falso".

Esempio 2.1.1. Sia π_1 il seguente programma:
$a :- \textit{not}\ b.$
$b :- \textit{not}\ a.$
π_1 ha due answer set: $\{a\}$ e $\{b\}$. Supponiamo ora di aggiungere a π_1 il seguente vincolo v:
$:- \textit{not}\ a,\ c.$
Dal momento che c risulta sempre falso in π_1, il vincolo sarà sempre soddisfatto e non porterà all'invalidazione di alcun answer set.
Sia $\pi_2 = \pi_1 \cup \{c\}$. L'applicazione del vincolo v a π_2 provoca la generazione di un solo modello stabile, $\{a,\ c\}$, mentre l'answer set $\{b\}$ di π_1 non è più tale per π_2. Infatti, *not* a e c non possono essere veri *nello stesso answer set*; ciò implica che *not* a deve risultare falso (dal momento che c è un fatto di π_2 ed è quindi sempre vero), e dalla sua falsità, per la struttura di π_1, discende l'impossibilità di derivare b.

2.2 Semantica

Intuitivamente, un programma descrive, attraverso le regole che lo governano, un *mondo* (dominio del problema). A seconda di quali assunzioni prendiamo per vere, otterremo una diversa *"visione del mondo"*. Tale visione è categorica: certi fatti saranno veri e tutti gli altri, invece, falsi. Per ogni mondo, è possibile avere molteplici diverse visioni: ciascuna di esse è un *modello stabile* del programma che codifica quel particolare mondo.

Per prima cosa, definiamo i modelli stabili/answer set della sottoclasse dei programmi positivi. Un programma positivo è un programma in cui, per ogni regola ρ, $neg(\rho) = \emptyset$.

Definizione 2.2.1. (Modelli stabili di programmi positivi)
Il modello stabile $a(\Pi)$ di un programma positivo Π è il più piccolo sottoinsieme di B_Π tale che per ogni regola (2.1.1) in Π:

$$A_1, \ldots, A_m \in a(\Pi) \Rightarrow A_0 \in a(\Pi) \tag{2.2.1}$$

Chiaramente, i programmi positivi hanno un unico modello stabile, coincidente con il modello che si ottiene applicando altre semantiche della programmazione logica[1]; in altre parole, i programmi positivi sono *categorici* (hanno cioè un solo modello).

Il concetto di modello stabile è collegato a quello di modello minimale (e, nel caso positivo, minimo) della logica classica.

Definizione 2.2.2. (Modelli stabili di programmi)
Sia Π un programma logico. Per ogni insieme S di atomi, sia $\Gamma(\Pi, S)$ un programma ottenuto da Π eliminando

[1] Grazie a questa coincidenza, possiamo, per esempio, ottenere il modello stabile di un programma positivo come *punto fisso* dell'operatore di *conseguenza logica immediata* T_Π iterato a partire da $\emptyset$.

(I) ogni regola che abbia una formula 'not A' nel suo corpo, con $A \in S$;

(II) tutte le formule della forma 'not A' nel corpo delle rimanenti regole.

Per quanto sopra, il programma $\Gamma(\Pi, S)$ non contiene 'not', perciò il suo modello stabile è già definito in Definizione 2.2.1. Se tale modello stabile coincide con S, l'insieme di atomi da cui siamo partiti, allora diremo che S *è un modello stabile di* Π. In altre parole, un modello stabile di Π è caratterizzato dall'equazione:

$$S = a(\Gamma(\Pi, S)). \tag{2.2.2}$$

Definiamo ora l'implicazione logica ($\models$) nella semantica dei modelli stabili. Un atomo *ground* α è *vero* in S se $\alpha \in S$, *falso* altrimenti, ovvero $\neg\,\alpha$ è vero in S. Diciamo che ϕ *è un teorema di* Π (scritto $\Pi \models_{sm} \phi$) se ϕ è vera in tutti i modelli stabili di Π.

Diremo che la risposta alla query γ è

si se γ è vera in tutti i modelli stabili di Π, cioè $\Pi \models_{sm} \gamma$

no se $\neg\gamma$ è vera in tutti i modelli stabili di Π, cioè $\Pi \models_{sm} \neg\gamma$

indefinita altrimenti

2.2.1 Non monotonia

È importante notare che i programmi logici sono non-monotoni rispetto all'aggiornamento: l'aggiunta di nuove informazioni, siano esse fatti o regole, al programma, può invalidare precedenti conclusioni.

Esempio 2.2.1. Sia π_1 il seguente programma:

$a.$

$b \leftarrow a.$

$c \leftarrow not\ d.$

π_1 ha un unico answer set, $S = \{a, b, c\}$.

Supponiamo ora che, in seguito ad una nuova osservazione sul mondo descritto da π_1, venga aggiunto al programma il seguente fatto: d.

Non sarà più possibile inferire c, perciò l'unico answer set di π_1 sarà $S' = \{a, b, d\}$.

2.2.2 Programmi con negazione esplicita

Gelfond e Lifschitz hanno introdotto in [3], a partire dalla semantica dei modelli stabili, la classe dei *programmi estesi*, programmi i cui atomi possono apparire preceduti dalla negazione esplicita ($\neg A$). Un atomo e la sua negazione esplicita sono detti *letterali*.

Per la classe dei programmi estesi è definita la semantica answer set, essenzialmente identica alla semantica dei modelli stabili, eccetto che per il seguente aspetto: se un answer set contiene contemporaneamente A e $\neg A$, allora contiene tutti i letterali.

Corollario 2.2.1. *(Gelfond e Lifschitz) Se un programma logico esteso ha un answer set inconsistente, allora tale answer set è unico.*

Per programmi senza negazione esplicita il concetto di modello stabile coincide con quello di answer set, perciò nel seguito i due termini verranno utilizzati indifferentemente. Vale il seguente

Lemma 2.2.2. *(Marek e Subramanian) Per ogni answer set A di un programma logico esteso* Π*:*

- *Per ogni istanza ground di una regola del tipo*

$$\rho : L_0 \leftarrow L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n \tag{2.2.3}$$

del programma Π*, se*

$$\{L_1, \ldots, L_m\} \subseteq A \text{ e } \{L_{m+1}, \ldots, L_n\} \cap A = 0 \tag{2.2.4}$$

allora $L_0 \in A$*.*

- *Se* A *è un answer set consistente di* Π *e* $L_0 \in A$*, allora esiste un'istanza ground di una regola del tipo* ρ *di* Π *tale che:*

$$\{L_1, \ldots, L_m\} \subseteq A \text{ e } \{L_{m+1}, \ldots, L_n\} \cap A = 0. \tag{2.2.5}$$

2.3 Answer Set Programming

È sulla semantica definita nella sezione precedente che si basa la *"programmazione per insiemi di risposte"* (Answer Set Programming o ASP).

Un *answer set* di un programma Π coincide con un modello stabile di Π e ne rappresenta una delle possibili soluzioni. Diversamente dalla programmazione logica tradizionale, le soluzioni di un problema non sono ottenute per sostituzioni di variabili in risposta ad una query, ma vengono espresse come "insiemi di risposte". Π potrà dunque avere un solo answer set (eventualmente vuoto), molteplici answer set oppure nessun answer set, nel qual caso Π sarà detto *inconsistente*.

2.4 Esempi

Esempio 2.4.1. Consideriamo il problema di assegnare 3 colori (rosso, blu e verde) ai vertici di un grafo in modo che vertici adiacenti abbiano colore diverso. Questo

problema, noto in letteratura come 3COL è *NP-Completo*, cioè intrattabile per input di grandi dimensioni.

Per modellare il problema, dobbiamo, in primo luogo, rappresentare il grafo tramite fatti che ne descrivono i nodi e gli archi ed elencare i colori che utilizzeremo per la sua colorazione:

```
node(0..3).

col(red).
col(blue).
col(green).

edge(0, 1).
edge(1, 2).
edge(2, 0).
edge(1, 3).
edge(2, 3).
```

Questi fatti descrivono il grafo in figura 2.1.

Figura 2.1: Grafo descritto nell'esempio 2.4.1

L'output desiderato è il seguente:

```
{color(0,red), color(1,blue), color(2,green), color(3,red)}
{color(0,red), color(1,green), color(2,blue), color(3,red)}
{color(0,blue), color(1,red), color(2,green), color(3,blue)}
{color(0,blue), color(1,green), color(2,red), color(3,blue)}
{color(0,green), color(1,blue), color(2,red), color(3,green)}
{color(0,green), color(1,red), color(2,blue), color(3,green)}
```

Vediamo ora come costruire un programma logico per cui il calcolo, sia esso costruzione di modelli stabili in ASP o sia esso risposta a query come in Prolog, restituisca esattamente le 3-colorazioni descritte sopra.

Esempio 2.4.2. (3-colorazione in Prolog)

Questo esempio di programma Prolog per la 3COL mostra le differenze fra i due stili di programmazione.

In Prolog, il calcolo corrisponde essenzialmente a delineare una lista di associazioni nodo/colore.

Vedremo, invece, come in ASP ogni modello stabile corrisponda ad un assegnamento che rispetta i vincoli dell'istanza.

```
graph_coloring(G, L):- gr_col(G, [], L).

gr_col([], L, L):- !.
gr_col([N|R], P, L):- vertex_color(N, C, P),
                      gr_col(R, [color(N,C)|P], L).

vertex_color(N, C, P):- col(C), ok_color(N, C, P).

neighbourhood(N, B, L):- (edge(N, N1); edge(N1, N)),
                         member(N1, B), !,
                         neighbourhood(N, [N1|B], L).
neighbourhood(_, L, L).

ok_color(N, C, P):- neighbourhood(N, [], L),
                    check_color(L, C, P).

check_color([], _, _):- !.
check_color([N1|B], C, P):- \ + member(color(N1, C), P),
                            !,
                            check_color(B, C, P).

member(E, [E|X]):- !.
member(E, [_|X]):- member(E, X).
```

Vediamo ora una sessione Prolog tipica:

```
?- consult[3col.pro]
?- graph_coloring([0, 1, 2, 3], L).
L=[color(0,red), color(1,blue), color(2,green), color(3,red)];
L=[color(0,red), color(1,green), color(2,blue), color(3,red)];
...
```

Esempio 2.4.3. (3-colorazione in ASP - continuazione dell'esempio 2.4.1)

Il seguente programma codifica il problema della 3-colorazione in ASP. Il connettivo '|' significa "or".

```
color(X, red) | color(X, blue) | color(X, green) :- node(X).
```

Se X è un nodo del grafo, allora il suo colore sarà rosso, oppure blu, oppure verde.

```
:- edge(X, Y), col(C), color(X, C), color(Y, C).
```

Se esiste un arco tra i nodi X e Y (cioè X e Y sono adiacenti), allora X e Y non possono avere lo stesso colore.

```
hide node(X).
hide edge(X, Y).
hide col(C).
```

Con il comando:

```
lparse 3col.txt | smodels 0
```

otteniamo:

```
Answer1
{color(0,red), color(1,blue), color(2,green), color(3,red)}
Answer2
{color(0,red), color(1,green), color(2,blue), color(3,red)}
...
```

Sebbene i risultati ottenuti siano i medesimi, risulta evidente la differenza tra i due approcci: in particolare, il programma ASP riesce a codificare il problema in maniera molto più sintetica e diretta del corrispondente programma Prolog, grazie soprattutto all'uso di vincoli.

2.5 Complessità del calcolo

I programmi logici nella semantica dei modelli stabili sono, in generale, altamente espressivi: in presenza di simboli di funzione, essi permettono di esprimere gli insiemi iper-aritmetici [4], cioè sono in grado di descrivere funzioni correntemente non computabili in tempi accettabili. Tuttavia, se il linguaggio usato viene ristretto all'ambito proposizionale, escludendo, perciò, i simboli di funzione, l'espressività di tale linguaggio – e quindi la complessità del calcolo – calano drasticamente.
Il problema consistente nel trovare un modello stabile per un programma logico proposizionale è NP-completo.

Teorema 2.5.1. *(da [4]) Dato un programma logico* Π *ground, il problema di decidere se* Π *ammette un modello stabile è NP-completo.*

Dimostrazione. La dimostrazione [5] consta di due passi. La NP-hardness discende dalla riduzione di SAT (il problema della soddisfacibilità di formule booleane) al problema ASP per mezzo di una traduzione abbastanza diretta. La completezza segue per mezzo di un algoritmo non-deterministico di verifica di interpretazioni, oppure per riduzione del problema a SAT. □

In altre parole, sebbene il problema ASP sia intrattabile, la sua complessità non è eccessiva e si situa al secondo livello della gerarchia polinomiale.

La NP-completezza del problema ASP ha varie conseguenze, sia pratiche sia teoriche. In particolare, questo risultato ha permesso al calcolo dei modelli stabili di entrare in *competizione* diretta coi sistemi di *model checking*, basati sulla risoluzione del problema SAT. Esistono in letteratura vari esempi di impiego di un risolutore ASP per il calcolo dei modelli stabili di programmi (attraverso un'opportuna codifica del problema). Per una panoramica recente dello stato dell'arte si veda [**?**].

2.6 Risolutori

Il calcolo dei modelli stabili è oggi un'area di fervida ricerca. Sono attualmente disponibili implementazioni piuttosto efficienti e competitive coi migliori risolutori per SAT, che vengono impiegati anche industrialmente nella verifica di sistemi (model checking).

Queste implementazioni sono in grado di risolvere la versione *esatta* del problema, calcolando tutti i modelli stabili del programma dato: sono cioè orientate all'esplorazione dell'intero spazio delle soluzioni (albero di decisione). D'altra parte, è spesso sufficiente la verifica dell'esistenza di *un* modello, senza esplorazione esaustiva.

Allo stato attuale, le implementazioni più diffuse e conosciute sono SMODELS (sviluppato dalla Helsinki University of Technologies) e DLV (sviluppato da TU Wien e Università della Calabria) [**?**]. Mentre SMODELS è un risolutore per ASP che accetta la sintassi dei programmi logici standard, DLV tratta il sovrainsieme detto dei *Programmi logici disgiuntivi* definito da [3]. Nel frammento proposizionale, i programmi logici disgiuntivi sono strettamente più espressivi di quelli standard: DASP (ASP per programmi disgiuntivi) è un problema completo per il terzo livello della gerarchia polinomiale [1].

Segue una breve descrizione per ciascuno dei risolutori considerati durante questo lavoro di tesi: oltre a SMODELS e DLV, sono stati esaminati CSMODELS e CMODELS [?], entrambi evoluzioni di SMODELS.

2.6.1 SMODELS

SMODELS ([?], [6]) è un noto sistema per l'Answer Set Programming, costituito da `smodels`, cioè l'implementazione vera e propria della semantica dei modelli stabili per programmi logici normali, e `lparse`, un front-end che ha il compito di trasformare i programmi utente in una forma comprensibile per `smodels` (vd. fig. 2.2 – [?]). In questa sottosezione, viene presentata la parte algoritmica del sistema; per una panoramica su `lparse`, si veda la sottosezione 2.6.2. D'ora in avanti, ci riferiremo con il termine SMODELS al motore di programmazione logica che si occupa di calcolare i modelli stabili di un programma.

Figura 2.2: Il sistema SMODELS

La funzione principale di SMODELS si chiama *smodels*; questa funzione prende in ingresso un programma ground Π e un insieme di letterali A e restituisce *true* se esiste un answer set di Π che sia consistente con A, oppure *false* se tale answer set non esiste. La funzione *smodels* invoca altre tre importanti funzioni:

- *expand(*Π*, A)*: il suo obiettivo è estendere A il più possibile (e il più efficientemente possibile) in modo che tutti gli answer set di Π consistenti con A siano anche consistenti con *expand(*Π*, A)*;

- *lookahead(*Π*, A)*: ha il compito di individuare rapidamente inconsistenze, in modo da evitare l'esplorazione di ampi rami dell'albero di ricerca che terminino con un'inconsistenza;

- *heuristic(*Π*, A)*: si occupa di assumere la verità di uno dei restanti letterali; dato l'atomo x, verrà scelto il letterale, tra x e *not* x, che può portare ad una maggiore espansione.

L'output prodotto da SMODELS fornisce informazioni sul processo di calcolo degli answer set. Vediamone un esempio:

```
smodels version 2.27. Reading...done
Answer: 1
...
True
Duration: 1.700
Number of choice points: 9
Number of wrong choices: 0
Number of atoms: 625
Number of rules: 20080
Number of picked atoms: 1963
Number of forced atoms: 82
Number of truth assignments: 94124
Size of searchspace (removed): 256 (0)
```

La prima linea di output riporta la versione del risolutore, mentre la linea successiva contiene il primo modello calcolato. Se SMODELS viene invocato con l'opzione '0', vengono stampati tutti i modelli stabili del programma e, immediatamente dopo, la parola 'False' (ad indicare che non vi sono altri modelli oltre a quelli visualizzati); se invece l'utente richiede un numero preciso di modelli, viene stampata la parola 'True' (potrebbero esserci altri modelli). Viene poi indicata la durata del processo di ricerca.

Il numero di *choice points* indica quante volte il sistema ha dovuto "indovinare" il valore di verità per un atomo ground; il numero di *wrong choices* indica invece quante volte tale scelta si è rivelata scorretta, causando il ricorso al *backtracking*[2]. Il numero di choice point può essere utilizzato come misura della complessità di un problema. Le due righe successive danno informazioni sulla dimensione del programma in input (numero di atomi e di regole).

Le restanti righe mostrano quanto le euristiche di SMODELS siano state impiegate nel risolvere il problema. Il numero di *picked atoms* indica quante volte l'euristica *lookahead* è stata in grado di assegnare un valore di verità ad un atomo. Il numero di *forced atoms* rappresenta il numero di atomi aggiunti al modello in quanto la loro negazione avrebbe causato una contraddizione. Il numero di *truth assignments* indica quante volte SMODELS ha assegnato un valore di verità ad un atomo. La dimensione dello spazio di ricerca rappresenta il numero massimo di scelte necessarie per essere certi che un modello esista (oppure non esista).

SMODELS è probabilmente il risolutore ASP attualmente più diffuso e conosciuto. Nuovi sistemi ad esso ispirati stanno nascendo in questi anni, con l'obiettivo di migliorarlo grazie all'applicazione di nuove euristiche. Esempi ne sono CSMODELS (vd. 2.6.3) e CMODELS (vd. 2.6.4).

2.6.2 LPARSE

I sistemi per la programmazione logica tradizionale sono *query-driven*: l'utente pone una domanda e il sistema tenta di trovarvi una risposta. In ogni istante del calcolo, solo quelle variabili in qualche modo coinvolte nella query hanno associato un

[2]Il backtracking consiste nel tornare sui propri passi per modificare quelle scelte che hanno portato ad un fallimento.

valore. Nell'Answer Set Programming, invece, tutte le variabili vengono rimosse dal programma sostituendovi tutti i possibili valori costanti, in tutte le regole. Questo é il compito di LPARSE ([?]).

Esempio 2.6.1. Consideriamo il seguente programma:

$libro(divina_commedia).$

$libro(promessi_sposi).$

$libro(orlando_furioso).$

$studente(elisa).$

$studente(daniele).$

$legge(S, L) :- studente(S), libro(L).$

LPARSE sostituirà all'unica regola non ground del programma 6 regole ground ottenute sostituendo a S ciascuna costante descrivente uno studente e a L ciascuna costante corrispondente ad un libro:

$legge(elisa, divina_commedia).$

$legge(elisa, promessi_sposi).$

$legge(elisa, orlando_furioso).$

$legge(daniele, divina_commedia).$

$legge(daniele, promessi_sposi).$

$legge(daniele, orlando_furioso).$

L'output prodotto da LPARSE codifica, tramite interi, le regole e gli atomi del programma: è su questo formato che il motore di inferenza di SMODELS lavora. Sebbene altri front-end siano disponibili (*parse*, *pparse*, *mcsmodels* [?]), LPARSE è quello più ricco di caratteristiche e più diffuso.

LPARSE viene utilizzato come grounder non solo da SMODELS, ma anche da

Csmodels e Cmodels.

2.6.3 Csmodels

Csmodels ([?], [?]) estende il motore inferenziale di Smodels tramite l'introduzione di una nuova euristica, detta euristica delle *criticalities*.

Uno dei passi chiave per l'individuazione di un answer set è il choice point. Ad ogni choice point, il sistema sceglie un atomo non ancora "coperto" (uncovered), cioè a cui non è stato ancora possibile assegnare valore vero o falso, e vi assegna uno di questi valori. Tale scelta può portare all'individuazione rapida di un answer set, oppure può provocare l'ingresso in un cammino di ricerca errato, con conseguente spreco di tempo e necessità di backtracking. È perciò utile impiegare delle euristiche nella scelta di un atomo scoperto, in modo da guidare tale scelta e migliorare le prestazioni del sistema.

Csmodels utilizza un'euristica basata sulle criticalities, usate in precedenza nell'ambito della pianificazione gerarchica. La pianificazione gerarchica viene impiegata per gestire la complessità dello spazio di ricerca nei problemi di pianificazione; si basa su una gerarchia di astrazione, che ha il compito di suddividere il problema in diversi livelli di dettaglio. Una buona gerarchia di astrazione pone ai livelli più alti (quelli più astratti) le parti più complesse del problema, in modo che il pianificatore tenti di risolverle per prime, limitando così il ricorso al backtracking tra i livelli. Il concetto di criticality (valore numerico assegnato ad un letterale, il cui scopo consiste nel misurare la difficoltà di trovare un piano che lo soddisfi) è stato introdotto proprio per generare automaticamente gerachie di astrazione.

In Csmodels, le criticalities dei letterali sono calcolate per indicare la difficoltà di trovare tali letterali in un answer set. Una criticality può essere descritta come

un'approssimazione del costo richiesto per soddisfare un letterale: un valore di criticality basso corrisponde ad un letterale semplice da soddisfare, mentre valori alti sono assegnati a letterali più complessi. Scegliendo per primi quei letterali indicati come "più difficili" ai primi choice point, si limita il backtracking e si aumenta la velocità di individuazione di un answer set. Perciò, le criticalities dei letterali sono utilizzate da CSMODELS come euristiche per guidare la scelta del sistema ai choice point.

L'algoritmo principale di CSMODELS è lo stesso di SMODELS, eccezion fatta per le parti relative ai choice point. Al primo punto di scelta, il sistema calcola i valori di criticality. L'euristica di cui fa uso non è dinamica, in quanto i valori calcolati al primo choice point sono riutilizzati a tutti i choice points successivi. Il numero di choice point può essere utilizzato per valutare quante volte il risolutore ha fatto ricorso alla sua euristica.

2.6.4 CMODELS

I sistemi per l'Answer Set Programming sono in qualche modo simili ai risolutori SAT; tuttavia, il problema di calcolare gli answer set di un programma è computazionalmente più complesso. È sulla base di questa osservazione che si basa CMODELS[3] ([?], [?], [?]).

Questo risolutore ASP calcola gli answer set di un programma utilizzando come motore di ricerca un risolutore SAT. Questa scelta è giustificata dal fatto che, per programmi logici che abbiano la proprietà di *tightness*, la semantica Answer Set equivale alla semantica di *completamento*.

[3]Esistono due versioni di questo risolutore. La versione utilizzata negli esperimenti di questo lavoro è CMODELS-2.

Definizione 2.6.1. Il *grafo delle dipendenze positive* di un programma di base annidato[4] Π è un grafo diretto G tale che

– i vertici di G sono gli atomi che compaiono in Π;

– G ha un arco da A ad A' se Π ha una regola con testa A' e che contiene A nella parte positiva del corpo.

Definizione 2.6.2. Un programma è detto *tight* se il suo grafo delle dipendenze positive non presenta cicli.

I programmi usati nell'Answer Set Programming sono, generalmente, tight; tuttavia, CMODELS-2 è in grado di gestire anche programmi che non godano di questa proprietà tramite il calcolo delle *loop formula* ([?]) e un approccio del tipo *generate and test*[5]. Nel caso tight, CMODELS semplifica il programma in input e ne forma il completamento. Tale completamento viene poi convertito in forma clausale e sottoposto ad un risolutore SAT. I modelli trovati saranno anche gli answer set del programma.

Attualmente, CMODELS può essere invocato con uno dei seguenti risolutori SAT ([?]):

- *mchaff*;
- *zchaff*;

[4]Un *programma di base annidato* è un insieme finito di regole del tipo

$$A_0 \leftarrow A_1, \ldots, A_k, not\, A_{k+1}, \ldots, not\, A_m, not\, not\, A_{m+1}, \ldots, not\, not\, A_n$$

[5]Viene *generato* un modello proposizionale che soddisfi le clausole proposizionali e successivamente si *verifica* se tale modello sia un answer set per il programma.

- *simo*;
- *relsat*.

Negli esperimenti condotti per questo lavoro di tesi, CMODELS è stato utilizzato con ciascuno dei suddetti motori SAT.

2.6.5 DLV

DLV ([?], [?], [?]), a differenza degli altri risolutori visti finora, tratta la classe dei programmi logici disgiuntivi. In altre parole, il linguaggio accettato da DLV arricchisce la sintassi pseudoproposizionale prevedendo l'utilizzo della disgiunzione nella testa delle regole. Un altro tratto distintivo di questo risolutore consiste nel mancato ricorso ad un front-end distinto, come LPARSE, per il processo di grounding. L'istanziazione viene operata da una parte del risolutore stesso.

La funzione principale dell'algoritmo su cui si basa DLV prende il nome *dlv*. Tale funzione prende in ingresso un programma logico disgiuntivo ground Π e una interpretazione I e stampa tutti gli answer set di Π che estendono I. Come accade in SMODELS, anche *dlv* chiama altre tre funzioni:

- $expand_{dlv}(\Pi, I)$: estende iterativamente I, aggiungendo le conseguenze deterministiche di I rispetto a Π, fino a quando viene raggiunto un punto fisso;
- $heuristics_{dlv}(\Pi, I)$: si occupa di selezionare un letterale PT (*possibly-true*) che estenda il più possibile I se vi viene assegnato il valore *true*;
- *isAnswerSet*(Π, S): restituisce *true* se S è un answer set di Π, altrimenti restituisce *false*.

Il sistema DLV è composto da diversi sotto-sistemi. I principali sono:

- *Grounder Intelligente*;
- *Generatore di Modelli*;
- *Controllore di Modelli*.

Il *Processore delle query* controlla l'esecuzione dell'intero sistema ed esegue operazioni di pre-processamento sull'input e post-processamento sui modelli generati; il suo compito principale consiste nel leggere il programma in input e passarlo al *Gestore delle regole e dei grafi*, che lo spezza in sotto-programmi. Questi sono poi inviati al *Modulo per il grounding intelligente*, il quale genera un sottoinsieme del programma ground, dotato degli stessi modelli stabili del programma originale, ma generalmente di dimensioni molto più contenute. Successivamente, il Processore delle query chiama nuovamente il Gestore delle regole e dei grafi, che produce due partizioni del programma ground: esse vengono utilizzate dal *Generatore di modelli* e dal *Controllore di modelli*, rispettivamente. Infine, viene avviato il Generatore di modelli, il quale genera un modello candidato e invoca il Controllore dei modelli per verificare che sia effettivamente un modello stabile; se l'operazione ha successo, il controllo passa nuovamente al Processore delle query per il post-processamento o per l'individuazione di altri modelli.

Attualmente, DLV rappresenta il principale concorrente di SMODELS. La popolarità di questo sistema sta crescendo grazie alle sue caratteristiche peculiari, come, ad esempio, l'efficienza del modulo di grounding e la disponibilità di predicati built-in per il calcolo degli aggregati.

Capitolo 3

Quake 3 Arena

Quake 3 Arena (*Q3A*) è l'*ambiente* sul quale si è costruita l'applicazione oggetto di questa tesi. *Qsmodels* è stato sviluppato inserendosi su *Q3A* rimpiazzando le routine di Intelligenza Artificiale native, e sfruttando tutti gli altri sofisticati meccanismi di base già presenti.

Nelle sezioni seguenti si spiegherà l'architettura di *Q3A* con particolare enfasi alle parti direttamente coinvolte nello sviluppo di *qsmodels*.

3.1 Introduzione

Q3A è un gioco in *prima persona*, uno dei più rappresentativi del genere; il giocatore vede il mondo attraverso gli occhi del proprio *avatar* e naviga all'interno di un mondo tridimensionale in real-time.

Il capostipite di questo genere di giochi fu Wolfenstain 3D, sviluppato da id Software, la stessa di *Q3A*. L'introduzione di un genere di giochi il cui aspetto fondamentale fosse l'impersonificazione del giocatore con il proprio alterego digitale fu una

rivoluzione del mercato del tempo, creando quello che è a tutt'oggi uno dei generi di maggior successo commerciale.

L'obiettivo del gioco è eliminare i propri avversari e sopravvivere usando il mondo virtuale come arena di combattimento; i giocatori partecipanti possono essere altri esseri umani o Intelligenze Arificiali, ciascuno connesso tramite LAN o Internet ad un unico *server* di gioco che governa il combattimento.

Il giocatore affronta i suoi avversari all'interno di vari mondi virtuali chiamati *livelli* o *mappe*; ogni *livello* è costituito da un insieme di stanze connesse da vari corridoi (*locazioni*). Ogni *locazione* della *mappa* può avere uno o più *bonus* che servono al giocatore per aumentare la propria capacità di fuoco o la propria resistenza ai colpi degli avversari. Una volta che un bonus è stato raccolto deve passare del tempo prima che diventi nuovamente disponibile.

Sono a disposizione del giocatore vari tipi di armi che si differenziano in base alla velocità, al danno inflitto dai proiettili ed alla velocità di ricarica; le armi che causano maggiori danni sono quelle con il più lungo tempo di ricarica (*railgun*) o con proiettili lenti (*rocket launcher*). Ogni arma ha un raggio di azione che determina la gittata del proiettile ed una sfera di influenza che indica la zona dove l'esplosione del proiettile causa danni. Se la distanza dal giocatore al punto di impatto è minore del raggio della sfera di influenza, il proiettile causa danno al giocatore stesso.

Il giocatore può saltare e accovacciarsi per schivare i colpi; tuttavia salti da altitudini troppo elevate causano danni all'avatar.

Ogni volta che il giocatore uccide un avversario guadagna un punto (*frag*), ogni volta che uccide se stesso ne perde uno.

Le modalità di gioco in *Q3A* sono fondamentalmente tre:

- *Duel*: un giocare ne affronta solo un altro in *mappe* solitamente piccole
- *Deathmach*: modalità tutti contro tutti o a squadre
- *Capture the flag*: i giocatori sono divisi in due squadre, l'obiettivo è di catturare la bandiera della squadra avversaria e portarla presso la proprio base.

L'applicazione sviluppata prende in esame solo il caso *Duel*, ma l'architettura è giàă predisposta per il controllo di più avversari e di più intelligenze artificiali contemporaneamente.

3.2 Architettura

Q3A è organizzato secondo il modello *client server*; un server regola il comportamento del mondo ed i client gli si collegano per richiedere lo stato del mondo ed aggiornarlo sulle mosse effettuate dai giocatori. Il client svolge un'azione di mera visualizzazione e raccolta input; cosà che si evita la possibilità che client contraffatti possano effettuare operazioni non consentite. Se il client controllasse parti del mondo fisico un giocatore male intenzionato potrebbe crearsi una versione modificata *ad-hoc* per concedere al proprio avatar maggiori capacità di danno, piuttosto che maggiore resistenza ai proiettili.

La comunicazione tra client e server avviene tramite `socket` in modo da poter operare via internet; per risparmiare banda il client riceve eslcusivamente le informazioni su ciò che è attualmente *visibile* dal giocatore. Il server è l'unica entità ad avere conoscenza completa del mondo.

Qsmodels è stato sviluppato sopra il server per poter facilmente intervenire su ogni componente del mondo[1].

Q3A è suddiviso in quattro macro entità

- *engine*: cuore di tutto il "sistema quake"
- *game*: logica del server
- *c_game*: logica del client
- *ui*: menu, informazioni a schermo

Ciascuna entità utilizza le funzionalità dell'*engine* attraverso una serie di funzioni chiamate `trap`. L'engine è l'unica parte che è rimasta *closed-source* e si occupa di: visualizzazione del mondo, *path-finding*, collision-detection, simulazione fisica.

Qsmodels, sviluppato come modifica di *game* opera come tutti gli altri moduli effettuando `trap` al sottosistema *engine*.

Il server (*game*) si occupa di gestire gli incontri mantenendo: le sessioni di gioco, i punteggi, le regole di gioco, nonchà l'intelligenza artificiale dei *BOT* che costituisce la parte propenderante dell'unità *server*. Esistono vari tipi di *BOT* inclusi in *Q3A*: tutti seguono uno stesso schema di ragionamento, i fattori di cambiamento sono dati dalla velocità di reazione, aggressività, precisione della mira e vari altri parametri.

[1] E' possibile tuttavia sviluppare anche *BOT* che funzionino sul lato client, e anzi potrebbe essere oggetto di ricerche future coordinare le intelligenze di più *BOT* senza avere un'entità con conoscenza completa.

3.3 Collision Detection

Un componente fondamentale dell'architettura di *Q3A* è l'*Area Awareness System(AAS)* che è alla base dell'Intelligenza Artificiale degli agenti. Una spiegazione più dettagliata dell'*AAS* verrà fornita nella prossima sezione; intanto forniamo una breve introduzione al sistema di collision detection usato in *Q3A*, che ne costituisce parte fondamentale.

Q3A utilizza un sistema molto semplice basato su bounding box per il controllo delle collisioni. L'agente sta dentro un bounding box che rappresenta la sua posizione ed il volume occupato; per semplificare ulteriormente i calcoli il box è allineato con gli assi, quindi non ruota in base al movimento del *BOT*.

Il mondo è costituito da *brush* che sono blocchi convessi. Un brush è un volume chiuso delimitato da un certo numero di piani. Le collisioni non vengono calcolate tra brush e bounding box ma tra brush "espansi" ed il centro del bounding box descrivente l'agente. L'espansione che avviene lungo il vettore normale a ciascuna delle facce delimitanti il brush è determinata in base alla dimensione del bounding box; a questo punto è molto semplice determinare se l'agente è all'esterno (*non collidende*) o all'interno (*collidente*) del *brush*.

La creazione dei *brush*, che ricordiamo devono essere convessi, avviene tramite suddivisione del mondo usando i *Binary Space Partitioning Trees* (BSP).

3.3.1 Binary Space Partitioning

Il Binary Space Partitioning è una tecnica di preprocesso dei dati per scene poligonali introdotta da Fuchs, Keden e Naylor [?]. Dal punto di vista geometrico la creazione

di un albero BSP richiede la suddivisione di uno spazio *n*-dimensionale tramite un iperpiano di dimensione *n-1*; il processo è eseguito ricorsivamente fino alla creazione di un numero potenzialmente infinito di celle.

La generazione di un albero BSP avviene selezionando un poligono tra la lista di quelli disponibili e suddividendoli in base alla loro posizione *front* o *back* rispetto al poligono scelto. Nel caso in cui un poligono sia sia *front* che *back* facing, il poligono viene diviso in due parti ciascuna inserita nel ramo corretto dell'albero. La selezione del poligono è una scelta fondamentale per evitare di dover dividere troppi poligoni e per non incorrere nella creazione di un albero troppo sbilanciato.

3.4 I BOT

> *(From "robot") Any type of autonomous software that operates as an agent for a user or a program or simulates a human activity.*

I *BOT* di *Q3A* servono per rimpiazzare la necessità di altri giocatori umani. Una partita può cominciare con un numero variabile di *BOT* e più giocatori umani possono giocare contro più giocatori artificiali. Per dare maggiore versatilità al gioco ogni *BOT* ha le proprie caratteristiche peculiari determinate da una serie di parametri di configurazione. Ogni *BOT* adatta il proprio comportamento in base al tipo di gioco che sta affrontando: dal più furtivo *duel* al più chiassoso *deathmatch*. Ogni *BOT* deve conoscere oltre alle regole del gioco le azioni basilari quali il sapersi muovere nel mondo, raccogliere oggetti, sparare. In questa sezione spiegheremo alcune parti fondamentali delle caratteristiche e del funzionamento dei *BOT* di *Q3A* da cui ci siamo ispirati per la realizzazione di *Qsmodels*.

3.4.1 Modello cognitivo

Come ogni essere vivente anche i *BOT* hanno bisogno di rappresentare e ricordare l'ambiente circostante. Il modello cognitivo riveste un ruolo fondamentale nell'economia dell'intelligenza sia in termini di potenza espressiva che di costo computazionale; in ogni caso, ogni agente ignora completamente qualunque aspetto del mondo che non sia stato descritto all'interno del modello.

Il modello utilizzato da *Q3A* e da tutti gli agenti nei giochi è una rappresentazione molto semplificata del modello reale; nella fattispecie è una semplificazione di quello utilizzato per descrivere graficamente e fisicamente il mondo. La percezione del mondo circostante è descritta da un insieme di celle convesse adiacenti di dimensioni variabili; ognuna di queste celle può o meno contenere oggetti utili alla propria sopravvivenza.

La conoscenza di se stesso si riduce a poche variabili che descrivono la propria posizione, la cella di appartenenza, lo stato delle armi, la salute.

L'acquisizione di informazioni sulle componenti mutevoli del mondo avviene tramite artifici che limitano la conoscenza affidata al singolo bot a quegli elementi presenti nella propria visuale. Questo serve per rendere il comportamento di gioco più realistico e possibilmente divertente: un agente non dovrebbe sapere sempre esattamente qual è la posizione dei propri nemici.

Abbiamo fin qui parlato delle informazioni "generiche" disponibili al *BOT*; esistono inoltre delle informazioni specifiche riguardanti il singolo livello che aiutano l'agente a navigare il mondo in modo più intelligente e a intrapprendere percorsi ottimali per sorprendere gli avversari. Queste informazioni sono statiche e impostate da chi crea i livelli.

3.4.2 Architettura a layer

Il *BOT* di *Q3A* è costruito su vari layer, le azioni più complesse sono eseguite dai layer più alti; le azioni dei livelli superiori sono compiuti passando istruzioni ai livelli inferiori.

<table>
<tr><td>4°</td><td colspan="6">Team Leader</td></tr>
<tr><td>3°</td><td colspan="2">Misc AI</td><td colspan="2">AI Network</td><td colspan="2">Commands</td></tr>
<tr><td>2°</td><td>Fuzzy</td><td>Character</td><td>Goals</td><td colspan="2">Navigation</td><td>Chats</td></tr>
<tr><td>1°</td><td colspan="3">Area awareness system</td><td colspan="3">Basic Actions</td></tr>
</table>

Layer 1: fornisce le azioni primitive di *input-output* per l'interazione con il mondo. L'*Area awareness system* (*AAS*), fornisce il supporto per la navigazione e tutte le informazioni concernenti il mondo; ogni fase del sensing avviene attraverso l'*AAS*. Le basic actions sono l'output del gioco: spara, salta, vai avanti, ecc. ecc.

Layer 2: fornisce quelle funzionalità che per un giocatore umano sono "inconsce": scelta *fuzzy* e raggiungimento dei goal, estrazione delle caratteristiche dei bot, movimento all'interno della mappa.

Layer 3: è un insieme di IF-THEN-ELSE e di una macchina a stati finiti dove ciascuno nodo è specializzato in base alla situazione del mondo e dell'agente stesso. Tutto il ragionamento ad alto livello avviene a questo livello; la complessità della FSM utilizzata è notevole e rappresenta un limite concettuale all'espandibilità degli agenti. In questo layer è presente anche un sistema di comandi per rispondere agli ordini provenienti da altri giocatori.

Layer 4: si occupa del coordinamento delle squadre: un agente viene nominato capitano della squadra ed invia comandi agli altri *BOT* per il raggiungimento degli obiettivi. Questo layer non è preso in considerazione nello sviluppo del progetto.

Qsmodels utilizza il primo layer e la navigazione del secondo per governare i propri agenti; gli altri vengono completamente ignorati.

3.4.3 Area awareness system

L'AAS è la parte centrale di tutta l'archidettura dei *BOT* di *Q3A*, fornisce agli agenti le informazioni per la navigazione del mondo nonchà dati sulla posizione di tutte le entità di *Q3A*.

Tutte le informazioni contenute nell'AAS sono frutto di *preprocessing* sui dati 3d del mondo per rendere semplice e veloce l'accesso alle informazioni di routing.

La maggioranza dei giochi in commercio usa un sistema basato su waypoint per la navigazione che, in breve, è una collezione di nodi collegati tra di loro da link con speciali proprietà. La caratteristica fondamentale dei link è che, se i due nodi sono collegati, deve essere possibile spostarsi da un nodo all'altro seguendo un percorso semplice; a questo punto navigare da un nodo all'altro della mappa diventa un'operazione banale usando algoritmi di navigazione dei grafi. La parte che rimane computazionalmente complessa è determinare a che nodo appartenga un punto nello spazio.

Q3A risolve questo problema specificando che ogni waypoint è un *convex hull* all'interno del quale è sempre possibile spostarsi in linea retta senza incontrare ostacoli. Se una faccia del *hull* è adiacente ad una faccia di un altro *hull* significa che i due *hull* - *waypoint* sono collegati ed è quindi possibile muoversi da uno all'altro. Eccezione

della regola si presenta quando non tutta l'area di una faccia di un *hull* è collegata alla faccia di un altro *hull*: in questo caso verifiche aggiuntive vanno effettuate per determinare se l'agente può realmente passare da un nodo all'altro.

Abbiamo specificato che ogni nodo deve essere convesso; ci sono delle eccezioni: un nodo potrebbe contenere un buco all'interno del pavimento che rende impossibile la navigazione su linea retta, fondamentale per ogni *waypoint*; questo tipo di caso viene risolto suddividendo ulteriormente il nodo in altri subnodi convessi.

Le aree dell'AAS vengono create utilizzando un BSP calcolato usando i brush espansi adoperati per il *collision detection*. Come abbiamo visto un set di poligoni può avere diversi possibili BSP associati: il più indicato è quello che genera il minor numero di nodi.

Ogni entià registra nell'AAS la propria posizione, dimensione, tipologia; cosicchà i *BOT* possano adattarsi alla situazione corrente; alcune entità in base alle loro dimensioni possono essere contenute in più nodi.

3.4.4 Routing

Abbiamo visto come l'AAS presenti informazione sulla navigabilità di un livello; vediamo ora come viene effettuato il calcolo della traiettoria.

I livelli di *Q3A* sono piuttosto statici, i cambiamenti possibili sulla topologia dell'AAS sono determinati solo da aperture e chiusure di porte o passaggi; quindi sarebbe possibile creare un sistema di precalcolo del routing per poter effettuare solo un semplice lookup in una tabella. I nodi di un livello sono facilmente più di cinquemila, risulta pertanto molto dispendioso in termini di memoria memorizzare tabelle di

lookup; allo stesso modo l'utilizzo dei tradizionali algoritmi di visita dei grafi quali Dijkstra o A* risulta impedito perchà troppo lenti per l'utilizzo in real-time.

Q3A supera queste limitazioni utilizzando alcuni artifici quali il routing gerarchico e il caching dei percorsi calcolati. Gruppi di nodi vengono uniti in cluster collegati tra loro da quelli che vengono chiamati *cluster portal*. Il routing viene quindi diviso in due livelli: uno macro, riguardante i cluster ed uno più preciso all'interno dei singoli cluster. Il passaggio tra un cluster e l'altro è reso possibile solo attraverso quei nodi a cui è stata assegnata la funzione di *cluster portal*.

Il caching del routing è organizzato seguendo lo schema a due livelli: si memorizzano le informazioni necessarie a livello di area e di cluster; per andare da un punto all'altro della mappa si individua il cluster di appartenenza, poi si interroga la cache sull'elenco dei cluster da percorrere (routing a livello cluster), infine dentro ogni singolo cluster si interroga la cache sulle aree da attraversare per muoversi tra i *cluster portal*.

Nella maggior parte dei casi l'area di destinazione non cambia, visto che le entità da raggiungere sono spesso statiche (porte, bonus, armi); quindi per maggiore efficienza la cache è organizzata per area di destinazione.

3.4.5 Obiettivi

L'obiettivo principale è ovviamente la vittoria del gioco che si ottiene capitalizzando il maggior numero di punti; abbiamo visto che il punteggio viene incrementato di uno per ogni nemico ucciso e decrementato ogni volta che si muore per danni subiti dalle proprie armi.

Il vero obiettivo è eliminare l'avversario e per riuscirci si raggiungono di volta in

volta vari sotto sotto-obiettivi che possono essere divisi in obiettivi a lungo termine (LTG) e obiettivi a breve termine (STG). Il raggiungimento di un LTG può richiedere il superamento di vari STG: un LTG potrebbe essere *armarsi* che si scompone in vari STG: raccogliere arma 1, raccogliere munizioni arma 1, raccogliere arma 2 ecc.

I STG sono le azioni basilari quali raccogliere un oggetto, raggiungere un punto ed in generale operazioni che non comportano grandi spostamenti dal percorso che l'agente sta attualmente seguendo. I LTG sono invece più complessi: i più comuni sono gli obiettivi in cui il BOT decide di volere delle determinate armi e bonus che non sono facilmente raggiungibili; in questi casi l'agente sceglie con un approccio fuzzy qual è l'obiettivo più urgente e cerca di perseguirlo.

Se il BOT si trovi in una situazione di combattimento può ricorrere ad una certa varietà di LTG: se si trova abbastanza "in forma" può decidere di ingaggiare lo scontro a fuoco, altrimenti può cercare di ripararsi in zone non visibili al nemico per prepararsi meglio all'attacco. La preparazione all'attacco prevede altri tipi di LTG: rifugiarsi in un punto critico della mappa dove è più semplice attaccare senza essere visti (questa strategia è detta *camping* da "accamparsi") oppure aspettare il riapparire di un determinato bonus.

3.4.6 Combattimento

La fase di combattimento vera e propria non prevede alcuna forma di ragionamento; tutto il codice descrivente questa parte dell'AI del *BOT* è un tipico esempio di comportamento *reattivo*. L'inizio di un combattimento avviene quando un agente percepisce la presenza di un nemico. Nel caso in cui l'agente venga colpito viene effettuato un controllo a 360°per capire dove si trovi l'avversario ed appena individuato

inizia lo scontro a fuoco.

L'agente *vede* i propri nemici con una visuale di 90°per rendere più credibile l'esperienza di gioco, pertanto è facile che un nemico riesca a cogliere di sorpresa il *BOT*; la visuale è limitata altresà dalla presenza della nebbia e dalla distanza, un nemico troppo distante o immerso nella nebbia non sarà notato dal *BOT*.

Se l'agente si trova faccia a faccia con il nemico deve decidere se è il caso di ingaggiare il combattimento o meno: se non è abbastanza armato o se è troppo ferito può essere una scelta intelligente quella di ritirarsi. Quando il *BOT* decide di iniziare un combattimento ci sono varie componenti da prendere in considerazione: l'arma da usare, dove mirare e come evitare i proiettili avversari.

Ci sono varie tipi di armi che si differenziano in base alla velocità dei proiettili alla frequenza di fuoco e ai danni inflitti dai proiettili. In genere il *BOT* sceglierà l'arma più potente a disposizione; nella scelta concorrono in parte anche altri fattori quali i gusti personali del *BOT* stesso, nonchà la posizione del nemico: se il nemico è molto distante è meglio utilizzare armi dai proiettili veloci.

Una volta scelta l'arma l'agente deve iniziare a mirare; ciascun *BOT* ha delle caratteristiche che specificano la precisione della mira e l'intelligenza nel prevedere la posizione del nemico. In base all'arma usata si aggiusterà il target di sparo: se il proiettile è di tipo istantaneo, come il *railgun*, il target sarà semplicemente la posizione del nemico, altrimenti verrà aggiustato in base al movimento dell'avversario. Gli agenti più sofisticati tengono anche conto della geometria del mondo circostante l'avversario per prevedere i suoi spostamenti; la previsione viene fatta ipotizzando una destinazione e chiedendo alla cache di routing il percorso migliore per raggiungerla.

Durante i combattimenti il *BOT* tenta di evitare i proiettili avversari usando varie

tecniche in base al proprio livello di bravura. Gli agenti più semplici non si muovono per niente durante il combattimento mentre quelli più evoluti provano a muoversi attorno all'avversario per rendere più difficile inquadrarlo; i *BOT* più sofisticati adottano anche altre manovre evasive quali saltare o rannicchiarsi o cambiare spesso la direzione del moto.

La strategia adottata durante il combattimento cambia in base all'arma utilizzata, lo shotgun è letale a distanza ravvicinata ma di scarso impatto a lungo raggio; quindi il *BOT* cercherà di stare vicino all'avversario. Contrariamente un'arma come il rocket launcher ha un raggio d'azione molto più elevato ma ha il problema che l'esplosione del proiettile provoca danni in una sfera attorno al punto d'impatto, il che rende pericoloso sparare da distanza ravvicinata.

In conclusione l'obiettivo delle routine di combattimento è quello di rendere lo scontro il più credibile possibile e ciò avviene implementando difficoltà più che reali routine di intelligenza visto che la vera difficoltà sta semplicemente nella mira accurata. Anche per questo motivo in *Qsmodels* il combattimento è stato affidato alle routine di basso livello.

3.4.7 AI Network

Il *cervello* di tutta l'architettura dei *BOT* è una macchina a stati finiti chiamata AI network. Ogni nodo di questa FSM rappresenta uno stato possibile dell'agente che può stare sempre solo in una delle posizioni. Ad ogni iterazione viene determinato quale sia il miglior nodo descrivente la situazione corrente.

Ciascun nodo contiene al suo interno una serie di regole `if-then-else` che determinano l'effettivo comportamento del bot in quello stato ed il cambio verso nodi

adiacenti in modo che quello corrente sia sempre il più appropriato.

Figura 3.1: AI network

Nello schema di figura 3.4.7 (L'immagine è stata presa da...) si vede la FSM utilizzata dal *BOT*. I quadrati rappresentano i nodi, le frecce direzionali i passaggi da uno stato all'altro. Si può notare come alcuni nodi siano collegati da link bidirezionali ad indicare che quegli stati possono indicare sia sorgente che destinazione. Sono presenti anche altri due nodi non raffigurati che sono il nodo `ovserver` ed il nodo `intermission`; in questi stati il *BOT* non sta giocando, e si può sempre confluire in questi due nodi speciali.

I nodi possibili sono:

- Respawn
- Stand
- Seek Long Term Goal
- Seek Short Term Goal
- Seek Activate Entity
- Battle Fight
- Battle Chase
- Battle Retreat
- Intermission

- Observer

L'agente può essere ucciso in ogni nodo ad eccezione di `observer`, `respawn` ed `intermission`. Se l'agente viene ucciso ritorna sempre allo stato `respawn`. Una volta apparso il *BOT* passa sempre allo stato `seek long term goal`. Se il gioco finisce si passa al nodo `intermission`; allo stesso modo se il *BOT* entra in modalità osservatore utilizza il nodo `observer`. All'inizio di ogni partita il primo nodo attivato è `stand`.

Il nodo `stand` è utilizzato per la chat; in questa situazione il *BOT* non può muoversi ne sparare; in compenso se vede un nemico può decidere di iniziare il combattimento passando al nodo `battle fight`. Finita la sessione di chat il *BOT* passa allo stato `seek long term goal`.

Lo stato `seek long term goal` è utilizzato per perseguire il LTG attualmente selezionato; nel mentre l'agente può decidere di raccogliere oggetti presenti nel suo cammino senza stravolgere il percorso selezionato per il LTG; in questi casi si passa allo stato `seek short term goal`. Raggiunto lo STG viene nuovamente attivato il nodo `seek long term goal`.

Il nodo `seek activate entity` viene attivato se durante il compimento di un LTG o STG il *BOT* si trova a dover superare degli ostacoli.

Se il nodo corrente è uno qualunque dei `seek` l'agente può trovarsi in condizione di dover cominciare un combattimento; se si sente pronto entra nel nodo `battle fight` ed inizia lo scontro a fuoco secondo le modalità indicate in 3.4.6. L'entrata nel nodo `battle chase` avviene se il nemico non è più visibile; quindi l'agente si dirige verso l'ultimo punto dove l'ha visto e aspetta che il nemico riappaia. Nel momento in cui l'avversaio compare si rientra nel nodo `battle fight`, se questo non accade entro un certo periodo, si torno al nodo `seek long term goal`.

Il nodo `battle retreat` è attivato se durante un combattimento il *BOT* non si sente pronto perchà non armato adeguatamente o perchà troppo ferito. In questo caso si passa allo stato `seek long term goal` sapendo che molto probabilmente il goal più urgente da soddisfare sarà il rimettersi in forma.

Quando l'agente sta utilizzado un nodo di tipo combattimento ha la possibilitàădi raccogliere bonus presenti nelle vicinanze utilizzando il nodo `battle short term goal`; una volta raccolto torna allo stato precedente.

Come si vede la FSM non è particolarmente complicata visto che i nodi sono piuttosto "generici"; tuttavia l'insieme delle regole di produzione inserite all'interno dei singoli nodi rendono difficile capire nel dettaglio il comportamento dell'agente.

Capitolo 4

Architettura di QSMODELS

L'applicazione oggetto di questa tesi è stata realizzata partendo dai sorgenti pubblici di *Q3A* ed eliminando gran parte delle routine di intelligenza artificiale presenti. Una volta isolato il *main loop* del'AI dei *BOT* siamo intervenuti inserendo la nostra applicazione. Il comportamento dell'agente è determinato da un pianificatore realizzato in Answer Set Programming. L'interprete `smodels` riceve un programma ASP descrivente lo stato del gioco e le azzioni possibili; il goal da raggiungere è posto come vincolo. Gli Answer Set trovati contengono la descrizione del piano da passare in esecuzione al *BOT*. L'esecuzione della serie di azioni costituenti il piano è il nuovo *main loop* dell'AI di *Q3A*.

In questo capitolo presentiamo l'architettura dell'applicazione con lo schema di funzionamento di *Qsmodels*, il cuore del nostro progetto. Verrà anche spiegato nel dettaglio il funzionamento di *QsmodelsServer*.

4.1 Introduzione

Il compito della nostra architettura è riuscire a rendere *visibile* l'esecuzione di un piano smodels utilizzandolo in un contesto di gioco quale *Q3A*. Per riuscirci bisogna trasformare delle informazioni disponibili all'interno del mondo di *Q3A* in informazioni utilizzabili dal planner e viceversa.

La realizzazione di *Qsmodels* ha previsto lo sviluppo di tre componenti fondamentali:

- il *planner* realizzato in Answer Set Programming
- *Qsmodels Server* che si occupa di calcolare il piano con l'ausilio di `smodels`
- *Qsmodels* che come modifica di Quake 3 Arena provvede alle osservazioni (*sensing*) ed all'esecuzione del piano.

L'applicazione consta fondamentalmente di due operazioni: la prima consiste nella discretizzazione e conversione in fluenti delle variabili che descrivono lo stato di *Q3A*, la seconda consiste nell'esecuzione del piano estratto "comandando" gli agenti tramite azioni primitive di *Q3A*.

L'elemento planner che verrà descritto nel prossimo capitolo riceve dal componente *Qsmodels* i *fluenti*, ciascuno dei quali specifica il valore di uno stato in un certo istante. La complessità computazionale del calcolo di un piano è fortemente legata al numero di regole; e quindi di fluenti, risulta pertanto fondamentale contenerne il numero. *Q3A* utilizza variabili di tipo *float* o *long* per gran parte dei propri stati, ma come abbiamo visto non si può associare ad un fluente una granularità pari a quella di un numero reale (pena l'esplosione dell'istanziazione equindi dei tempi di calcolo),

pertanto nella trasformazione *variabili* a *fluenti* si effettua una discretizzazione che trasforma ogni valore di variabili in un predicato definito su di un range di valori piuttosto limitato.

Esempio 4.1.1. Variabile *health*: valori compresi tra 0 (morto) e 100. La variabile può crescere fino a 200 ma sopra il 100 viene decrementato automaticamente di un punto al secondo.

Q3A Var.	→	fluent	meaning
[0]	→	0	dead
[1 – 30]	→	1	low
[31 – 60]	→	2	medium
[61 – 90]	→	3	high
[91 – 200]	→	4	excellent

La ricerca delle variabili da utilizzare per gli scopi del planner è stato uno dei punti critici nello sviluppo dell'applicazione; il codice sorgente di *Q3A*, pur ben scritto e leggibile, non è minimamente documentato; quindi spesso si è dovuto ricorrere a tentativi e *reverse-engeneering* per capire dove risiedessero le informazioni ricercate. Lo stesso si può dire per la parte complementare del progetto: far eseguire le azioni; nel capitolo 6 sarà spiegato in dettaglio ogni utilizzo del codice di *Q3A*.

4.2 L'architettura

Abbiamo visto come l'applicazione consista di tre componenti: *Qsmodels*, il *Planner* e *QsmodelsServer* oltre ovviamente a *Q3A*.

Lo schema generale di esecuzione prevede che *Qsmodels*, realizzato come modifica di *Q3A*, abbia il duplice scopo di informare il *planner* sullo stato del mondo ed eseguire

le azioni calcolate dal pianificatore. Si può quindi pensare di dividere *Qsmodels* in due parti MOD-1: *q3a* → *planner* e MOD-2: *planner* → *q3a*. Il termine MOD è derivato dal mondo dei videogiochi dove viene utilizzato per una qualunque modifica operata da amatori su giochi commerciali.

MOD-1 passa le informazioni a *QsmodelsServer* che, applicate le dovute modifiche al planner, provvede prima al *grounding* del piano tramite l'utilizzo di `lparse` ed in seguito al calcolo del piano tramite `smodels`. Il piano ottenuto viene analizzato e passato a MOD-2 che provvede ad eseguirlo. Lo schema proposto è visibile in figura 4.1.

4.3 Ciclo di esecuzione

Un ulteriore obiettivo di questo progetto è il dare un'implementazione completa all'architettura di agente proposta da Baral, Gelfond, Provetti [?]. L'applicazione segue quindi il ciclo seguente.

1. *sensing*
2. *scelta del goal più urgente*
3. *pianificazione*
4. *esecuzione di una porizione del piano*
5. *ritorno a 1*

Questo schema è molto astratto ed è stato modificato secondo lo schema di figura 4.2 per adattarsi alle esigenze dell'interattività. Al ciclo principale di alto livello viene

affiancato un ciclo di basso livello responsabile delle emergenze e delle situazioni in cui il piano non è disponibile. I due cicli sono mutuamente esclusivi, l'AI di alto livello decide se lasciare il controllo al basso livello quando lo ritiene più opportuno.

Nell figura 4.2 i rettangoli rappresentano un possibile stato del programma, i passaggi di stato possibili sono indicati dalle frecce direzionali.

All'inizio di ogni ciclo avviene il *sensing*: si estraggono le informazioni sullo stato dell'agente e del mondo da *Q3A*. Se sono presenti situazioni di emergenza si controllano le regole estratte da smodels per scegliere il comportamento più adatto: la prosecuzione del piano corrente oppure la ricerca di una "reazione" attraverso l'IA di basso livello; se non c'è emergenza si controlla la presenza di un piano: in caso affermativo si controlla se è ancora valido, altrimenti si passa all'intelligenza di basso livello. Se esiste un piano ancora valido si continuuerà con l'esecuzione, altrimenti si passerà al basso livello.

4.3.1 Sensing

Nella fase di sensing vengono estratte le informazioni riguardanti il planner. Per prima cosa si controlla lo stato di salute del *BOT* il suo livello di armi e la posizione all'interno del mondo corrispondenti alle variabili globali associate alla struttura `bot_state_s`.

L'agente memorizza quali sono gli oggeti / bonus presenti nella stanza dove si trova. Il *BOT* conosce in partenza la posizione di ciascun bonus in quanto fa parte della descrizione della mappa stessa; quindi l'operazione consiste nel verificare che il bonus risulti disponibile.

In seconda fase si controllano eventuali emergenze.

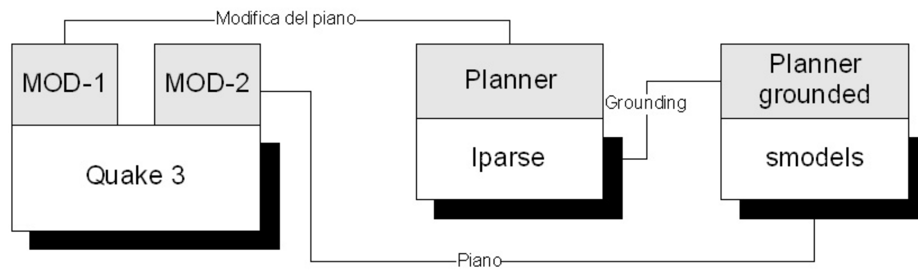

Figura 4.1: Schema Architettura Proposta

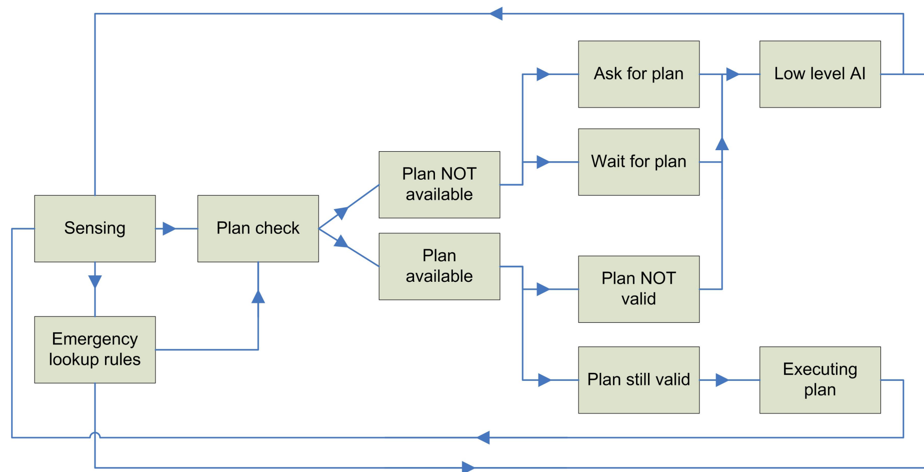

Figura 4.2: Ciclo di Esecuzione

1. L'agente è sotto attacco
2. L'agente è faccia a faccia con il nemico
3. L'agente vede il nemico ma non può essere visto

Se nessuna di queste emergenze è in atto si procede normalmente altrimenti si passa alla gestione delle emergenze.

Seguendo lo schema logico di *Q3A*, per dare maggior realismo al gioco non si è voluto dare al *BOT* maggiori informazioni di quante ne potrebbe avere un giocatore umano; il BOT vede solo ciò che è dentro il suo campo visivo, e scopre se un oggetto è in una stanza o meno solo se lo ha effettivamente raggiunto.

4.3.2 Emergenze

In letteratura è stato più volte osservato che in casi di emergenza è necessario attivare un comportamento reattivo; è meglio ottenere una decisione subottimale ma immediata che chiamare il pianificatore ed aspettare i risultati. La soluzione adottata prevede l'utilizzo di un comportamento reattivo ad *alto livello*: al momento della generazione del piano vengono estratte una serie di regole aggiuntive che descrivono i comportamenti d'emergenza.

Esempio 4.3.1. Supponiamo di avere una regola che descriva il comportamento da seguire quando il BOT si trova faccia a faccia con il nemico

```
% face_to_face_attack(Location,Time).
```

La regola è stata creata supponendo che sia meglio affrontare il nemico in spazi aperti; a meno che il *BOT* non sia in ottima salute e molto ben equipaggiato, nei corridoi è meglio scappare.

```
face_to_face_attack(3,T) :-
                holds(health_level(H),T),
                holds(ammo_level(A),T),
                ammo(A), health(H),
                A >= 4, H>=4.
```

Una volta calcolato il piano otterremo tra le informazioni aggiuntive una serie di *regole* del tipo

```
face_to_face_attack(2,4)
```

Questa regola dice che se il BOT si trova in posizione 2 al tempo 1 allora può attaccare.

Una volta individuata una situazione d'emergenza verrà ricercata e scelta la regola applicabile alla situazione corrente. Rifacendosi all'esempio 4.3.1, si controlla se esiste una regola che all'istante e nella posizione in cui si trova il BOT suggerisca di attaccare.

Si noti come questo particolare dell'architettura costituisca un discostamento dal modello originale proposto da Baral et al [?]. La sua introfuzione concilia l'esigenza di reazioni *real-time* con quella di una semantica descrittiva: le *regole aggiuntive* proposte descrivono il comportamento da attuare in situazioni d'emergenza.

Le regole restituiscono sempre delle semplici azioni da eseguire quali *attacca* o *scappa* che vengono eseguite dall'intelligenza di basso livello. Utilizzare una regola d'emergenza invalida il piano corrente. Nel caso in cui non esistano regole per gestire la situazione d'emergenza si procede con la normale esecuzione del piano.

4.3.3 Richiesta del piano

Passata la fase di *sensing* (senza *emergenze*) si controlla che sia presente un piano da eseguire, se manca si fa un ulteriore controllo per vedere se è già stata inviata una richiesta, nel qual caso si passa il controllo all'intelligenza di basso livello.

La richiesta di un nuovo piano prevede due passaggi: il primo ottiene i fluenti da passare al pianificatore, il secondo prevede l'invio a *QsmodelsServer* di tutte le informazioni necessarie.

I fluenti che vengono analizzati sono:

1. Posizione del BOT
2. Stato di salute
3. Stato delle armi
4. Ultima posizione in cui è stato visto il nemico
5. Posizione degli oggetti attivi

Oltre ai fluenti si ricava lo storico delle posizioni: durante la fase di sensing che avviene ad ogni ciclo di esecuzione si controlla se il nemico è visibile, se scompare per più di un certo numero di secondi viene memorizzata l'ultima posizione in cui è stato visto; appena riappare si salva la posizione in cui è comparso. Questo meccanismo serve per cercare di prevedere i movimenti del nemico, in modo da poter studiare un attacco più intelligente. Si noti che la memorizzazione viene attivata solo se il nemico non è visibile per un certo lasso di tempo (attualmente dieci secondi) per evitare che si memorizzino informazioni anche quando l'avversario per esempio passa dietro ad una colonna.

Tutte le informazioni raccolte vengono inviate al server che provvederà a calcolare il piano; intanto che il piano viene elaborato *qsmodels* continuerà ad essere governato dall'intelligenza artificiale di basso livello.

4.3.4 Validità del piano

La presenza di un piano non garantisce la sua validità nel gioco, ma solo rispetto allo *snapshot* statico del dominio al tempo d'inizio. Ci sono numerose situazioni che possono portare all'invalidazione di un piano. Nel paragrafo precedente abbiamo visto come lo stato del mondo venga osservato nell'istante in cui si effettua la richiesta al pianificatore; dal momento in cui si mandano i dati a *QsmodelsServer* a quando il piano è effettivamente pronto possono passare alcuni secondi. Un piano prevede l'esecuzione di un certo numero di azioni; ogni azione richiede un certo lasso di tempo per essere eseguita (ad esempio muoversi), cosicché da quando il piano viene ricevuto a quando viene eseguita la n-esima azione passano secondi. Durante questo tempo è possibile che la situazione del mondo vari in modo tale da rendere il piano non più corretto per impossibilità di esecuzione o per situazioni di pericolo; in questi casi il piano viene dichiarato non valido e nel prossimo ciclo di esecuzione se ne richiederà uno nuovo.

Esempio 4.3.2. Un tipico caso di invalidamento del piano si ha quando il piano richiede di raccogliere un bonus che sia già stato prelevato dall'avversario; in questi casi in realtà il BOT aspetta per qualche secondo che il bonus riappaia prima di dichiarare il piano non più valido.

Un altro caso di invalidazione del piano si ha quando si passa alla fase di attacco che conclude ogni piano, se il nemico non è presente nella locazione prevista il piano

non è ovviamente più valido. Il *BOT* si sposta in una posizione casuale e poi passa alla richiesta di un nuovo piano; se il piano prevede di rimanere fermi ad aspettare il nemico ed il nemico non si presenta è meglio muoversi in direzione casuale per uscire dall'empasse e confidare che in breve tempo lo si incontrerà.

4.3.5 Esecuzione del piano

Un piano è costituito da una serie di azioni primitive. Ad ogni azione è associato una funzione di *Qsmodels* come modifica di *Q3A* che provvede ad eseguirla. Le azioni primitive sono

1. `move_towards`
2. `pick_health`
3. `pick_ammo`
4. `attack`
5. `elude`

Il significato di ogni azione è evidente dal nome. Tutte le azioni eccetto attack sono in realtà delle varianti di `move_towards`: le azioni di raccolta ricavano dove si trova l'oggetto richiesto e si recano verso il punto indicato, l'azione `elude` trova qual è il punto più adatto per scappare e vi si reca.

Le azioni di movimento sono realizzate utilizzando il primo layer dell'intelligenza artificiale di *Q3A* (cfr. 3.4.2), si imposta come goal il raggiungimento di una certa *area* (cfr. 3.4.3) e gli si dice di raggiungerla; una volta arrivato a destinazione si usa

una delle proprietà dell'AAS (cfr. 3.4.3): si procede in linea retta verso il punto esatto da raggiungere.

L'azione `attack` comporta invece la "sospensione" dell'intelligenza artificiale di alto livello: il controllo passa a *Q3A* che decide che armi utilizzare e come attaccare. *Qsmodels* mantiene la supervisione decidendo come e quando riprendere il controllo del *BOT*. Il passaggio da un livello di intelligenza all'altro è governato dal valore di ritorno della funzione che gestisce l'intelligenza di alto livello, se la funzione ha successo significa che è riuscita a svolgere tutti i compiti preposti, se invece fallisce il basso livello deve sopperire.

4.3.6 Intelligenza di basso livello

Abbiamo visto come l'intelligenza di basso livello (*LLAI*) è un componente molto importante di tutta l'architettura; sovrintende tutte le funzioni del *BOT* in caso di emergenza, in caso di mancanza di piano da eseguire e in caso di attacco.

Ogni azione ordinata dal piano viene in qualche modo eseguita dalla *LLAI* con diversi gradi di libertà: quando deve eseguire un azione di movimento ci si riduce a chiamare delle funzioni di *path-finding* e nelle fasi di emergenza viene utilizzata per eseguire un "mini-piano" che si riduce sempre alle azioni `attack` o `elude`. Diversa è invece la situazione in cui la *LLAI* deve sovraintendere ad uno scontro a fuoco; in questi casi il livello superiore funge solo da sovraintendente, decidendo quando interrompere l'azione, mentre tutte le decisioni operative (muoversi, spararare, cambiare arma) sono delegata alla *LLAI*. La scelta è motivata dal fatto che durante uno scontro armato non c'è tempo di effettuare un ragionamento ma si adotta un semplice processo reattivo che può essere ben svolto dalle regole di produzione del nodo *Battle*

Fight dell'*AI Network* di *Q3A* (cfr. 3.4.7).

Nel caso di mancanza di piano da eseguire la *LLAI* assume il controllo totale e provvede a mantenere il *BOT* attivo fin quando il nuovo piano non sarà disponibile.

Stato	*Livello*	*Utilizzo*
Esecuzione piano	basso	*path-finding*
Emergenze	basso	esecuzione azioni `elude` o `attack`
Combattimento	medio	controllo operativo
Mancanza piano	alto	controllo totale

Tabella 4.1: Schema di utilizzo della Low Level AI

4.4 Stati del programma

Abbiamo visto nella sezione precedente come il ciclo di esecuzione adottato sia descrivibile da una FSM dipendente dallo stato del piano. Esistono due variabili di stato che descrivono esattamente cosa stia facendo *Qsmodels* in quel ciclo di esecuzione (`m_eState` e `m_ePlanState`).

Gli stati sono associati a ciascun *BOT* in quanto l'architettura è predisposta per lavorare con agenti e piani diversi. Le variabili definiscono una lo stato di esecuzione, l'altra lo stato di validità del piano.

Lo stato di esecuzione può valere:

1. `EXECUTING_ACTION`

2. `IDLE`

3. `THINKING`

Il primo valore indica che si sta eseguendo un'azione del piano, finita l'esecuzione, se il piano è finito si passa allo stato IDLE altrimenti si rimane nello stesso stato.

Lo stato IDLE indica che il piano è stato correttamente portato a termine, quindi procede alla raccolta dei dati per la richiesta di un nuovo piano. Lo stato successivo è THINKING.

Lo stato THINKING è quello di attesa di un nuovo piano: la mancanza di un piano da eseguire comporta il passaggio del controllo all'intelligenza di basso livello. Il controllo torna all'alto livello una volta che il nuovo piano è stato ricevuto da *QsmodelsServer* e lo stato diventa EXECUTING_ACTION.

Lo stato di validità del piano può assumere i seguenti valori:

1. PLAN_VALID
2. PLAN_NOT_AVAILABLE
3. PLAN_NOT_VALID_ELUDE
4. PLAN_NOT_VALID_ATTACK

Il piano è valido se ci troviamo nello stato EXECUTING_ACTION altrimenti è PLAN_NOT_AVAILABLE. Ogni qualvolta ci si trovi in uno stato d'emergenza (cfr. 4.3.2) si interrogano le regole aggiuntive per determinare il micro piano da eseguire. Se si passa in uno stato di piano non valido, lo stato di esecuzione rimane EXECUTING_ACTION .

Quando si passa allo stato PLAN_NOT_VALID_ATTACK si manda in esecuzione un azione di attacco, mentre con lo stato PLAN_NOT_VALID_ELUDE vengono interrogate ulteriormente le regole aggiuntive per capire quale sia la posizione da raggiungere.

Una volta conclusa l'azione di attacco o di fuga si passa allo stato PLAN_NOT_AVAILABLE mentre a livello di stato di esecuzione si passa a IDLE.

4.5 QsmodelsServer

Il componente server dell'architettura ha il compito di calcolare piani di esecuzione in base allo stato del mondo fornitogli da *QSmodels*. E' un processo che può essere eseguito su una macchina distinta da quella di gioco.

Il programma è diviso in cinque parti:

1. Comunicazione in ingresso
2. Modifica Planner
3. Calcolo piano
4. Estrazione piano
5. Comunicazione in uscita

Il ciclo di esecuzione è sequenziale nei punti indicati.

QsmodelServer è il componente più semplice di tutta l'architettura, è realizzata in *C++* e consta semplicemente di due classi, una principale che si occupa della gestione della comunicazione ed è invocata direttamente da `main` e l'altra `CPlanExtractor` che si occupa di tutto il resto.

4.6 Comunicazione

QsmodelsServer comunica con *Qsmodels* tramite socket TCP/IP. La comunicazione è gestita da un semplicissimo protocollo binario; per prima cosa il server attende un comando che può essere `SEND_FLUENTS` `GET_PLAN`.

4.7 Conclusioni

In questo capitolo abbiamo descritto il lato concettuale dell'applicazione oggetto della nostra tesi, i dettagli implementativi verranno analizzati nel capiptolo 6.

Si noti l'importanza che le cosiddette *regole aggiuntive* ricoprono nel conferire espressività all'intera architettura. La definizione di un sistema ad alto livello per la trattazione del comportamento reattivo in un contesto di ragionamento automatico ha notevolmente migliorato l'efficacia del nostro agente.

Capitolo 5

Il Pianificatore automatico

Nella nostra architettura il carattere e – in un certo senso – l'intelligenza del *BOT* sono dati principalmente in termini della specifica di un'istanza di pianificazione, che è oggetti di questo capitolo.

In questo capitolo viene descritto il pianificatore automatico (*Planner*) corrispondente ad un Answer Set Program. Come spiegato, da ogni Answer Set di tale programma si può estrarre la lista di azioni da eseguire.

E' altresì presente una specifica del comportamento reattivo, il cosiddetto basso livello dato tramite la definizione di *regole aggiuntive*. Le regole aggiuntive non sono presenti nell'architettura originariamente proposta in [?] e sono state introdotte per rendere più realistico ed efficace il comportamento del nostro *BOT*. Questa innovazione verrà descritta approfonditamente in seguito.

Uno degli obiettivi dell'applicazione è quello di realizzare un laboratorio virtuale per l'Intelligenza Artificiale; il *planner* proposto va inteso come esempio per la realizzazione di altri esperimenti. Un eventuale planner sostitutivo che preveda l'utilizzo di nuove azioni o un modello cognitivo diverso da quello proposto richiedrà modifiche al nucleo di *Qsmodels*, ma nel prossimo capitolo vedremo come ciò sia facilmente

realizzabile.

5.1 Modello Cognitivo

Il nostro agente richiede, come abbiamo visto per *Q3A* (cfr. 3.4.1), la specifica di un modello cognitivo che descriva la percezione che il *BOT* ha di se stesso e del mondo circostante, nel quale però i nemici agiscono come entità separate e modellate a parte.

5.1.1 Il mondo

La descrizione del mondo è costruita sopra l'Area Awareness System (AAS) (si veda 3.4.3) di *Q3A*. Il mondo - il piano di gioco - è diviso in un numero limitato di macro aree (7-10) che rappresentano i punti strategici di ogni mappa; per esempio, si può associare ciascuna di queste macro aree ad un corridoio o ad una stanza del livello. La caratteristica fondamentale delle aree è l'omogeneità delle caratteristiche di combattimento: se l'agente si trova in un punto qualunque dell'area non deve avere particolari vantaggi o svantaggi rispetto al trovarsi in un altro punto della stessa.

Ricordiamo che la navigazione è affidata all'intelligenza artificiale di basso livello, (si veda 4.3.5); pertanto qui non è necessario porre particolare rilievo alla forma o percorribilità delle zone. Abbiamo definito ciascuna macro area come un insieme di *triangoli* che definiscono un area chiusa; un'area può assumere una forma qualunque e la sua posizione e le sue dimensioni non sono vincolate alla morfologia del livello: una zona può coprire aree esterne al livello o comunque non sovrapponibili ad alcuna parte del livello. È importante notare che una corretta individuazione delle aree è un fattore determinante nel rendere il *BOT* realistico e vincente.

Il mondo è descritto come un grafo completo i cui archi vengono etichettati con la distanza tra le relative aree (tempi di percorrenza).

Non è obbligatorio che le aree siano adiacenti, mentre lo è il fatto che siano disgiunte; in ogni istante l'agente deve trovarsi in una ed una sola area. Se l'agente si trova in un punto non appartenente ad alcuna area, l'ultima area valida verrà considerata quella corrente.

A ciascuna area possono essere associati uno o più bonus: di ogni bonus si specifica il tipo (*health* o *ammo*) ed il valore (nel range [5-100]). Ricordiamo che ogni bonus una volta colto non è più disponibile per un certo ammontare di tempo, quindi la mappa cambierà ad ogni invocazione del pianificatore. Resta da implementare un trattamento di fluenti *ottimi* con cui rendere il "riapparire" di un bonus.

La descrizione del mondo è definita da parti statiche (la topologia del livello) e da parti dinamiche (la presenza di bonus); per questo la topologia è descritta come una serie di predicati atemporali, mentre i bonus sono associati a fluenti.

5.1.2 L'agente

Le informazioni riguardanti l'agente sono il proprio stato di salute / armi e la posizione all'interno del mondo; ognuna di queste è altamente dinamica, quindi associata a fluenti.

Lo stato di salute e la quantità di armi associate al *BOT* viene modificato dalle azioni `pick_health` e `pick_ammo`, mentre `move_towards` cambia la posizione. Prima dell'esecuzione del planner, i valori iniziali dei fluenti vengono estratti da *Q3A* (cfr. 4.1) attraverso la fasi di discretizzazione dei valori ed inseriti nel planner tramite il

predicato `initially/1`. Lo stato di salute è associato al fluente `health_level/1` che può avere cinque valori (vedi Tabella 5.1).

Analogo trattamento è riservato allo stato delle armi che è associato al fluente `ammo_level/1`. Si noti che mentre in *Q3A* si ha un livello di munizioni associato a ciascuna arma, nel planner si utilizza un generico *stato* delle armi perché la scelta dell'arma più adatta è lasciata all'intelligenza di basso livello. Il calcolo del livello di armi è fatto normalizzando la quantità di proiettili associata a ciascun arma in modo da dare maggior peso ai proiettili che causano maggior danno. Valgono i seguenti pesi:

$FluentAmmo = Ammo[MACHINE_GUN]+$

$Ammo[SHOTGUN] * 3+$

$Ammo[ROCKET] * 10+$

$Ammo[PLASMA] * 5;$

Valore fluente	Significato
0	dead / no ammo
1	low
2	medium
3	high
4	excellent

Tabella 5.1: Valori fluenti

La posizione dell'agente è impostata associando al fluente `hero_at/1` una delle macro aree (si veda 5.1.1).

5.1.3 Il nemico

La conoscenza del nemico è basata su informazioni dedotte dalla pratica di gioco. L'agente non conosce l'esatta posizione del nemico, tantomeno il suo stato di salute.

In effetti lo stato di salute del nemico non rientra nelle informazioni utilizzate per decidere gli attacchi; in *Q3A* non si ha un riscontro visivo ai danni causati all'avversario; quindi anche al giocatore umano risulta difficile capirlo.

La posizione del nemico è invece un informazione *supposta*: si parte da una posizione *certa*, chiamata `last_known_enemy_position/1` a cui si associa il predicato `guessed_enemy_pos/1` che determina tramite l'ausilio di `enemy_motion/2` il luogo dove l'avversario dovrebbe recarsi. I predicati `enemy_motion/2` indicano la possibile destinazione del movimento a partire da una posizione nota; queste informazioni vengono create durante una partita osservando gli spostamenti del nemico: se il nemico non è più visibile dal *BOT* per un periodo superiore a dieci secondi, si procede alla creazione di un nuovo predicato `enemy_motion/2` da inserire nel planner. Il predicato prevede come primo parametro l'ultima posizione nota dell'avversario prima dell'uscita dal campo visivo dell'agente e come secondo la posizione assunta dal nemico appena riapparso.

Lo scontro si concluderà con il *BOT* che comincia l'attacco nella posizione che il *planner* ha ritenuto più favorevole, peraltro l'architettura esterna di *Qsmodels* dovrà determinare se il piano attualmente in esecuzione è ancora valido rispetto alle mutate condizioni (si veda 4.3.6).

5.2 Struttura

Il planner nella sua struttura è ispirato alla codifica in ASP mostrata da Chitta Baral in [?], si divide in due sezioni principali: una che regola il calcolo del piano ed una che descrive il mondo. La descrizione del mondo è effettuata tramite un elenco di fluenti, azioni e di coppie precondizioni / conseguenze associate a ciascuna azione. Il calcolo del piano è associato ad una serie di predicati che descrivono come scegliere un'azione.

La divisione netta di queste due sezioni permette al neofita di AI di sviluppare e mantenere un pianificatore, modificando le regole descrittive del mondo, che normalmente sono di semplice comprensione.

La descrizione del mondo è divisa in un preambolo nel quale si descrivono le caratteristiche statiche ed in un corpo che descrive i comportamenti dinamici associati alle azioni e ai fluenti.

5.3 Il preambolo

In questa sezione si trova la descrizione del livello, del *modello cognitivo* e delle *regole aggiuntive*.

Per prima cosa vengono dichiarati gli atomi validi.

```
time(0..length).
way_point(0..no_of_waypoints-1).
ammo(0..no_of_ammo_levels-1).
health(0..no_of_health_levels-1).
delay_type(0..delay_time-1).
```

Il significato di ciascun atomo è evidente dal nome, l'unico sul quale porre l'attenzione è `delay_type/1`: come vedremo in seguito descrive il tempo speso durante gli spostamenti; va sottolineata la differenza con `time/1` che invece indica il tempo dell'azione, ogni azione occupa esattamente uno slot di tempo.

Il secondo gruppo di definizioni riguarda la descrizione della topologia del mondo come grafo completamente connesso.

```
%%% Map Description %%%
% Travel times related to map q3dm1 http://www.pdvluca.net/ai/q3dm1.gif
% time_to_reach ( From, To, Time )
delay_to_reach(0,0,0). delay_to_reach(0,1,1). delay_to_reach(0,2,1).
delay_to_reach(0,3,2). delay_to_reach(0,4,3). delay_to_reach(0,5,3).
delay_to_reach(0,6,4).
```

I link che collegano i nodi del grafo sono monodirezionali in quanto può accadere che il tempo necessario per effettuare il percorso dipenda dalla direzione; il caso tipico è quello del BOT che si trovi su di un ponte: per raggiungere la parte che passa sotto il ponte basta un salto, mentre per risalire è necessario fare un percorso più lungo. (Inserire figura con ponte per spiegare differenze).

Una parte importante del preambolo è la definizione delle *regole aggiuntive*, usate nelle situazioni di emergenza (si veda 4.3.2)

```
%%% Danger Location %%%
% danger(Location,Time)
% Describes if it's dangerous being at Location
% seeing the enemy during a movement from
% location From to location To

danger(4,T) :-
     occurs(move_towards(F,6),T), way_point(F), time(T).
danger(5,T) :-
     occurs(move_towards(F,6),T), way_point(F), time(T).
```

In queste *regole aggiuntive* si specifica che è pericoloso trovarsi in posizione 4 o 5 al momento T se in T il BOT si sta muovendo verso la posizione 6. Questa regola è stata introdotta perché le posizioni 4 e 5 sono degli stretti corridoi e se il nemico si trova in posizione 6 l'agente farà fatica a schivare i proiettili.
Le informazioni fino a qui inserite fanno parte dell'insieme di regole *statiche*, ad ogni invocazione del planner rimangono identiche.

Vediamo ora le informazioni sulla previsione del movimento del nemico, che fanno parte del gruppo di informazioni che cambiano ad ogni invocazione del planner.

```
%%% enemy_motion(OldLoc,NewLoc) %%%
% Facts that describe the opponent's movement habits

enemy_motion(0,3).
enemy_motion(1,2).
```

A queste regole si aggiunge un predicato `guessed_enemy_pos/2` che determina la posizione prevista.

```
guessed_enemy_pos(GuessPos) :-
  enemy_motion(LastPos,GuessPos),
  way_point(LastPos),
  way_point(GuessPos),
  last_known_enemy_location(LastPos).

guessed_enemy_pos(LastPos) :-
  way_point(LastPos),
  way_point(GuessPos),
  last_known_enemy_location(LastPos),
  not enemy_motion(LastPos,GuessPos).
```

Queste due regole descrivono come utilizzare le informazioni ottenute da `enemy_motion/2` se ne esistono di adatte, altrimenti si suppone che l'avversario sia rimasto fermo. Si

noti l'utilizzo di `last_known_enemy_location/1` che è l'ultima informazione inserita nel preambolo.

```
last_known_enemy_location(3).
```

5.4 I Fluenti

I fluenti servono per indicare un aspetto atomico dello stato del mondo in un certo istante: abbiamo visto come vengano utilizzati per descrivere i componenti dinamici del mondo quali la posizione dell'agente o il suo stato di salute. Si noti che la posizione del nemico non è un fluente e viene trattata come costante del piano.

```
fluent(hero_at(Loc)) :- way_point(Loc).
fluent(ammo_loc(Loc,Ammo)) :- way_point(Loc), ammo(Ammo).
fluent(health_loc(Loc,Health)) :- way_point(Loc), ammo(Health).
fluent(ammo_level(Ammo)) :- ammo(Ammo).
fluent(health_level(Health)) :- health(Health).
fluent(time_traveling(Time)) :- travel_time(Time).
fluent(ammo_loc(Pos,Ammo)) :- way_point(Pos), ammo(Ammo).
fluent(health_loc(Pos,Health)) :- way_point(Pos), ammo(Health).
fluent(claims_victory).
```

Questo è il completo elenco dei fluenti utilizzati nel pianificatore; si noti, rifacendosi a quanto visto nel preambolo (cfr. 5.3), come `time_traveling/1` sia un fluente. L'idea è che il tempo speso muovendosi è un fluente che aumenta ad ogni azione di movimento. Associato al particolare fluente `time_traveling/1` introduciamo il predicato `delay/2` che indica il tempo speso per muoversi.

```
%%% delay(Delay,Time)
% The delay represents the time spent during traveling
% Delay is increased if a movement action occurs

delay(0,0).
```

Il `delay` inizialmente è nullo.

```
delay(D,T) :-
  time(T),
  T>0,
  delay(D2,T-1),
  delay_type(D),
  delay_type(D2),
  delay_type(D3),
  way_point(From),
  way_point(To),
  occurs(move_towards(From,To),T-1),
  delay_to_reach(From,To,D3),
  D = D3 + D2.
```

Il `delay` viene incrementato se l'azione corrente è un movimento.

```
delay(D,T) :-
  time(T),
  T>0,
  not happened_motion(T-1),
  delay_type(D),
  delay(D,T-1).
```

Se l'agente al tempo T non si muove, il delay non viene incrementato. Il fluente `time_traveling` è stato introdotto per imporre all'agente un percorso che minimizzi il tempo perso negli spostamenti; vedremo in sezione 5.6 come la generazione di un piano con percorsi non ottimali venga rifiutato.

I fluenti devono avere un valore iniziale che viene indicato tramite il predicato `initially/1`. Allo stesso modo `finally/1` specifica il valore imposto al fluente alla fine del piano.

5.5 Le Azioni

Le azioni sono la parte più importante di tutto il planner, dato che il piano calcolato consta semplicemente in una sequenza di azioni.

La descrizione delle azioni si divide in tre parti: il prototipo dell'azione, la descrizione delle precondizioni e la descrizione delgli effetti.

Precondizioni ed effetti sono dati come ccombinazione (in generale) di condizioni booleane e condizioni sui fluenti.

5.5.1 La Dichiarazione

La dichiarazione serve per fornire una *segnatura* al predicato d'azione, specifica il numero ed il tipo di parametri utilizzati.

```
%%% Actions Declaration

% Non combat actions

action(move_towards(From,To)) :-
    way_point(From),
    way_point(To).
action(pick_ammo(From)) :-
    way_point(From).
action(pick_health(From)) :-
    way_point(From).
action(do_nothing).

% Combat actions

action(skirmish).
action(escape).
```

Le azioni si dividono in azioni di movimento e azioni di combattimento. Ogni azione ha un corrispettivo all'interno di *Qsmodels* che provvede all'esecuzione della stessa tramite la classe `qsAction`.

Il significato di ogni azione è chiaro dal nome, quelle su cui ci vogliamo soffermare sono `do_nothing` e `skirmish`. L'azione `do_nothing` serve come riempitivo nel caso in cui la lunghezza del piano specificata nel *planner* sia inferiore a quella effettivamente richiesta; in questo caso il planner inserisce delle azioni di *padding* per colmare le mancanze. Va notato come il `do_nothing` sia solo un suggerimento per il planner. Una serie di azioni quali:

$$pick_ammo(A); move(A, B); move(B, A)$$

è semanticamente identica alla sequenza:

$$pick_ammo(A); do_nothing; do_nothing$$

I piani da noi generati non sono garantiti minimi. Questa scelta è dovuta alla complessità del problema piano minimo che è maggiore di *NP* e non permette sotto le consuete assunzioni nella gerarchia polinomiale, di dare una codifica concisa per `smodels` e per ASP non-disgiuntivi in generale.

L'azione `skirmish` è invece l'azione che termina il piano, cioè l'inizio dello scontro a fuoco. Un piano per essere valido deve avere questa azione come ultima.

5.5.2 Le Precondizioni

Ogni azione per essere eseguita deve soddisfare delle richieste che vengono speficate tramite il predicato `executable/2`. La condizione di eseguibilità è riferita all'istante T. Rimandiamo all'appendice A per una descrizione esaustiva di tutte le precondizioni, in questa sezione ci limitiamo a descrivere due azioni particolarmente significative, la `move_towards/2` e la `skirmish`.

```
%%% move_towards(From,To)
executable(move_towards(From,To), T)  :-
    time(T),
  T < length,
  way_point(From),
  way_point(To),
  holds(hero_at(From),T).
```

L'agente si può muovere dal punto `From` a `To` all'istante T se in quell'istante si trova in `From` e sia le locazioni che il tempo sono riducibili ad atomi validi.

```
executable(skirmish, T)  :-
    time(T),
    T < length,
    way_point(Loc),
    holds(hero_at(Loc),T),
    guessed_enemy_pos(Loc),
    holds(ammo_level(A),T),
    holds(health_level(H),T),
    ammo(A),
    health(H),
    A >= 3,
    H >= 3.
```

L'azione `skirmish` è eseguibile se il BOT si trova nella locazione in cui si presume si trovi il nemico e se le sue condizioni di salute e di armamento sono sufficientemente elevate.

Il predicato `holds/2` mostrato nel codice è stato introdotto per meta-interpretare la verità di un fluente, qui rappresentati come termini e non come atomi.

5.5.3 Le Conseguenze

Ogni azione comporta delle conseguenze allo stato dei fluenti che vengono specificate dal predicato `causes/3`. Come per le precondizioni ci limitiamo a descrivere due azioni significative rimandando all'Appendice A per un quadro completo. L'azione presa in considerazione è `move_towards`.

```
causes(move_towards(OldPos,NewPos),hero_at(NewPos),T) :-
    time(T),
    way_point(OldPos),
    way_point(NewPos).

causes(move_towards(OldPos,NewPos),neg(hero_at(OldPos)),T) :-
    time(T),
    way_point(OldPos),
    way_point(NewPos).
```

L'azione `move_towards` causa lo spostamento dell'agente da `From` a `To`; quindi viene negato il fluente indicante la vecchia posizione e reso vero quello della nuova.

5.6 La generazione del piano

Il *calcolo* del piano è realizzato da una serie di predicati che costituiscono il nucleo del pianificatore. Ricordiamo che l'Answer Set Programming è un sistema di programmazione dichiarativo, quindi il termine *calcolo* è improprio dato che occorre riferirsi all'interprete: in questa sezione vengono descritte le caratteristiche che un piano deve

soddisfare, la procedura di calcolo vera e propria è demandata al motore inferenziale, nel nostro caso `smodels`.

Il calcolo parte dal vincolo che l'esistenza di un piano è obbligatoria in ogni answer set . Un piano esiste se al tempo T, pari alla lunghezza fissata del piano, si è raggiunto il goal. Il goal è definito come il raggiungimento delle condizioni `finally/1` di tutti i fluenti.

```
fail_goal(T):- time(T),
    literal(X),
    finally(X),
    not holds(X,T).

goal(T) :- time(T), not fail_goal(T).

exists_plan :- goal(length-1).
:- not exists_plan. % La non esistenza di un piano non è ammessa
```

Il calcolo dei fluenti è affidato al predicato `holds/2` che come abbiamo visto in precedenza asserisce che un fluente è valido al tempo T. Un fluente parte come vero se è specificato in uno dei predicati `initially/1`.

```
holds(F, 0) :-
    literal(F),
    initially(F).
```

Un fluente diventa vero al tempo $T+1$ se al tempo T viene eseguita un azione che lo faccia diventare vero dove il predicato `literal` indica un fluente od il suo negato.

```
holds(F, T+1) :-
    literal(F),
    time(T),
    T < length,
    action(A),
```

```
    executable(A,T),
    occurs(A,T),
    causes(A,F,T).
```

Altresì un fluente rimarrà vero al tempo $T + 1$ se al tempo T è vero e se non è accaduta alcuna azione che abbia fatto diventare vero il suo contrario.

```
holds(F, T+1) :-
    literal(F),
    literal(G),
    contrary(F,G),
    time(T),
    T < length,
    holds(F,T),
    not holds(G, T+1).
```

Si notino le ultime due condizioni che rendono la nostra formulazione della legge d'inerzia un *default.*Un azione è eseguibile al tempo T se soddisfa il predicato `executable/2`.

```
possible(A,T) :-
    action(A),
    time(T),
    executable(A,T),
    not goal(T).
```

Il planner prevede l'utilizzo del predicato `happened/2` per indicare un'azione come data, magari come risultato di una pianificazione precedente solo parzialmente eseguita. Il planner attuale non lo utilizza, per i motivi spiegati in **??**.

Un'azione viene effettivamente eseguita, quindi viene selezionata tramite il predicato `occurs/2` se è possibile eseguirla e se nessun'altra azione viene selezionata contemporaneamente.

```
occurs(A,T) :-
    action(A),
    time(T),
    possible(A,T),
    not other_occurs(A,T).

other_occurs(A,T) :-
    action(A),
    action(AA),
    time(T),
    occurs(AA,T),
    neq(A,AA).
```

Capitolo 6

L'implementazione di QSMODELS

Il cuore dell'applicazione oggeto di questa tesi è una libreria realizzata in C++ che viene linkata staticamente al componente server di *Q3A*. Abbiamo già visto nel quarto capitolo lo schema funzionale di questo componente, in questo daremo una spiegazione sull'implementazione.

Lo sviluppo dell'applicazione è stato preceduto da una lunga fase di esperimenti atti ad individuare le funzioni necessarie per l'interazione con *Q3A*. La metodologia di ricerca si è basata principalmente sul *reverse-engineering*; non è garantito che le funzioni utilizzate siano in assoluto le più adatte, così come le informazioni sull'utilizzo delle strutture dati interne di *Q3A* possono contenere delle inesattezze. L'unica prova sull'esattezza delle deduzioni è data dal funzionamento dell'applicazione.

Una volta compresa l'interazione con l'ambiente sottostante si è proceduti allo sviluppo dell'applicazione vera e propria. Facendo riferimento alla figura 4.1 in cui si evidenzia una distinzione tra *MOD-1* (responsabile del sensing) e *MOD-2* (responsabile dell'esecuzione del piano), chiariamo che la distinzione è prettamente concettuale: queste due parti sono confluite in un'unica entità nella fase implementativa. La distinzione è rimasta a livello di ciclo esecutivo, la fase di *sensing* (*MOD-1*) precede

la fase di *esecuzione* (*MOD-2*).

Il capitolo si divide in una prima parte in cui illustreremo l'interazione con *Q3A*, ed una seconda in cui verrà illustrato il progetto nelle sue classi principali. Per la comprensione completa di questo capitolo si assume una discreta conoscienza del C++; per una spiegazione più concettuale si rimanda invece al capitolo 4.

6.1 L'interazione con Quake 3 Arena

L'interazione con *Q3A* si articola in tre componenti: le aree di memoria dove *Q3A* memorizza i dati necessari per il planner e le funzioni di sistema `trap` da chiamare per far eseguire al *BOT* le azioni volute e le funzioni di utilità.

Q3A è stato sviluppato in C e quindi non è fornito di meccanismi di protezione dei dati che aiutino a comprendere quali siano modificabili senza danno. Si consiglia di prestare particolare attenzione a modificare i valori presenti; nella nostra applicazione ci siamo limitati alla sola lettura dei dati, provvedendo a chiamare le funzioni di sistema `trap` quando fosse necessario modificarli.

Abbiamo visto come *Q3A* si diviso in quattro parti separate: il server, il client, il menù e l'engine; ciò comporta che alcune funzioni siano duplicate in più componenti con nomi diversi; si ricorda che è obbligatorio utilizzare quella appartamente allo stesso componente che si sta modificando per evitare problemi di link.

Esempio 6.1.1. La funzione di stampa a schermo è presente in ogni componente: nel server si chiama `G_Printf` nel client `CG_Printf` e nei menù `Com_Printf`

Come regola generale le funzioni appartenenti al server iniziano con il prefisso `G_` mentre quelle appartenenti al client per `CG_`.

6.1.1 Le aree di memoria

Le informazioni utilizzate dal planner sono tutte ricavabili dalla struttura `gentity_s`. Ogni elemento dinamico del mondo di *Q3A* è descritto tramite un entità: i bonus, le armi e i giocatori (sia umani che *BOT*) sono descritti da questa struttura dati. Il livello di gioco è descritto tramite l'Area Awareness System (AAS) che non è stato resto pubblico.

Tutti i calcoli del gioco vengono effettuati dal componente server, i *risultati* vengono trasmessi ai client. Il server mantiene due versioni per i dati, una completa per se stesso ed una ridotta (`playerState_t`) da utilizzare per comunicare con il client.

Le informazioni sui *BOT* sono contenute nella struttura `bot_state_s`. E' da notare che questa struttura viene associata anche a giocatori umani, come discriminante viene utilizzato il campo `inuse`. Questa struttura è quella che assieme a `gentity_s` ha costituito la principale fonte di informazioni per il *planner*.

Molte delle strutture dati descritte in questo paragrafo sono utilizzate sia per le entità dei giocatori "umani" che per quelle dei *BOT*; utilizzeremo il termine *giocatore* per riferirci a questo tipo di entità.

Q3A mantiene una serie di variabili globali che descrivono lo stato corrente del mondo. Quelle che a noi interessano sono `bot_state_t botstates[64]` e `gentity_t g_entities[1024]`. Il gioco può contenere al massimo 64 giocatori e 1024 entità. Ogni *BOT* è un entità: `bot_state_t::entitynum` indica la relazione tra entità e *BOT*.

Esempio 6.1.2. Il *BOT* numero `N` fa riferimento all'entità:

```
gentity_t *pEnt = &g_entities[[botstates[N].entitynum];
```

Informazioni utili riguardanti il *BOT* numero `N`:

- Posizione: `botstates[N].origin`
- Direzione di vista: `botstates[N].viewangles`
- Direzione di moto: `botstates[N].velocity`
- Stato di salute: `botstates[N].lasthealth`
- Stato delle armi: `botstates[N].cur_ps.ammo[ARMA_SELEZIONATA]`

Lo stato delle armi è ricavato dal campo `cur_ps` che è del tipo `playerState_t`, una struttura usata dal server per comunicare con il client. Tutti i dati inviati al client possono essere rintracciati tra quelli disponibili al server, molte informazioni come la posizione e la visuale sono facilmente riconoscibili sia nella struttura `bot_state` che nella playerState_t, altre, come lo stato delle armi, le abbiamo individuate solo nel lato client. La pratica generale seguita è stata il cercare le informazioni utili senza fare distinzioni tra quelle disponibili solo al server e quelle inviate al client; questo approccio è possibile perché, come detto, *Qsmodels* è sviluppato come modifica del server e quindi abbiamo una conoscenza completa del mondo. Tutto questo non sarebbe valido se il *BOT* fosse stato realizzato in modalità client.

Un certo numero di variabili vengono utilizzate per rilevare gli "eventi" destinati ai giocatori. Un contatore indica il numero di volte che l'evento si è verificato.

Esempio 6.1.3. Per controllare se il giocatore ha subito un danno viene utilizzato il contatore

```
if(ps->damageEvent!=ops->damageEvent && ps->damageCount)
  UnderAttack();
```

dove `ps` e `ops` sono puntatori a strutture `playerState_t`. Si noti come *Q3A* mantenga due versioni dello stato una relativa al frame corrente ed una a quello precedente in modo da poter fare i raffronti del caso.

6.1.2 Le funzioni di sistema

Le funzioni di sistema sono tutte le funzionalità esposte dall'engine, sono indentificate dal prefisso `trap_` e sono tutte degli alias all'unica funzione esposta da *engine.dll* chiamata `syscall(int arg,...)`. Una qualunque funzione `trap_*` viene eseguita dalla chiamata `syscall(TIPO_FUNZIONE,PAR_FUNZIONE)`.

Esempio 6.1.4. La funzione `trap_Printf` per stampare stringhe sulla console è realizzata da

```
void trap_Printf( const char *fmt )
{
   syscall( G_PRINT, fmt );
}
```

Le *syscall* realizzano funzionalità quali:

- Navigazione: `trap_BotMoveToGoal`
- Movimento in una direzione: `trap_BotMoveInDirection`
- Check di collisione / visibilità: `trap_Trace`
- Sparo: `trap_EA_Attack`
- Tutte le interrogazioni all'AAS: `trap_AAS_*`

6.1.3 Le funzioni di utilità

Il componente server compone più *syscall* con codice aggiuntivo per realizzare funzioni di utilità. *Qsmodels* fa largo uso di queste funzionalità.

Una funzione che è stata particolarmente utilizzata è la `BotEntityVisible` che determina se un *BOT* ha un entità specifica nel proprio campo visivo. Il prototipo della funzione è:

```
BotEntityVisible(int viewer, vec3_t eye,
                  vec3_t viewangles, float fov,
                  int ent)
```

Significato parametri:

- `viewer`: numero dell'entità *BOT*
- `eye`: posizione del *BOT* nel mondo
- `viewangles`: direzione di vista
- `fov`: ampiezza visuale in gradi
- `ent`: numero dell'entità di cui controllare la visibilità.

Il numero di un entità fa riferimento all'array `g_entities`.

La funzione `BotSetupForMovement` è un'altra funzione di utilità molto importante, serve per inizializzare lo stato del *BOT* prima di un movimento.

```
BotSetupForMovement(bot_state_t *bs)
```

Significato parametri:

- `bs`: stato descrivente il *BOT*

6.2 Organizzazione classi

Il componente *Qsmodels* è sviluppato su due classi chiamate `qsManager` e `qsBot` che gestiscono rispettivamente, le risorse dell'applicazione e i *BOT*. Con risorse dell'applicazione intendiamo tutti i componenti accessori che costituiscono il modello cognitivo del nostro agente: la mappa con relativi bonus le azioni eseguibili e le *regole aggiuntive*.

(inserire schema classi)

Le azioni eseguibili e le *regole aggiuntive* sono implementante come risorse per facilitare lo sviluppo di estensioni al programma. Ad ogni azione eseguibile viene associata un'istanza[1] della classe `qsAction` che mantiene informazioni relative al tipo di azione ed un puntatore alla funzione *C* con il codice per l'esecuzione. La classe `qsBot` si occupa di gestire l'intelligenza del *BOT* in ogni suo aspetto utilizzando le risorse messe a disposizione da `qsManager`.

Sono inoltre presenti alcune classi di utilità

- `qsFluent`: Gestione dei flueni
- `qsBonus`:
- `qsLocation`: Elemento costitutivo della mappa
- `qsMap`: Collection di `qsLocation`
- `qsAction`: Descrizione delle azioni
- `qsRule`: Regole Aggiuntive

[1] Il termine *istanza* viene utilizzato in questo capitolo come traduzione di *instance* nonostante l'evidente inesattezza data la diffusione che ormai si riscontra o il comune significato che ha ormai assunto nella lingua italiana.

Tutte queste classi vengono utilizzate da `qsBot` e da `qsManager`

Il punto di ingresso della nostra applicazione è la funzione globale `::qsDoStuff`

```
int qsDoStuff(gentity_t *_aEntities, bot_state_s **_apStates,
              int (QDECL *syscallptr)(int arg, ...))
```

Significato parametri:

- `_aEntities`: Array di entità
- `_apStates`: Array a puntatori di stati dei *BOT*
- `syscallptr`: Funzione punto di ingresso per le `trap` a *Q3A*

La funzione viene invocata ogni frame dal file `ai_main.c` di *Q3A* e restituisce `true` se l'intelligenza di alto livello è riuscita a mantenere il controllo del *BOT*. Se la funzione restituisce `false` il controllo del *BOT* in quel frame viene passato alla *LLAI* (Low Level AI).

```
int qsDoStuff(gentity_t *_aEntities, bot_state_s **_apStates,
              int (QDECL *syscallptr)(int arg, ...))
{
    if(!g_poManager)
    {
        g_poManager = new qsManager;
        g_poManager->Initialize(_apStates,1,NULL);
        dllEntry(syscallptr);
        return true;
  }
  else
  {
      return g_poManager->DoStuff(_aEntities);
  }
  return false;
}
```

La prima volta che si invoca `::qsDoStuff` viene creato un oggetto globale di tipo `qsManager` e viene inizializzato il sistema delle `trap`.

Nel resto di questo capitolo andremo a descrivere le classi più significative ed alcune funzioni chiave per capire il funzionamento di *qsModels*.

6.3 La classe `qsManager`

La classe `qsManager` è il *gestore* del modulo *Qsmodels*. Il compito principale dell'oggetto è sopravvedere al lavoro dei *BOT*; nel ciclo principale la classe controlla lo stato di esecuzione di ogni *BOT* si comporta di conseguenza.
Le variabili membro della classe sono:

```
qsBot *m_aoBots;
unsigned m_iBots;
qsMap *m_poMap;
qsAction *m_aActions;
unsigned m_iActions;
SOCKET m_Socket;
```

La classe mantiene: un eleco di *BOT* da gestire `m_aoBots` (attualmente l'intelligenza ne gestisce solo uno), un oggetto che descrive la mappa `m_poMap`, un elenco di azioni eseguibili `m_aActions` ed una socket TCP/IP per comunicare con *QsmodelsServer*.

6.3.1 `qsManager::Initialize`

La funzione di inizializzazione viene chiamata all'interno della prima invocazione della funzione globale `::qsDoStuff`.

Lo scopo della funzione è di inizializzare le variabili membro della classe. Il primo compito svolto è la creazione e l'inizializzazione degli oggetti `qsBot` che avviene collegando l'istanza allo stato di quake `bot_state_t`.

```
m_aoBots = new qsBot[_iBots];
m_iBots = _iBots;
qsBot::SetManager(this);
unsigned ii=0,iCnt=0;
while(iCnt<_iBots)
{
    if(_apStates[ii]->inuse)
    {
        m_aoBots[iCnt].Initialize(ii,_apStates[ii]);
        iCnt++;
    }
    ii++;
}
```

La funzione statica `qsBot::SetManager` serve per permettere al *BOT* di chiamare le funzionalità messe a disposizione dal `qsManager`.

La funzione `Initialize` provvede inoltre a: caricare la definizione della mappa con le posizioni delle macro-aree e dei bonus e a registrare le azioni consentite. L'ultimo compito di `Initialize` è stabilire una connessione TCP con *QsmodelsServer*.

6.3.2 `qsManager::DoStuff`

`DoStuff` è il *main* punto ingresso della classe oggetto di questa sezione. La funzione controlla per il *BOT* in esame la stato di esecuzione, provvede a chiedere a *qsModelsServer* un nuovo piano qualora il *BOT* lo richieda. Se il piano è disponibile lo manda in esecuzione.

```
bool qsManager::DoStuff(int _iBot, gentity_t *_aEntities)
```

```
    {
        static unsigned iActions=0;
        static qsAction **paActions=NULL;
        static qsActionParams *asActionParams=NULL;

        s_aEntities = _aEntities;

        m_aoBots[_iBot].RetrieveCurState();
```

Aggiorna il *BOT* sulla situazione corrente

```
        switch(m_aoBots[_iBot].GetState())
        {
        case qsBot::EXECUTING_ACTION:
            m_aoBots[_iBot].ExecutePlan();
            break;
```

Il piano è disponibile e va eseguito. `qsBot::ExecutePlan` provvederà a eseguire l'azione più appropriata e controllare lo stato di validità del piano.

```
        case qsBot::IDLE:
            DbgMessage("Sending data");
            m_aoBots[_iBot].CalcFluents(m_poMap);
            SendData();
            break;
```

Il *BOT* ha finito di eseguire un piano e richiede a *QsmodelsServer* il calcolo di uno nuovo.

```
        case qsBot::THINKING:
            DbgMessage("Waiting for plan");
            m_aoBots[_iBot].LowLevelAI();
            if(NewPlanReady(_iBot,&paActions,&asActionParams,&iActions))
            {
                m_aoBots[_iBot].InitializePlan(paActions,asActionParams,iAction
            }
            break;
        }
```

Il *BOT* sta aspettando il calcolo di un nuovo piano, in questo periodo operera in LLAI. Una volta che il piano diventa disponibile `qsManager` provvede ad inizializzare il *BOT* con un nuovo piano.

```
        if(m_aoBots[_iBot].GoingToLLAI())
        {
            return false;
        }
        return true;
    }
```

Se durante l'esecuzione del piano si è riscontrato un errore si passa alla gestione a basso livello.

6.3.3 `qsManager::NewPlanReady`

La funzione verifica la disponibilità di un nuovo piano controllando lo stato della socket. Se il piano è disponibile provvede alla decodifica del protocollo per memorizzare le azioni da eseguire e le *regole aggiuntive*.

6.4 La classe `qsBot`

La classe `qsBot` gestisce l'esecuzione del piano e la raccolta delle informazioni da *Q3A*. Il suo stato è determinato da due variabili `EState m_eState` e `EPlanState m_ePlanState`.

```
    enum EState
    {
        EXECUTING_ACTION,
        THINKING,
        IDLE
    };
```

L'enum `EState` rispecchia quanto visto in ??.

```
enum EPlanState
{
    PLAN_VALID,
    PLAN_NOT_AVAILABLE,
    PLAN_NOT_VALID_ATTACK,
    PLAN_NOT_VALID_ELUDE
};
```

L'enum `EPlanState` indica lo stato di esecuzione del piano come visto in 4.4.

Le variabili della classe si dividono in tre gruppi:

```
unsigned m_iOldDamageEvent; //Used to check if the BOT is under attack
unsigned m_iClient; //Which BOT is this?
unsigned m_iEntity; //Identify the bot in the quake entity array
bot_state_s *m_psQuakeState;
```

Il primo serve a gestire l'interazione con *Q3A*, mantenendo un link allo stato del *BOT*.

```
EState m_eState; //Is the bot executing actions ?
EPlanState m_ePlanState; //Plan current state
qsAction *m_poCurAction; //Action currently being executed
unsigned m_iCurPlanAction;
qsAction **m_apPlanActions; //Array of plan actions to be executed
unsigned m_iPlanActions; //Number of actions in current plan
qsActionParams *m_apPlanActionsParams;
qsRule *m_aRules; //Array of Additional Rules
unsigned m_iRules;
```

Il secondo gruppo mantiene informazioni sul piano e sul suo stato di esecuzione. Si notino le ultime due variabili che indicano le *regole aggiuntive* da utilizzare per il piano corrente. L'array `m_apPlanActions` va inteso come array di puntatori a funzioni, i cui parametri sono contenuti in `m_apPlanActionsParams`.

```
unsigned m_iMapPos; //Current area the BOT is in
maVector3 m_vPosition;
unsigned m_iHealth;
unsigned m_aiAmmo[MAX_NUM_GUNS];
```

L'ultimo gruppo contiene le informazioni traducibili in fluenti da passare al planner.

6.4.1 `qsBot::RetrieveCurState`

`RetrieveCurState` è la funzione che opera la fase di *sensing* e controlla la validità del piano in esecuzione. L'unione inunica funzione delle due fasi è apparentemente in contraddizione con lo schema previsto da [?]: in realtà in questa si eseguono quelle azioni di *sensing* i cui risultati verranno utilizzati per verificare la validità del piano.

Il primo compito della funzione è aggiornare il *BOT* sul suo stato

```
m_vPosition.Set(m_psQuakeState->origin[0],
                m_psQuakeState->origin[2],
                m_psQuakeState->origin[1]);
if(m_psQuakeState->lasthealth == 0)
    m_iHealth = 100;
else
    m_iHealth = m_psQuakeState->lasthealth;

m_aiAmmo[MACHINE_GUN] = m_psQuakeState->cur_ps.ammo[2];
m_aiAmmo[SHOTGUN] = m_psQuakeState->cur_ps.ammo[3];
m_aiAmmo[ROCKET] = m_psQuakeState->cur_ps.ammo[5];
m_aiAmmo[PLASMA] = m_psQuakeState->cur_ps.ammo[8];
```

Si noti come la nostra applicazione e *Q3A* utilizzino un sistema di riferimento geometrico diverso. (figura assi quake e assi applicazione)

La seconda parte della funzione si occupa di evidenziare eventuali situazioni d'emergenza: il primo controllo riguarda verifica se il nemico sta attaccando il *BOT*.

```
if(m_psQuakeState->cur_ps.damageEvent !=
   m_iOldDamageEvent && m_psQuakeState->cur_ps.damageCount)
{
    G_Printf("I'm under attack\n"); // I'm under attack
    szStrategy = s_poManager->LookupRules(
                 "under_attack",szMapPos,szTime);
    if(!stricmp(szStrategy,"attack"))
        m_ePlanState = PLAN_NOT_VALID_ATTACK;
    else if(!stricmp(szStrategy,"elude"))
        m_ePlanState = PLAN_NOT_VALID_ELUDE;
}
```

Una volta individuato che il BOT è sotto attacco si consultano le *regole aggiuntive* per individuare il comportamento da seguire; se si trova un predicato adatto alla situazione corrente si invalida lo stato del piano e si procede all'esecuzione del minipiano indicato.Un meccanismo analogo viene utilizzato per gestire la situazione in cui il BOT si trova faccia a faccia con il nemico e quella in cui il BOT può vedere il nemico ma non viceversa.

6.4.2 `qsBot::CalcFluents`

La funzione `CalcFluents` discretizza i valori ottenuti nella sensing per essere utilizzati dal planner.

```
  unsigned iAmmo = m_aiAmmo[MACHINE_GUN]
                 + m_aiAmmo[SHOTGUN]*3
                 + m_aiAmmo[ROCKET]*10
                 + m_aiAmmo[PLASMA]*5;
iAmmo /= 100;
if(iAmmo > 4)
    iAmmo = 4;
else if(iAmmo == 0)
  iAmmo = 1;
```

In questo frammento di codice vediamo come avviene la discretizzazione dello stato delle armi. Procedimento analogo avviene per lo stato di salute. La determinazione della posizione all'interno della mappa avviene invece utilizzando la funzione `qsMap::GetMapPos`, la cui implementazione verrà descritta in seguito.

6.4.3 qsBot::ExecutePlan

La funzione `ExecutePlane` utilizza i risultati ottenuti durante la fase di *sensing* per mandare in esecuzione il piano corretto. Il BOT mantiene un'elenco di azioni descriventi il piano corrente; un'azione viene mandata in esecuzione fin tanto che non ha svolto la sua funzione.

Esempio 6.4.1. Supponiamo di voler mandare il BOT in posizione A, ogni frame viene chiamata la funzione `move_towards(A)` fintanto che questa non restituisce il valore `true` che indica il successo dell'azione; in questo caso il raggiungimento della destinazione.

Quando una funzione termina si passa alla successiva nell'elenco del piano, al termine del quale si imposta lo stato del BOT ad `IDLE`.

```
case PLAN_VALID:
    if(m_poCurAction->Execute(this,m_apPlanActionsParams + m_iCurPlanAction))
    {
        m_iCurPlanAction++;
        if(m_iCurPlanAction == m_iPlanActions)
        {
            m_eState = IDLE;
        }
        else
        {
            m_eState = EXECUTING_ACTION;
            m_poCurAction = m_apPlanActions[m_iCurPlanAction];
```

```
        }
    }
```

Vediamo nel fragmento di codice l'esecuzione di un piano valido: se l'azione corrente è terminata si passa a quella successiva.

6.5 La gestione della mappa

La mappa è espressa come collezione di `qsLocation`. La funzione principale della classe è determinare, data la posizione del *BOT*, la macro-area di appartenenza. Ogni macro-area è identificata dalla classe`qsLocation`, collezione di triangoli. Ogni bonus è associato ad un istanza di `qsLocation` in base alla propria posizione.

Per determinare l'area di appartenena del nostro agente si scorre l'elenco di macroaree alla ricerca di una che contenga la posizione cercata. Un punto si trova all'interno di una macro-area se è contenuto in uno dei triangoli che costituiscono l'area.

6.6 La classe `qsAction`

Le azioni vengono eseguite dalla classe `qsAction` che si pone come contenitore il cui elemento principale è un puntatore ad una funzione che è responsabile dell'azione vera e propria.

```
unsigned m_iCode;
char *m_szPrimitiveName;
bool (*m_fnAction)(qsBot *_poBot,
                   qsActionParams *_pParams);
```

I membri `m_iCode` e `m_szPrimitiveName` servono per riconoscere l'azione in fase di *debugging* e di comunicazione con *qsModelsServer*. La classe ha solo due funzioni

mebro, GetActionName ed Execute che al suo interno invoca la funzione nella variabile membro m_fnAction.

6.7 Le azioni

Il punto centrale nell'esecuzione delle azioni si trova nella funzione GoToPosition; le azioni move_towards, pick_health, pick_ammo e indirettente elude, si riducono ad un invocazione della funzione GoToPosition con una diversa verifica del raggiungimento dell'obiettivo.

6.7.1 GoToPosition

La funzione GoToPosition cerca di far raggiungere al *BOT* la posizione ottenuta in input. L'esecuzione della funzione utilizza l'AAS per determinare il tracciato, nonché il sistema dei goal per muovere il *BOT*.

Il primo passo per muovere il *BOT* consiste nel determinare l'area di destinazione.

```
areanum = trap_AAS_PointAreaNum(q_Pos);
```

Se il punto non è in alcun area si cerca l'elenco di aree che vengono attraversate dal segmento che parte dalla destinazione e si dirige verso l'alto per cinquanta unità[2].

```
if(!areanum)
{
    VectorCopy(q_Pos, q_Pos2);
    q_Pos2[2] += 50;
    numareas = trap_AAS_TraceAreas(q_Pos, q_Pos2,
                                   (int *)areas,
                                   NULL, 10);
```

[2] *Q3A* utilizza un sistema di misura basato su unità "generiche" che sono grossomodo paragonabili ai pollici

```
        areanum = areas[0];
}
```

Il numero di aree è attraversabili è limitato a dieci, a noi interessa la prima incontrata.

Trovata l'area di destinazione si controlla con un meccanismo simile l'area di partenza, se le due aree coincidono si sfrutta la proprietà degli AAS che permette la navigazione semplice e si dice al BOT di muoversi in linea retta verso la destinazione. Se le aree di destinazione e di partenza sono diverse si imposta uno Short Term Goal (STG) che ha come obiettivo il raggiungimento dell'area di destinazione.

```
if(bot_areanum != areanum)
{
    trap_BotMovementViewTarget(_poBot->GetQuakeState()->ms,
                               &goal,_poBot->GetQuakeState()->tfl,
                               300,q_Pos);
    trap_BotMoveToGoal(&moveresult,
                       _poBot->GetQuakeState()->ms,&goal,
                       _poBot->GetQuakeState()->tfl);
    if(moveresult.failure)
        g_bError = true;
    else if(moveresult.blocked)
        g_bError = true;
}
else
{
    q_TargetPos[2] = q_Pos[2] = 0;
    VectorSubtract(q_TargetPos,q_Pos,q_TargetDir);
    VectorNormalize(q_TargetDir);
    trap_BotMoveInDirection(_poBot->GetQuakeState()->ms,
                            q_TargetDir,400,MOVE_RUN);
}
```

La variabile membro `tfl` di `bot_state_s` usata nella funzione `trap_BotMoveToGoal` indica il *Travel Flag* che indica qual è lo stato di movimento attuale.

6.7.2 qs_MoveTowards

A titolo di esempio presentiamo `qs_MoveTowards` per capire come funzionano le funzioni di movimento.

```
bool qs_MoveTowards(qsBot *_poBot,
                    qsActionParams *_psParams)
{
    qsLocation *poLoc = (qsLocation *)_psParams->m_aValues[0];
    const maVector3 &vPos = _poBot->GetPosition();
    if(poLoc->Inside(vPos))
        return true;
    GoToPosition(_poBot,poLoc->GetCenter());
    return false;
}
```

La funzione restituisce `true` se ha raggiunto un punto della locazione obiettivo. Tutte le funzioni *azioni* utilizzano lo stesso meccanismo per ricevere i parametri in input: la classe `qsActionParams` contiene un array di dimensione variabile i cui elementi costituiscono l'input della funzione; sta alla funzione sapere quanti di questi valori le servono.

`qs_MoveTowards` estrae dai parametri la locazione destinazione, se la posizione corrente del BOT ne è all'interno allora il compito della funzione è esaurito e quindi restituisce `true` altrimenti invoca la funzione `GoToPosition` passandogli come parametro un punto appartenente alla locazione.

Appendice A

Codice del pianificatore

In questa appendice, viene riportato il codice del pianificatore oggetto di questa tesi. La versione riportata è per LPARSE. *Una versione più aggiornata può essere scaricata, insieme a tutti i file prodotti nell'ambito di questo lavoro, presso il sito* `http://mag.usr.dsi.unimi.it`.

```
% Qplanner v. 1.5
% Jan. 6 2003
% by Luca Padovani luca@l3p.it
% updated version from http://mag.dsi.unimi.it/
% typical call: lparse qplanner.sm | smodels
% However, this command is usually invoked from Qsmodels
% This planner is based on the work of Chitta Baral

%%% Plan Settings %%%

const length=5.
const delay_time=15.

%%% Map Settings %%%

const no_of_waypoints=7.

%%% Game Settings %%%
```

```
const no_of_ammo_levels=5.
const no_of_health_levels=5.

%%% Atoms declaration %%%

time(0..length).
way_point(0..no_of_waypoints-1).
ammo(0..no_of_ammo_levels-1).
health(0..no_of_health_levels-1).
delay_type(0..delay_time-1).

%%% Map Description %%%
% Travel times related to map q3dm1 http://www.pdvluca.net/ai/q3dm1.gif
% time_to_reach ( From, To, Time )

delay_to_reach(0,0,0). delay_to_reach(0,1,1). delay_to_reach(0,2,1).
delay_to_reach(0,3,2). delay_to_reach(0,4,3). delay_to_reach(0,5,3).
delay_to_reach(0,6,4).
delay_to_reach(1,0,1). delay_to_reach(1,1,0). delay_to_reach(1,2,1).
delay_to_reach(1,3,2). delay_to_reach(1,4,3). delay_to_reach(1,5,3).
delay_to_reach(1,6,4).
delay_to_reach(2,0,1). delay_to_reach(2,1,1). delay_to_reach(2,2,0).
delay_to_reach(2,3,1). delay_to_reach(2,4,2). delay_to_reach(2,5,2).
delay_to_reach(2,6,3).
delay_to_reach(3,0,2). delay_to_reach(3,1,2). delay_to_reach(3,2,1).
delay_to_reach(3,3,0). delay_to_reach(3,4,1). delay_to_reach(3,5,1).
delay_to_reach(3,6,2).
delay_to_reach(4,0,3). delay_to_reach(4,1,3). delay_to_reach(4,2,2).
delay_to_reach(4,3,1). delay_to_reach(4,4,0). delay_to_reach(4,5,2).
delay_to_reach(4,6,1).
delay_to_reach(5,0,3). delay_to_reach(5,1,3). delay_to_reach(5,2,2).
delay_to_reach(5,3,1). delay_to_reach(5,4,2). delay_to_reach(5,5,0).
delay_to_reach(5,6,1).
delay_to_reach(6,0,4). delay_to_reach(6,1,4). delay_to_reach(6,2,3).
delay_to_reach(6,3,2). delay_to_reach(6,4,1). delay_to_reach(6,5,1).
delay_to_reach(6,6,0).


%%% Danger Location %%%
% danger(Location,Time)
% Describes if it's dangerous being at Location seeing the enemy during a
% movement from location From to location To
```

```
%danger(4,T) :-
% occurs(move_towards(F,6),T), way_point(F), time(T).
%danger(5,T) :-
% occurs(move_towards(F,6),T), way_point(F), time(T).

%danger(2,T) :-
% occurs(move_towards(F,0),T), way_point(F), time(T).
%danger(2,T) :-
% occurs(move_towards(F,1),T), way_point(F), time(T).

%%% Settings mofified by Qsmodels %%%
% <ADDINFO>

last_known_enemy_location(5).

%%% enemy_motion(OldLoc,NewLoc) %%%
% Facts that describe the opponent's movement habits

enemy_motion(0,3).
enemy_motion(1,2).
enemy_motion(2,3).
enemy_motion(4,4).
enemy_motion(6,4).

%%% Initial fluents of the BOT %%%

initially(health_level(2)).
initially(ammo_level(4)).
initially(hero_at(3)).
initially(time_traveling(0)).

%%% Bonus locations %%%

initially(ammo_loc(1,2)).
initially(ammo_loc(4,3)).
initially(health_loc(2,3)).
initially(health_loc(4,2)).

%%% Bot GOAL %%%
finally(claims_victory).
```

```
%%% End of settings mofified by Qsmodels %%%
% </ADDINFO>

%%% guessed_enemy_pos/1 %%%
% The future enemy position is guessed using the past experience encoded in enemy_motion/2
% If no rules match the current situation, the enemy is supposed not to move

guessed_enemy_pos(GuessPos) :-
enemy_motion(LastPos,GuessPos),
way_point(LastPos),
way_point(GuessPos),
last_known_enemy_location(LastPos).

guessed_enemy_pos(LastPos) :-
way_point(LastPos),
way_point(GuessPos),
last_known_enemy_location(LastPos),
not enemy_motion(LastPos,GuessPos).


%%% Plan costraint
% The total delay time must not be more then twice the delay needed to move from
% the BOTs initial location to the enemy's guessed location
% This is to find a plan which is "not too long"

:-
delay(D,length-1),
initially(hero_at(StartPos)),
    guessed_enemy_pos(EnemyPos),
delay_to_reach(StartPos,EnemyPos,DToReach),
way_point(StartPos),
way_point(EnemyPos),
delay_type(D),
delay_type(DToReach),
D > (DToReach*4).


%%% delay(Delay,Time)
% The delay represents the time spent during traveling
% Delay is increased iff a movement action occurs

delay(0,0).
```

```
delay(D,T) :-
time(T),
T>0,
delay(D2,T-1),
delay_type(D),
delay_type(D2),
delay_type(D3),
way_point(From),
way_point(To),
occurs(move_towards(From,To),T-1),
delay_to_reach(From,To,D3),
D = D3 + D2.

delay(D,T) :-
time(T),
T>0,
not happened_motion(T-1),
delay_type(D),
delay(D,T-1).

happened_motion(T) :-
time(T),
way_point(From),
way_point(To),
occurs(move_towards(From,To),T).

%%% Fluents Declaration

fluent(hero_at(Loc)) :- way_point(Loc).
fluent(ammo_loc(Loc,Ammo)) :- way_point(Loc), ammo(Ammo).
fluent(health_loc(Loc,Health)) :- way_point(Loc), ammo(Health).
fluent(ammo_level(Ammo)) :- ammo(Ammo).
fluent(health_level(Health)) :- health(Health).
fluent(time_traveling(Time)) :- travel_time(Time).
fluent(ammo_loc(Pos,Ammo)) :- way_point(Pos), ammo(Ammo).
fluent(health_loc(Pos,Health)) :- way_point(Pos), ammo(Health).
fluent(claims_victory).

%%% Actions Declaration

% Non combat actions
```

```
action(move_towards(From,To)) :-
      way_point(From),
      way_point(To).
action(pick_ammo(From)) :-
      way_point(From).

action(pick_health(From)) :-
      way_point(From).
action(do_nothing).

% Combat actions
action(skirmish).
action(escape).


%%% executable(Action,Time)
% determines if an action can be executed at time T

%%% move_towards(From,To)
% Can be executed at time T if the hero is at location From
% at time T

executable(move_towards(From,To), T)  :-
    time(T),
T < length,
    way_point(From),
    way_point(To),
    holds(hero_at(From),T).

%%% pick_health(From)
% Can be executed at time T if the Health bonus is present at
% location From at time T

executable(pick_health(From), T)  :-
    time(T),
T < length,
    way_point(From),
    health(Health),
    holds(hero_at(From),T),
    holds(health_loc(From,Health),T).
```

```
%%% pick_ammo(From)
% Can be executed at time T if the Ammo bonus is present at
% location From at time T

executable(pick_ammo(From), T) :-
    time(T),
T < length,
    way_point(From),
    ammo(Ammo),
    holds(hero_at(From),T),
    holds(ammo_loc(From,Ammo),T).

%%% skirmish
% The BOT can attack the enemy if both it and the enemy stand at the
% same location and if the BOT's Health and Ammo levels are high
% enough

executable(skirmish, T)  :-
    time(T),
T < length,
    way_point(Loc),
    holds(hero_at(Loc),T),
    guessed_enemy_pos(Loc),
    holds(ammo_level(A),T),
    holds(health_level(H),T),
    ammo(A),
    health(H),
A >= 3,
H >= 3.

executable(do_nothing,T) :-
time(T),
T < length.


%%% escape
% If the BOT cannot attack the enemy maybe it's better to go away

executable(escape, T)  :-
    time(T),
T < length,
    not executable(skirmish,T).
```

```
%%% causes(Action,Fluent,Time)
% Describes actions consequences on fluents

%%% Moving let the old location fluent become false and the new
% one true

causes(move_towards(OldPos,NewPos),hero_at(NewPos),T) :-
time(T),
way_point(OldPos),
way_point(NewPos).
causes(move_towards(OldPos,NewPos),neg(hero_at(OldPos)),T) :-
time(T),
way_point(OldPos),
way_point(NewPos).

%%% Picking an amno raises the BOTs ammo_level fluent

causes(pick_ammo(From),ammo_level(AmmoNew),T) :-
time(T),
holds(ammo_level(AmmoOld),T),
holds(ammo_loc(From,Ammo),T),
way_point(From), ammo(Ammo), ammo(AmmoOld), ammo(AmmoNew),
AmmoNew = AmmoOld + Ammo.
causes(pick_ammo(From), neg(ammo_level(AmmoOld)),T) :-
time(T),
holds(ammo_level(AmmoOld),T),
holds(ammo_loc(From,Ammo),T),
way_point(From), ammo(AmmoOld), ammo(Ammo), ammo(AmmoNew).
causes(pick_ammo(From),neg(ammo_loc(From,Ammo)),T) :-
time(T),
holds(ammo_loc(From,Ammo),T),
way_point(From), ammo(Ammo).
causes(pick_ammo(From), ammo_loc(From,0),T) :-
time(T),
holds(ammo_loc(From,Ammo),T),
way_point(From), ammo(Ammo).

%%% Picking an Health bonus raises the BOTs ammo_level fluent

causes(pick_health(From),health_level(HealthNew),T) :-
time(T),
```

```
holds(health_level(HealthOld),T),
holds(health_loc(From,Health),T),
way_point(From), health(Health), health(HealthOld), health(HealthNew),
HealthNew = HealthOld + Health.
causes(pick_health(From), neg(health_level(HealthOld)),T) :-
time(T),
holds(health_level(HealthOld),T),
holds(health_loc(From,Health),T),
way_point(From), health(HealthOld), health(Health), health(HealthNew).
causes(pick_health(From),neg(health_loc(From,Health)),T) :-
time(T),
holds(health_loc(From,Health),T),
way_point(From), health(Health).
causes(pick_health(From), health_loc(From,0),T) :-
time(T),
holds(health_loc(From,Health),T),
way_point(From), health(Health).


causes(skirmish,claims_victory,T) :- time(T).

causes(escape,neg(claims_victory),T) :- time(T).


%%% A plan exists if the goal is reached in "length" time
% and if the not-existence of a plan is false

fail_goal(T):- time(T),
    literal(X),
    finally(X),
    not holds(X,T).

goal(T) :- time(T), not fail_goal(T).

exists_plan :- goal(length-1).
:- not exists_plan.

%%% A fluent and it's negationa are literals

literal(G)  :-
    fluent(G).
literal(neg(G)) :-
```

```
    fluent(G).

%%% contrary rule declaration

contrary(F, neg(F)) :-
    fluent(F).

contrary(neg(F), F) :-
    fluent(F).

%%% holds(Fluent,Time) %%%
% Describes wich fluents are true at a specified time
% A fluent is true at time 0 if it has been set by the
% initially/1 statement.
% At time t+1 a fluent it's true if an action happens which
% let it become true

holds(F, 0) :-
    literal(F),
    initially(F).

holds(F, T+1) :-
    literal(F),
    time(T),
    T < length,
    action(A),
    executable(A,T),
    occurs(A,T),
    causes(A,F,T).

holds(F, T+1) :-
literal(F),
    literal(G),
    contrary(F,G),
time(T),
T < length,
    holds(F,T),
    not holds(G, T+1).

%%% possible(Action,Time) %%%
% Actions are executable only if Time is less than plan length
```

```
possible(A,T) :-
    action(A),
    time(T),
    executable(A,T),
    not goal(T).

%%% occurs(Action,Time) %%%
% An action occurs if it can be executed and if it's the only
% one being executed

occurs(A,T) :-
    action(A),
    time(T),
    happened(A,T).

occurs(A,T) :-
    action(A),
    time(T),
    possible(A,T),
    not other_occurs(A,T).

other_occurs(A,T) :-
    action(A),
    action(AA),
    time(T),
    occurs(AA,T),
    neq(A,AA).

%%% Predicates to be hidden %%%

hide time(T).
hide action(A).
hide initially(F).
hide contrary(F,G).
hide fluent(F).
hide literal(L).
hide fail_goal(T).
hide executable(A,T).
hide holds(F,T).
hide not_occurs(A,T).
hide possible(A,T).
hide happened(A,T).
```

```
%hide possible1(A,T).
%hide possible2(A,T).
hide exists_plan.
hide finally(X).
hide goal(T).
hide not_goal(T).
hide way_point(T).
hide direction(X).
hide causes(A,F,T).
hide ammo(X).
hide ammo_loc(A,Q).
hide health(X).
hide health_loc(H,Q).
hide enemy_motion(A,B).
hide delay_to_reach(A,B,T).
hide last_known_enemy_location(P).
hide delay_type(T).
hide other_occurs(A,T).
hide guessed_enemy_pos(A).
hide happened_motion(A).
hide delay(T1,T2).
hide total_delay(A).
hide start_point(P).

%%% End of Planner %%%
```

Glossario

In considerazione del fatto che durante la trattazione sia necessario, per brevità, l'utilizzo di acronimi, segue una lista di quelli più importanti, con i relativi significati.

Definizioni

AAS	Area Awareness System
AI	Artificial Intelligence
ASP	Answer Set Programming
BOT	Robot
BSP	Binary Space Partitioning
FSM	Finite State Machine
LLAI	Low Level Artificial Intelligence
LTG	Long Term Goal
MOD	Modifica (ad un gioco)
NP	Non Polynomial
Q3A	Quake 3 Arena
STG	Short Term Goal

Bibliografia

[1] T. Eiter, G. Gottlob, and H. Mannila, *Disjunctive datalog*, ACM Transactions on Databases, 22(3) (1997), 364–417.

[2] M. Gelfond and V. Lifschitz, *The stable model semantics for logic programming*, Proc. of the 8th International Workshop on Non-Monotonic Reasoning, 1988.

[3] ———, *Classical negation in logic programs and disjunctive databases*, New Generation Computing (1991), pp. 365–387.

[4] W. Marek and M. Truszczyński, *Stable models an alternative logic programming paradigm*, The Journal of Logic Programming, 1999.

[5] I. Niemelä and P. Simons, *Logic programs with stable model semantics as a constraint programming paradigm*, Proc. of NM'98 workshop Extended version submitted for publication, 1998.

[6] I. Niemelä, P. Simons, and T. Syrjanen, *Smodels: a system for answer set programming*, Proc. of 5th ILPS Conference (2000), 1070–1080.

[7] G. Pfeifer and W. Faber, *A disjunctive datalog system (and more)*, http://www.dbai.tuwien.ac.at/proj/dlv/, 2002.