



UNIVERSITÀ DEGLI STUDI “Aldo Moro”

DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA IN INFORMATICA MAGISTRALE

**Progetto di Sistema esperto
Intelligenza Artificiale**

*Raggiungimento target
in presenza di ostacoli mobili:
ambiente grigliato – A**

Professore:

N. Di Mauro

F. Esposito

Studente:

Alessandro Balestrucci

617937

ANNO ACCADEMICO 2014-15

Capitolo 1

Introduzione.....	5
1.1 Introduzione.....	6
1.2 Finalità del Sistema	6
1.3 Hardware e Software usati	7
1.3.1 Hardware & Sistema Operativo	7
1.3.2 Software.....	7
1.3.3 Modalità di sviluppo.....	8
1.4 Lancio del sistema e visualizzazione dei risultati	8
1.5 Considerazioni sull'interprete prolog.....	9

Capitolo 2.

Prolog e programmazione logica.....	10
2.1 Programmazione logica.....	11
2.2 Sintassi prolog	12
2.3 Interprete logico SWI-PROLOG.....	13

Capitolo 3

Progettazione.....	17
3.1 Il problema e l'analisi.....	18
3.1.1 L'ambiente.....	18
3.1.2 Ostacoli Fissi	19
3.1.3 Ostacoli MOBILI (agenti).....	20
3.1.4 Agente (NEO).....	21
3.1.5 Target.....	22
3.2 Strategia risolutiva e algoritmi	23
3.2.1 Approccio iniziale-risoluzione dei triangoli.....	23
3.2.2 Breadth First Search – ricerca in ampiezza	24
3.2.3 A* - ricerca euristica	26
3.3 Interazione con il Sistema Esperto	29
3.3.1 Immissione della Base di Conoscenza a Run-Time	29
3.3.2 Inserimento delle coordinate AGENTE – TARGET	30
3.3.3 Inserimento di un Ostacolo Fisso a Run-Time.....	30

Capitolo 4

Esecuzione del sistema.....	32
4.1 Panoramica	33
4.2 Strutturazione moduli	33
4.3 Computazione.....	36
4.3.1 Esaurimento dei passi percorribili in una direzione	40
4.3.2 Collisione NON FRONTALE OSTACOLI MOBILI (agenti)	40
4.3.3 Collisione FRONTALE tra 2 OSTACOLI MOBILI (agenti)	41
4.3.4 Collisine tra agente e OSTACOLO FISSO	42
4.4 Output Files	47
4.4.1 file.txt	47
4.4.2 decisionDir.txt	47
4.4.3 moveDetails.txt.....	48
4.5 Esempio computazione su visualizzatore.....	49

Capitolo 5

Sviluppi Futuri.....	51
5.1 Finalità alternative del Progetto	52
5.2 Work in progress	52

Bibliografia.....	513
--------------------------	------------

Premessa

Questo elaborato rappresenta la documentazione di un sistema esperto da me realizzato come applicazione pratica dei concetti teorici appresi a lezione e delle esercitazioni svolte in laboratorio.

Questo progetto è stato realizzato da una sola persona (me) facendo riferimento al Prof.Di Mauro, per chiarimenti e aiuto nelle strategie risolutive.

La novità di questo paradigma di programmazione e il nuovo modo di approcciarsi ai problemi si sono rivelati un ostacolo molto impegnativo, che inizialmente si credeva insormontabile.

Capitolo 1.

Introduzione

1.1 Introduzione

In questo capitolo saranno espone in primis gli obbiettivi del sistema, e a seguire verranno descritti gli strumenti Hardware e software impiegati nella sua realizzazione, nonché le modalità di sviluppo; per poi descrivere la modalità utilizzata per l'esecuzione e la visualizzazione dei risultati derivanti dalla computazione del sistema esperto.

1.2 Finalità del Sistema

Finalità iniziale:

- Provvedere affinché un agente, immerso in un ambiente grigliato (una matrice quadrata o non) con presenza di ostacoli MOBILI, possa avanzare nell'ambiente evitando questi ostacoli mobili.

Come si può ben capire la formulazione iniziale dava bene a intendere una vasta gamma di applicazioni specifiche, soprattutto nell'ambito dei giochi. Si pensi ad esempio a tutti i game a scorrimento (Kong, supermario, ecc.) dove l'agente doveva compiere uno specifico task senza scontrarsi con gli ostacoli mobili (si pensi ai funghi del supermario)



Finalità definitiva:

- Un agente immerso in un ambiente grigliato (matrice) e con la presenza di ostacoli FISSI e MOBILI, deve poter giungere da una posizione iniziale a una posizione finale senza scontrarsi con questi ostacoli e scegliendo un buon percorso.

Paragonando il tutto alla trilogia di Matrix: “**Neo deve arrivare al Telefono (per uscire da Matrix) evitando gli agenti Smith**”

Adesso l’obiettivo del sistema è più delineato e preciso e fan ben intendere cosa effettivamente il sistema esperto deve fare e anche in che modo; lasciando tuttavia libertà per quanto concerne la fase di concettualizzazione e quella rappresentazione, rappresentanti la fase progettazione.

1.3 Hardware e Software usati

1.3.1 Hardware & Sistema Operativo

Le macchine utilizzate per costruire il sistema sono piuttosto datate, e questo ha comportato notevoli sacrifici nella realizzazione, nonché anche una maggior attenzione sull’utilizzo della memoria. I computer usati sono:

	Acer Aspire 1800	HP Compaq nx7010
CPU	Intel Pentium 4 540	Intel Pentium M
Frequenza CPU	3.2 Ghz	1.5 Ghz
RAM	1 GB DDR	1 GB DDR
Sistema Operativo	Windows 7 Enterprise	Windows XP Professional (mod.)

1.3.2 Software

Per quanto riguarda la parte software, diversi e molteplici tool sono stati utilizzati.

In primis si vuole dire che è stato usato come interprete Prolog SWI-Prolog.

Seppur nota l’inferiorità computazionale rispetto a YAP (rapporto 1 : 4), SWI-Prolog ha una documentazione online molto più accessibile, scorrevole e meglio rappresentata di YAP,

oltretutto ha strumenti di debug e trace che a livello di Usabilità e UX sono molto più accettabili e più facilmente comprensibili rispetto a quelli di YAP (unicamente da linea di comando).

I tool software utilizzati sono:

- Interprete Prolog:
 - ✓ SWI-Prolog version 6.6.6 for i386-win32 -> <http://www.swi-prolog.org/>
- Ambiente di sviluppo:
 - ✓ Eclipse Luna EE Developers: <https://www.eclipse.org/downloads/>
 - ✓ Eclipse Kepler EE Developers: <https://www.eclipse.org/downloads/>
- Plugin SWI-Prolog for Eclipse:
 - ✓ Prolog Development Tool (IDE for Eclipse) - PDT
<http://sewiki.iai.uni-bonn.de/public-downloads/update-site-pdt/nightly/>

1.3.3 Modalità di sviluppo

Al fine di evitare notevoli quanto rovinosi accumularsi di versioni; e al fine di evitare il trasporto del workspace da un pc all'altro ho utilizzato approcci e risorse tipiche del Branch-programming con l'uso di Github e BitBuckets.

- Github: <https://github.com/dracoIntelligentSystem/ArtificialIntelligenceProject>
- Bitbucket: <https://bitbucket.org/dracoIntelligentSystem/support2expertsys>

1.4 Lancio del sistema e visualizzazione dei risultati

Per quanto concerne il lancio della computazione del sistema esso avviene tutto attraverso Eclipse attraverso la View Interface propria del plugin Prolog Development Tool (PDT).

Al suo interno presenta una schermata di output che sta a indicare la Console da linea di comando del prolog, solo che con questa modalità di interrogazione risulta più facile poter tenere sotto controllo il codice durante il run-time

Inoltre, dettaglio importante è l'AUTO(Re)CONSULT. Ogni qual volta si salvano le modifiche al codice prolog, verrà immediatamente fatto il Consult del file.pl e di tutti i moduli al suo interno connessi. Per un semplice consult si potrà eventualmente premere lo SHORTCUT F9 che consente la consultazione automatica, in caso di voler testare particolari/singole funzioni.

Per quanto concerne la visualizzazione dei risultati, dal fatto che si tratteranno matrici, la stampa in prolog è piuttosto fastidiosa ed è per questo che ho implementato un VISUALIZZATORE (supportSE.exe) scritto in C++ che permette di poter vedere da Prompt il risultato della computazione effettuata.

In aggiunta, al fine di una più corretta e comoda interazione con il sistema, è stata implementata una funzione di visualizzazione in PROLOG che consente di poter avere una panoramica del nostro ambiente grigliato, direttamente sulla CONSOLE di PROLOG.

Nella prossima sezione si esporranno cenni della programmazione logica, di PROLOG e una panoramica globale di SWI-Prolog, al fine di comprendere meglio questo interprete, che reputo a mio parere molto comodo e aumenta di molto la velocità di coding.

1.5 Considerazioni sull'interprete prolog

Infine vorrei poter dire che l'efficienza computazionale di YAP è di gran lunga superiore a quella di SWI-Prolog, e di utilizzare quest'ultimo solo per imparare e prendere padronanza con il paradigma e la sintassi; una volta accumulata una certa dimestichezza con il paradigma e il linguaggio, è meglio passare a YAP come interprete.

Capitolo 2.

Prolog e programmazione logica

2.1 Programmazione logica

La programmazione logica è un paradigma di programmazione che adotta la logica del primo ordine (FOL) sia per rappresentare che per manipolare l'informazione, utilizzando le clausole di Horn, nonché per elaborare teorie logiche e per compiere inferenze. La programmazione logica nasce all'inizio degli anni '70 grazie agli studi di due ricercatori, Kowalski e Colmerauer. Il primo elaborò le basi teoriche del paradigma, affermando la doppia interpretazione (procedurale e dichiarativa) delle clausole di Horn, che ha permesso l'implementazione su calcolatore dei linguaggi di programmazione logica. Il secondo progettò ed implementò un interprete per un linguaggio logico. Da queste basi nasce Prolog.

La programmazione logica appartiene alla famiglia dei **paradigmi di programmazione descrittivi**, nei quali la componente descrittiva (i dati sul quale il programma opera) è totalmente slegata dalla componente operativa (come i dati vengono elaborarli), che viene presa in carico totalmente dall'elaboratore.

Ciò permette all'utente di operare ad un livello di astrazione più alto rispetto ad un linguaggio imperativo, in quanto egli deve solo definire le specifiche del problema senza istruire la macchina sul "come" fare.

Nella programmazione logica, i programmi sono descrizioni delle soluzioni (goal) e non del processo solutivo attraverso il quale raggiungerlo. Tale programma viene elaborato descrivendo in un linguaggio formale (le clausole di Horn) il dominio applicativo (oggetti, relazioni, fatti, ecc).

Prolog è un linguaggio di programmazione logica adatto a problemi che riguardano oggetti, strutturati e non, e relazioni fra di essi. Per poter elaborare le informazioni richieste, il Prolog permette di interagire "rispondendo" alle domande che l'utente gli pone: difatti, sfruttando la base di conoscenza (che l'utente gli ha impartito), egli è capace di eseguire delle deduzioni che corrispondono alle risposte alle domande /richieste dell'utente.

In sintesi, le componenti fondamentali sono:

- Dichiarazione di fatti sugli oggetti e sulle loro relazioni
- Dichiarazioni di regole sugli oggetti e sulle loro relazioni
- Interrogazioni sugli oggetti e sulle loro relazioni

Le prime due componenti permettono di definire il dominio del problema (in maniera descrittiva), mentre l'ultima permette di eseguire il programma (in maniera operativa).

2.2 Sintassi prolog

Un programma Prolog è costituito da un insieme di clausole definite della forma:

- 1) $A.$
- 2) $A \text{ :- } B_1, B_2, \dots, B_n$

in cui A e B_i sono formule atomiche, A viene detta **testa** della clausola e la congiunzione B_1, \dots, B_n viene detto **corpo della clausola**. Una clausola come la 1 (ossia una clausola il cui corpo è vuoto) prende il nome di fatto (o asserzione), mentre una clausola come la 2 prende il nome di regola. Il simbolo “,” viene utilizzato per indicare la congiunzione mentre il simbolo “:-” indica l’implicazione logica. Una formula atomica è una formula del tipo: $p(t_1, t_2, \dots, t_m)$ in cui p è un simbolo di predicato e t_1, \dots, t_m sono termini. Ricordiamo la definizione di termine in modo ricorsivo:

- le **costanti** sono termini; in Prolog le costanti sono costituite dai numeri (interi e floating point) e dagli atomi alfanumerici (il cui primo carattere deve essere un carattere alfabetico minuscolo)
- le **variabili** sono termini; in Prolog le variabili sono stringhe alfanumeriche aventi come primo carattere un carattere alfabetico Maiuscolo oppure il carattere “_”
- $f(t_1, t_2, \dots, t_k)$ è un **termine** se f è un simbolo di funzione (o operatore) a **k argomenti** e t_1, \dots, t_k sono termini

Le **costanti** possono essere considerate come simboli funzionali a zero argomenti, come ad esempio *gino*, *pluto*, *pippo*, *a91*, *10.134*.

Le **variabili** sono caratterizzate dalla prima lettera maiuscola, come ad esempio *X*, *X1*, *Pippo*, *x*, (quest’ultima, definita con il solo carattere “_”, prende il nome di variabile anonima).

Una **costante** può essere anche composta da un programma prolog o da clausole definite.

Un **goal** (o **query**) ha come forma generale:

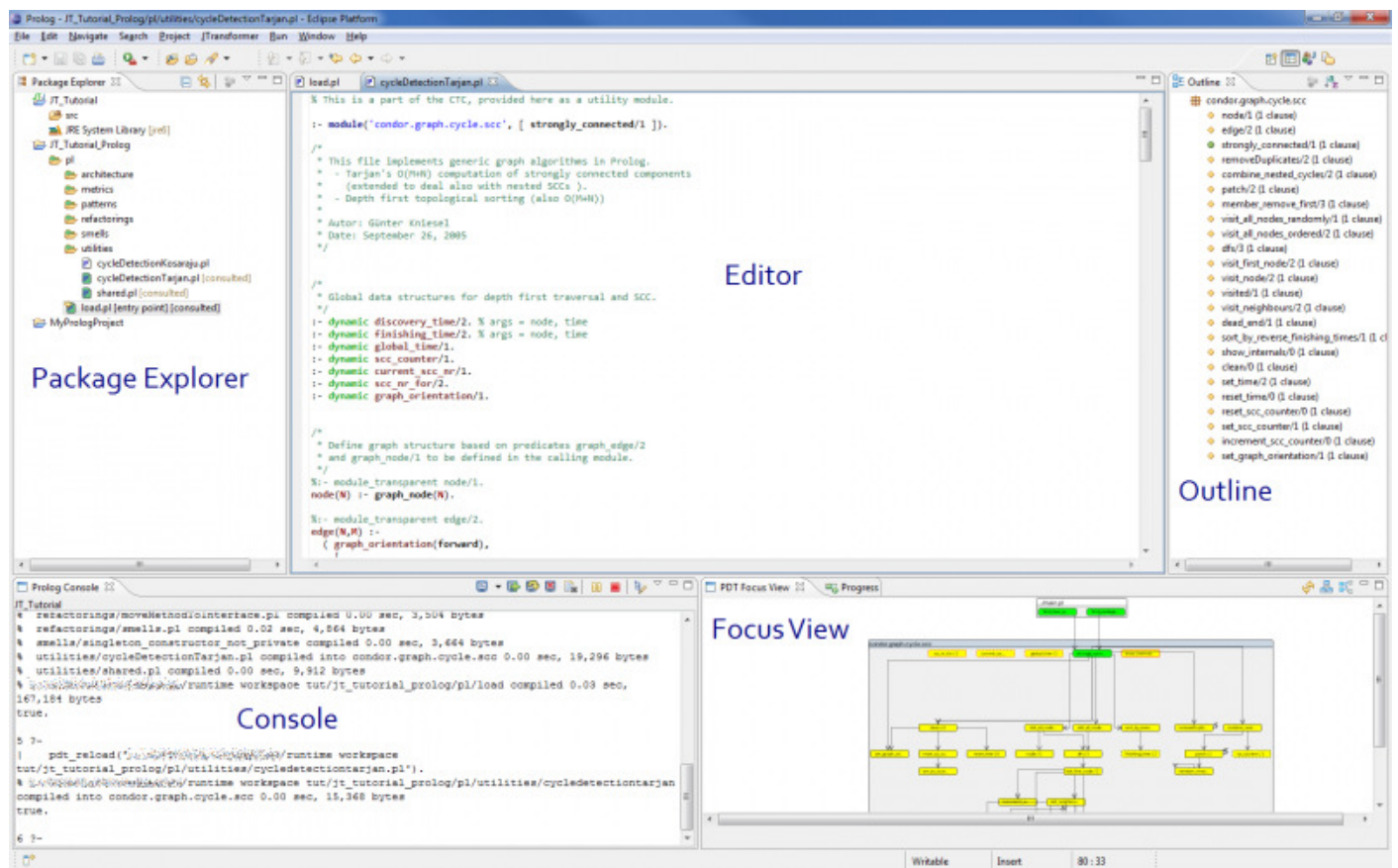
$\text{:- } B_1, B_2, \dots, B_n$, dove B_1, \dots, B_n sono **formule atomiche**.

Il problema di individuazione di moduli e predicati utilizzati da un dato modulo o predicato è riconducibile ad un problema di scoperta dei nodi adiacenti in un grafo ciclico, a partire da determinati nodi.

2.3 Interpretare logico SWI-PROLOG

In questa sezione saranno esposte, in linea del tutto generale, alcune qualità e interfacce che SWI-Prolog possiede, cercando di far evincere l'utilità del Tool.

Si mostra la panoramica generale del tool:

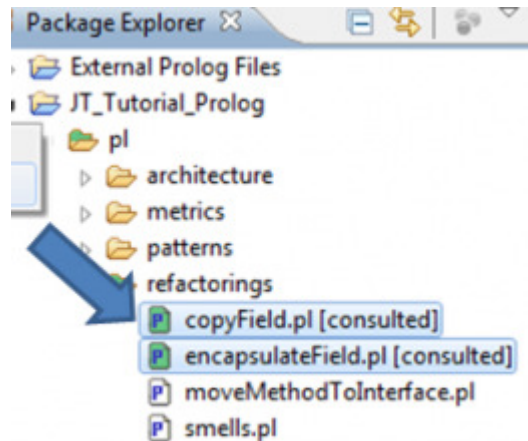


Qui è possibile notare come l'interfaccia di eclipse e il PROLOG PROSPECTIVE siano molto espressivi e orientati a un rapido apprendimento dell'interfaccia, garantendo un rapido apprendimento del tool.

Si ha:

- PROJECT EXPLORER

Dove è possibile navigare all'interno del General project, distinguendo file.pl CONSULTati (con icona verde) e quelli non consultati (con icona bianca).



- PROLOG EDITOR

La sezione dove propriamente si scrive codice Prolog.

Vengono forniti degli indicatori in situazioni di NON correttezza sintattica o strutturale del codice, oltre che ad alcuni warning che sicuramente verranno sollevati anche durante la consultazione.

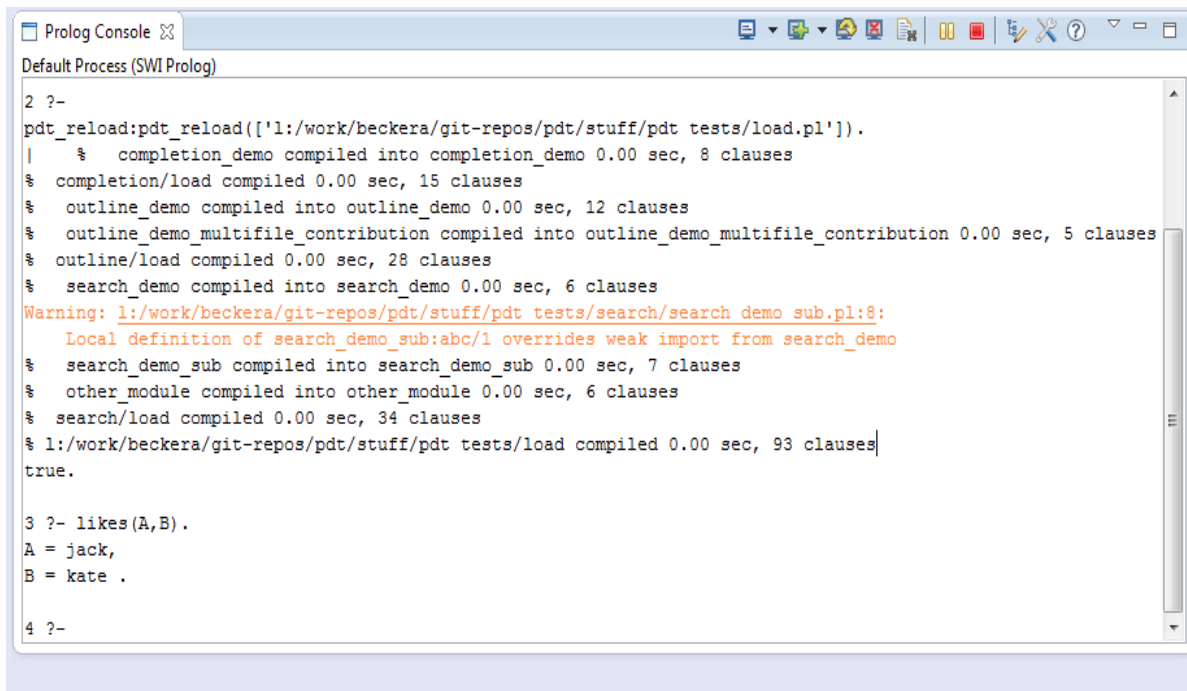
Oltretutto, il codice e eventuali manipolazioni, vengono sempre evidenziate con degli Highlights al fine di avere una miglior comprensione (successiva) su ciò che si è andato a scrivere; permettendo una rapida comprensione delle tipologie di keyword che si è andati ad utilizzare. Ne riporto una tabella

Highlighting	Indicated Property
<code>likes(Y, X)</code>	Predicato Dinamico
<code>atomic('')</code>	Predicato Built-in
<code>call(A)</code>	Meta predicato
<code>clause(Head, Body)</code>	Modulo predicato trasparente
<code>pair(X, Y)</code>	Predicato semplice
<code>'atom'</code>	Atomo
<code>% True iff</code>	Commento

- PROLOG CONSOLE

Corrisponde alla console di Prolog che si avrebbe lanciandola da linea di comando.

Avendola direttamente a disposizione nell'ambiente di sviluppo, semplifica di molto le cose e velocizza la fase di realizzazione e test. Incrementando il Fast Coding Process



```
Prolog Console
Default Process (SWI Prolog)

2 ?-
pdt_reload:pdt_reload(['1:/work/beckera/git-repos/pdt/stuff/pdt tests/load.pl']).
|   % completion_demo compiled into completion_demo 0.00 sec, 8 clauses
% completion/load compiled 0.00 sec, 15 clauses
% outline_demo compiled into outline_demo 0.00 sec, 12 clauses
% outline_demo_multifile_contribution compiled into outline_demo_multifile_contribution 0.00 sec, 5 clauses
% outline/load compiled 0.00 sec, 28 clauses
% search_demo compiled into search_demo 0.00 sec, 6 clauses
Warning: 1:/work/beckera/git-repos/pdt/stuff/pdt tests/search/search_demo_sub.pl:8:
Local definition of search_demo_sub:abc/1 overrides weak import from search_demo
% search_demo_sub compiled into search_demo_sub 0.00 sec, 7 clauses
% other_module compiled into other_module 0.00 sec, 6 clauses
% search/load compiled 0.00 sec, 34 clauses
% 1:/work/beckera/git-repos/pdt/stuff/pdt tests/load compiled 0.00 sec, 93 clauses
true.

3 ?- likes(A,B).
A = jack,
B = kate .

4 ?-
```

- SWI PROLOG DEBUGGER

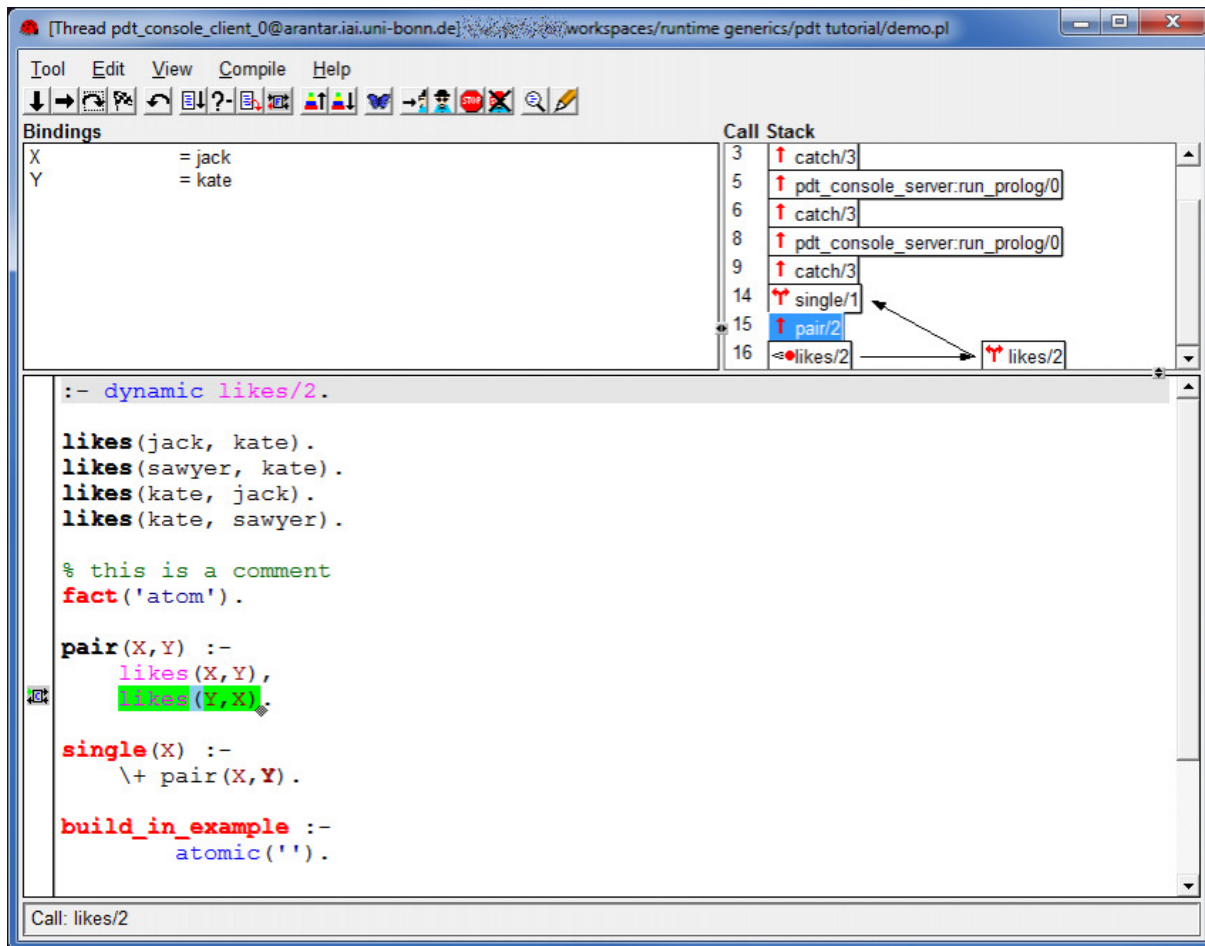
Questa è la componente che più ho utilizzato (insieme al predicato *trace*), l'utilità di questa schermata è che era possibile visualizzare comodamente, attraverso un interfaccia grafica, lo stato di avanzamento direttamente nel codice e poter visualizzare in tempo reale, lo stack delle chiamate (così da permettermi di tenere sotto controllo eventuali chiamate ricorsive) e l'aggiornamento delle variabili durante i processi di unificazione o Pattern Matching.

Oltretutto nel mio caso, dovendo gestire matrici, con doppio click sul termine/variabile all'interno della sezione *Bindings* si apriva una schermata che permetteva la completa visualizzazione della struttura dati.

Con l'utilizzo degli short cut (Space per riportarsi all'interno del predicato richiesto, evidenziando eventuali matching o miss matching) e (s minuscolo per proseguire a livello più alto dei predicati nella clausola [senza entrare nelle specifiche istruzioni di dettaglio]) si

velocizza di molto il trace&debug del codice, permettendo anche di velocizzare situazioni di controllo di predicati ricorsivi.

Se ne riporta una schermata, della finestra di debug:



Da notare che l'evidenziazione verde sta ad indicare il predicato che allo stato attuale deve essere verificato/computato, se andrà a buon fine (matching o verità) l'evidenziazione VERDE si sposterà verso il punto/virgola che sia; in caso di MISS matching l'evidenziazione diventerà di colore ROSSO, indicante una situazione di FALSE.

Capitolo 3.

Progettazione

3.1 Il problema e l'analisi

Il problema prevede l'esistenza di un ambiente, dove al suo interno siano contenuti gli elementi di interesse. Ricordando il goal del progetto:

“In un ambiente grigliato, vi è un agente e ostacoli (MOBILI e fissi), l'agente deve arrivare a un determinato punto TARGET evitando gli ostacoli mobili e naturalmente quelli fissi”.

Gli elementi rilevanti al fine del Goal sono rilevabili in:

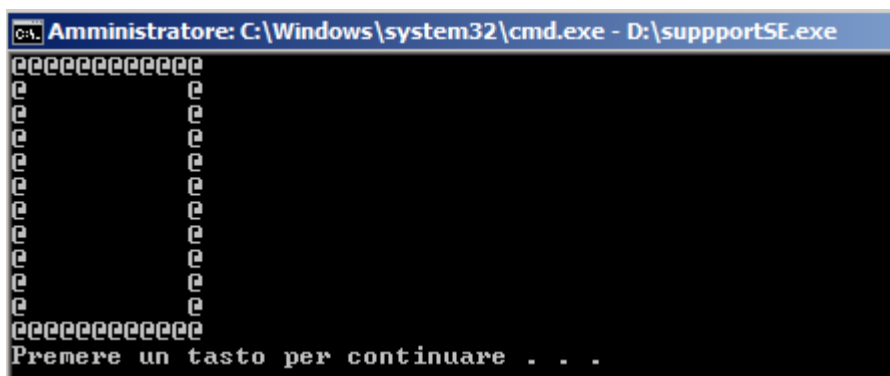
- Ambiente (la matrice dove si svolge il tutto)
- Gli ostacoli FISSI, che collettivamente e in posizioni tra loro adiacenti possono formare strutture tipo muri o stanze
- Gli ostacoli MOBILI che verranno identificati come agenti anch'essi
- L'agente che deve arrivare al TARGET(Neo)
- TARGET

3.1.1 L'ambiente

L'ambiente rappresenta il luogo dove gli ostacoli fissi, gli ostacoli mobili (agenti), e l'agente giacciono.

Trovandoci in un ambiente grigliato, l'ambiente è visto come una matrice (quadrata o rettangolare non importa); in questo progetto comunque è stata considerato il caso di costruzione di ambienti quadrati.

La costruzione è dinamica, ovvero non è presente nella base di conoscenza; l'ambiente viene generato a Run-Time dal sistema. Si mostra un esempio grafico proveniente da visualizzatore:



La costruzione dell'ambiente è la prima cosa che il sistema deve provvedere a fare.

3.1.2 Ostacoli Fissi

Questo tipo di ostacoli, non erano stati inizialmente previsti dal progetto originario, ma comunque ho voluto inserirli per dare al progetto una sorta di completezza operativa e di definizione.

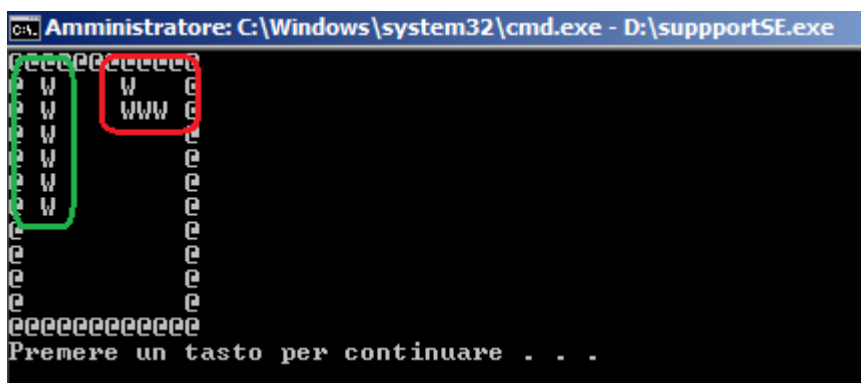
La loro caratteristica principale è quella di non poter compiere nessuno spostamento, quindi di poter interagire in modo attivo direttamente con gli agenti, rappresentano un punto di vista statico facente parte dell'ambiente.

Sono caratterizzati dal fatto di:

- Essere di un certo tipo, ostacoli_fissi,
- Da un identificatore numerico a livello di sistema
- identificativo a livello come carattere alfabetico comune a tutti ('W' che sta per Wall)
- Dal possedere una posizione all'interno dell'ambiente (X,Y), e tale posizione è immutabile.

L'unico tipo di interazione che possiedono questi ostacoli è da un punto di vista passivo, ovvero l'agente NEO o un ostacolo fisso può andare a sbatterci contro.

Si riporta un esempio di ambiente con soli ostacoli fissi, con uno screen del visualizzatore.



Come detto prima al 3.1, collettivamente gli ostacoli fissi possono identificare strutture tipo muri (in verde) oppure stanze (in rosso). Così facendo si ha modo di complicare ulteriormente ciò che potremmo definire uno *scenario di gioco*.

3.1.3 Ostacoli MOBILI (agenti)

Questo tipo di ostacoli sono caratterizzati dalla possibilità di movimento all'interno dell'ambiente, dando all'ambiente una caratteristica di dinamicità.

Per la realizzazione di questo tipo di ostacolo mi sono ispirato ad un gioco (Super Mario Bros) dove i nemici (ostacoli mobili) si muovono in una direzione (ma si svolge tutto su una dimensione) di un certo numero di posti(mattoni).

Spostandosi nell'ambiente possono interagire attivamente con Ostacoli fissi (scontrandosi), tra di loro (scontrandosi frontalmente o collidere) ; le situazioni di collisione vengono gestite dal sistema, dove per ogni situazione applica una procedura differente a seconda del tipo di collisione se frontale o laterale, e con il tipo di ostacolo con il quale si scontra.

Si spiegherà in dettaglio nel capitolo successivo

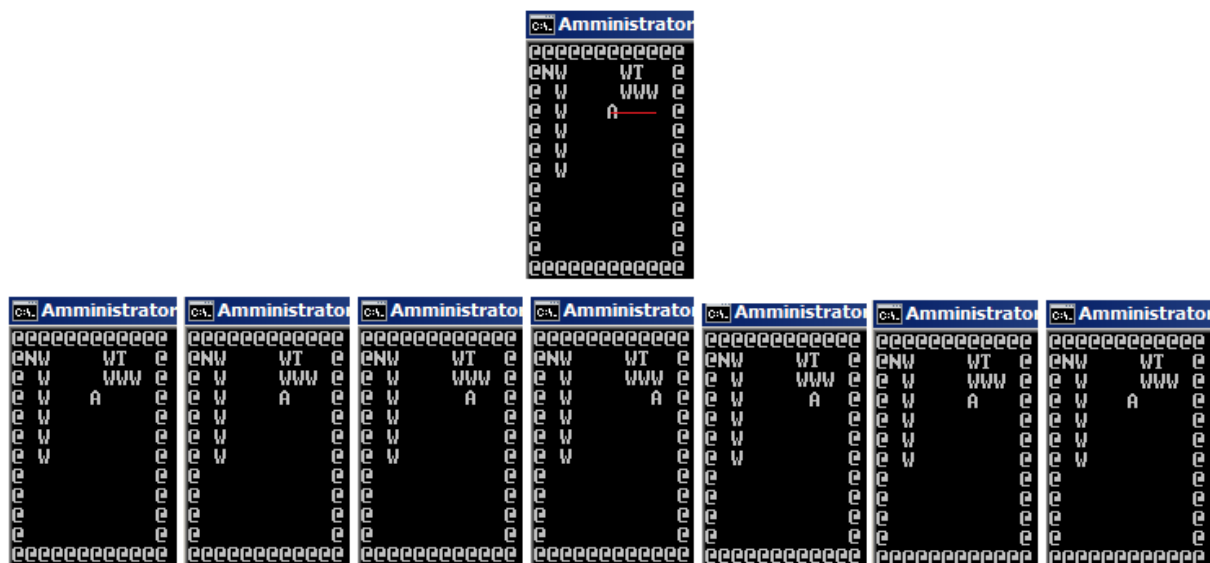
Le caratteristiche di interesse che sono state rappresentate sono le seguenti:

- Esplicitazione del tipo di ostacolo (è un agente)
- Un identificativo a livello di sistema (alfanumerico)
- Un identificativo a livello grafico (alfabetico) visualizzabile nell'ambiente, ogni agente ne ha uno diverso
- Le sue coordinate spaziale all'interno dell'ambiente (X,Y)
- La direzione verso cui spostarsi, una direzione cardinale
- Numero di passi che potrà compiere in quella direzione
- Numero di passi che potrà compiere per ogni step (indicante la velocità dell'agente (è stata fissata a 1 per il mio progetto)
- Il numero di passi che ha compiuto nella sua direzione

Quando l'ostacolo mobile (agente) avrà un numero di passi effettuati uguale al numero di passi

da compiere, verrà reimpostata la direzione verso cui muoversi nella direzione opposta all'attuale, in modo da poter tornare indietro (proprio come i funghi del super mario).

Si riporta un semplice esempio dal visualizzatore.



3.1.4 Agente (NEO)

L'agente intelligente NEO, può anch'esso essere inteso come un ostacolo mobile, e come per quest'ultimo condivide molte caratteristiche.

Il dettaglio principale che lo differenzia è il fatto che deve raggiungere un Target, una posizione obbiettivo: "Neo deve raggiungere il telefono per uscire da Matrix...".

Per poter arrivare al suo target, è necessario che l'agente impari un percorso ottimale, o per lo meno sappia in che direzione potersi muove al fine di fare una mossa (se NON ottima almeno buona).

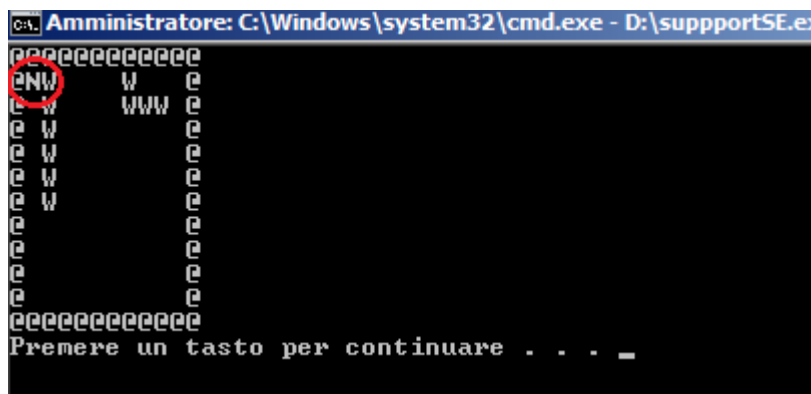
Inizialmente allo start del sistema, non viene specificata la direzione in cui NEO deve muoversi, l'agente in questione viene calato in un ambiente e deve poter capire come arrivarci.

Per trovare il percorso migliore, diversi algoritmi sono stati tentati, e altri sono stati realizzati.

L'agente intelligente (NEO), possiede molte caratteristiche in comune con l'ostacolo mobile, si elencano in basso:

- Dichiarazione del tipo di agente (Neo o eletto o TheOne)
- Un identificativo parola a caratteri alfabetici a livello di sistema (neo)
- Un identificativo a livello grafico (carattere alfabetico 'N') visualizzabile nell'ambiente, ne esiste solo uno
- Le sue coordinate spaziale all'interno dell'ambiente (X,Y)
- ~~• La direzione verso cui spostarsi, una direzione cardinale all'inizio non può saperlo~~
- ~~• Numero di passi che potrà compiere in quella direzione~~
- Numero di passi che potrà compiere per ogni step (indicante la velocità dell'agente è stata fissata a 1 per il mio progetto) paritariamente agli ostacoli mobili
- Il numero di passi che ha compiuto verso l'obiettivo (a scopo puramente indicativo)

Si riporta un esempio del visualizzatore di come appare l'agente intelligente (Neo) nell'ambiente:



3.1.5 Target

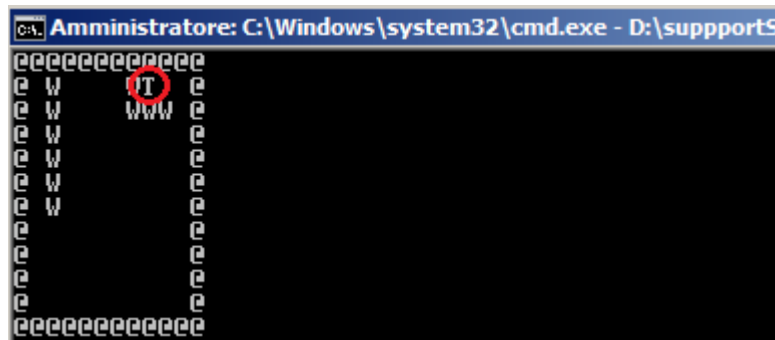
Riguardo al target, effettivamente c'è poco da dire, esso viene espresso con:

- Identificatore a livello di sistema,
- Identificatore a livello di visualizzazione (carattere alfabetico 'T'), unico nell'ambiente,

- Coppia di coordinate cartesiane all'interno della matrice/Ambiente.

Non rappresenta un oggetto in se per sé, esso più che altro rappresenta un punto di arrivo, il concetto di goal dell'agente.

Si riporta una grafica del visualizzatore per vedere come appare nell'ambiente come prima fatto per l'agente intelligente.



3.2 Strategia risolutiva e algoritmi

In questa sezione verranno descritte le procedure e gli algoritmi utilizzati dal sistema esperto, per permettere all'agente intelligente Neo di poter perseguire il proprio obiettivo (primario), ovvero sia quello di arrivare al target.

La procedura che avevo realizzato inizialmente si è rilevata inefficace per alcune situazioni ed è per questo che poi è stato preferito utilizzare un approccio più generale e validato.

Comunque il sistema al suo interno incorpora ancora il primo approccio, solo che è stato migliorato attraverso l'integrazione di algoritmi più robusti come il BFS e L'A* per il pathfinding.

3.2.1 Approccio iniziale-risoluzione dei triangoli

Nella fase iniziale del progetto, si era concepito l'agente come un'entità del tutto disinformata dell'ambiente e di ciò che conteneva.

Quindi l'agente intelligente non sapeva dove doveva andare, o meglio non conosceva la direzione primaria verso cui muoversi, ma doveva ricavarcela attraverso un procedimento computazionale.

Questo procedimento si sviluppava sulla conoscenza di 2 dettagli:

- La posizione dell'agente
- La posizione del target.

Compito che si poneva era di poter estrarre, tramite l'utilizzo della trigonometria e della risoluzione dei triangoli, la direzione cardinale verso cui procedere, e di eseguire il suo step.

La procedura di estrazione della direzione era relativamente molto semplice:

1. Prendeva in INPUT
 - Posizione (coordinate dell'ambiente) dell'AGENTE Neo
 - Posizione (coordinate dell'ambiente) del TARGET
2. Si calcola l'angolo in gradi sull'Agente
3. A ogni Punto cardinale (8 direzioni) corrispondeva una direzione POTENZIALE, verso cui compiere il proprio step.
4. Per ogni casella adiacente all'Agente si calcolava l'angolo tra la Potenziale Nuova posizione e il target e li si inseriva in una lista.
5. Si faceva un ranking sulla lista in ordine crescente, sulla base dell'angolo attuale tra agente-target e posizione potenziale-target

In questo modo si otteneva l'ordine di priorità con le direzioni che l'agente doveva spostarsi, per lo step attuale.

Dopodiché, a seconda che la cella fosse occupabile o meno, si spostava; in caso di cella occupata e quindi non occupabile, si faceva ricorsione sulla lista delle direzioni, cercando di spostarsi nella direzione successiva e quindi di miglior approssimazione al raggiungimento del target.

3.2.2 Breadth First Search – ricerca in ampiezza

Nella teoria dei grafi, la **ricerca in ampiezza** (in inglese *breadth-first search*, **BFS**) è un algoritmo di ricerca per grafi che partendo da un vertice (o nodo) detto *sorgente* permette di cercare il cammino fino ad un altro nodo scelto e connesso al nodo sorgente.

BFS è un metodo di ricerca non informato, ed ha il suo obiettivo quello di espandere il raggio d'azione al fine di esaminare tutti i nodi del grafo sistematicamente, fino a trovare il nodo cercato. In altre parole, se il nodo cercato non viene trovato, la ricerca procede in maniera esaustiva su tutti i nodi del grafo. Questo algoritmo non è di tipo euristico.

Il tempo di esecuzione totale di questo algoritmo è $O(V+E)$ dove V è l'insieme dei vertici del grafo ed E è l'insieme degli archi che collegano i vertici.

Questo algoritmo è stato applicato all'ambiente ma SOLO IN FASE INIZIALE, e nello specifico viene adoperato per stabilire, dopo aver inserito le coordinate di START e TARGET, SE avviare la computazione aveva senso oppure NO.

In altre parole il Sistema da me realizzato utilizza questo algoritmo non per cercare un cammino, ma per vedere se data la composizione STATICA all'interno dell'ambiente, L'AGENTE (Neo) può riuscire in qualche modo a raggiungere il target; quindi un ipotesi di raggiungibilità del GOAL.

Per questo motivo, essendo gli ostacoli mobili una componente dinamica dell'ambiente, non possono venir presi in considerazione, in quanto cambiando sempre posizione potrebbero in un dato momento occupare una posizione nell'ambiente, che potrebbe essere l'unico passaggio per arrivare al target.

Si riporta un esempio grafico di come il BFS viene applicato all'ambiente:

@	@	@	@	@	@	@	@	@	@	@	@
@	1	W	11	11	11	11	W	T	14	14	@
@	N	W	10	10	10	10	W	W	W	13	@
@	1	W	9	9	9	9	10	11	12	13	@
@	2	W	8	8	8	9	10	11	12	13	@
@	3	W	7	7	8	9	10	11	12	13	@
@	4	W	6	7	8	9	10	11	12	13	@
@	5	5	6	7	8	9	10	11	12	13	@
@	6	6	6	7	8	9	10	11	12	13	@
@	7	7	7	7	8	9	10	11	12	13	@
@	8	8	8	8	8	9	10	11	12	13	@
@	@	@	@	@	@	@	@	@	@	@	@

E di come apparirebbe se venissero presi in considerazione gli OSTACOLI MOBILI (A).

@	@	@	@	@	@	@	@	@	@	@	@
@	1	W	11	11	11	11	W	T		@	@
@	N	W	10	10	10	10	W	W	W	A	@
@	1	W	9	9	9	9	10	11	12	13	@
@	2	W	8	8	8	9	10	11	12	13	@
@	3	W	7	7	8	9	10	11	12	13	@
@	4	W	6	7	8	9	10	11	12	13	@
@	5	5	6	7	8	9	10	11	12	13	@
@	6	6	6	7	8	9	10	11	12	13	@
@	7	7	7	7	8	9	10	11	12	13	@
@	8	8	8	8	8	9	10	11	12	13	@
@	@	@	@	@	@	@	@	@	@	@	@

Il target come si vede non verrà mai raggiunto in quanto il passaggio risulta occupato dall'ostacolo mobile.

Questa situazione non ha senso ed è per questo che non vengono considerati.

3.2.3 A* - ricerca euristica

È un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo goal (o che passi un test di goal dato). Utilizza una "stima euristica" che classifica ogni nodo attraverso una stima della strada migliore che passa attraverso tale nodo. Visita il nodo in base a tale stima euristica. L'algoritmo A* è anche un esempio di ricerca best-first.

L'algoritmo è stato descritto nel 1968 da Peter Hart, Nils Nilsson, e Bertram Raphael.

A* comincia a partire dal nodo selezionato. Per ogni nodo è definito un costo di entrata (di solito zero per il nodo iniziale). A* allora valuta la distanza dal nodo meta a partire da quello corrente. Questa stima ed il costo assieme formano l'euristica che sarà assegnata al percorso passante per questo nodo. Il nodo è aggiunto allora a una lista, spesso chiamata "open".

L'algoritmo allora rimuove il primo nodo dalla lista (perché avrà valore della funzione euristica più basso). Se la lista è vuota, non ci saranno percorsi dal nodo iniziale al nodo meta e l'algoritmo si arresterà. Se il nodo è il nodo meta, A* ricostruisce e pone in output il percorso ottenuto e si arresta. Questa ricostruzione del percorso a partire dai nodi più vicini significa che non è necessario memorizzare il percorso in ogni nodo.

Se il nodo non è il nodo meta, nuovi nodi saranno creati per tutti i nodi vicini ammissibili; il modo di fare questo dipende dal problema. Per ciascun nodo successivo A* calcola il "costo" di entrata nel nodo e lo salva col nodo. Questo costo è calcolato dalla somma cumulativa dei pesi immagazzinati negli antenati, più il costo dell'operazione per raggiungere questo nuovo nodo.

L'algoritmo gestisce anche una lista di "closed", un elenco di nodi che sono già stati controllati. Se un nuovo nodo generato è già nella lista con un costo uguale o più basso, non ci sarà alcuna futura indagine su tale nodo o col percorso ad esso associato. Se un nodo nella lista di closed è uguale ad uno nuovo, ma è stato immagazzinato con un costo più alto, allora sarà rimosso dalla lista di closed, e il processo continuerà a partire dal nuovo nodo.

Successivamente, una stima della distanza dal nodo nuovo alla meta è aggiunta al costo per formare il suo valore euristico. Tale nodo è aggiunto allora alla lista "open", a meno che non esista un nodo identico con valore euristico minore o uguale.

L'algoritmo sarà adottato ad ogni nodo vicino, fatto ciò il nodo originale è preso dalla lista e posto nella lista di "closed". Il prossimo nodo sarà ottenuto dalla lista di open e con esso si ripeterà il processo descritto.

Questo è l'algoritmo usato per il Pathfindig ottimale, in quanto garantisce un ottima prestazione computazionale, NON deve visitare necessariamente tutti i nodi ma solo quelli che l'euristica suggerisce.

L'euristica che ho utilizzato si basa prettamente sulla distanza che intercorre in linea d'aria tra Agente (neo) e il target, lo spostamento verso una cella adiacente in qualsiasi direzione abbia sempre costo fisso pari a 1, dalla cella occupata.

Posto:

- ✓ $g(n)$: distanza percorsa dalla posizione iniziale sino al nodo/cella (n); all'inizio varrà **zero**, essendo ancora in posizione iniziale. Nel mio caso assumerà

sempre valori interi che coincideranno con il numero di passi compiuti dall'agente.

- ✓ $h(n)$: è la stima euristica della distanza tra il nodo/cella (n) e il TARGET; all'inizio avrà valore massimo. Utilizzando la distanza euclidea nel mio caso, compariranno valori in dominio Reale.
- ✓ $f(n)$: è la stima della distanza totale stimata dalla posizione iniziale dell'Agente (neo) sino al TARGET attraverso il nodo/cella (n). Tale stima viene calcolata come:

- $f(n) = g(n) + h(n)$

All'istante iniziale essa varrà:

- $f(\text{start}) = g(\text{start})[=0] + h(\text{start}) = h(\text{start})$

Nel mio procedimento solutivo, ho utilizzato l'approccio di cui al sotto paragrafo 3.2.1 unitamente al pathfinding dell'A*.

Primariamente si ottiene il RANK delle direzioni dove effettuare lo step, e successivamente si lancia l'algoritmo A* che restituisce il path migliore. Il passo successivo è il confronto tra la direzione della prima cella A*, con la prima direzione del vettore direzioni RANK; se coincidono viene eseguito lo step altrimenti viene lasciata priorità all'esecuzione del path derivato dall'A*.

Questo perché, qual'ora il Path A* dovesse essere inconcludente in quanto una cella chiave risulti occupata, entra in gioco e in applicazione il procedimento di risoluzione dei triangoli, che non computa un percorso MA LA SINGOLA DIREZIONE verso cui procedere, quindi ha un goal computazionale a breve termine (1 singolo step).

Si riporta graficamente un esempio di come l'A* opera:

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

3.3 Interazione con il Sistema Esperto

In questo sistema è possibile interagire attivamente, attraverso l'interprete prolog.

Si tratta prevalentemente di operazioni di inserimento e modifica dello stato dell'ambiente nel quale si sta operando:

- ❖ Caricamento della Base di Conoscenza, al cui interno sono descritti/rappresentati gli ostacoli FISSI e MOBILI (agenti) -> richiesto dal sistema
- ❖ Inserimento di coordinate iniziali dell'Agente (neo) -> richieste dal sistema
- ❖ Inserimento di coordinate del Punto di arrivo (TARGET) -> richieste dal sistema
- ❖ Inserimento di un ostacoli FISSO durante la computazione -> a discrezione dell'utente.

3.3.1 Immissione della Base di Conoscenza a Run-Time

Questa operazione è necessaria per il caricamento e il riempimento dell'ambiente.

In questo file, di cui va specificato l'absolut Path, sono descritte le proprietà degli OSTACOLI FISSI e OSTACOLI MOBILI.

Questi successivamente saranno inseriti nell'ambiente, inizialmente vuoto, e andranno a realizzare la situazione iniziale di partenza; in altre parole lo scenario nel quale l'Agente (neo) sarà immerso.

3.3.2 Inserimento delle coordinate AGENTE – TARGET

Una volta completato caricato lo “scenario” in base di conoscenza, il sistema richiede che siano inserite le posizioni INIZIALI dell'Agente (neo) e del TARGET che dovrà raggiungere.

Vengono effettuati alcuni controlli sul target e l'agente:

- Sull'AGENTE (neo):
 - Non deve avere una posizione già occupata da un OSTACOLO FISSO
 - Non deve avere una posizione già occupata da un OSTACOLO MOBILE
 - Non deve essere all'esterno dell'ambiente
 - Non deve essere adiacente a un OSTACOLO MOBILE
- Sul TARGET.
 - Non deve avere una posizione già occupata da un OSTACOLO FISSO
 - Non deve avere una posizione già occupata da un OSTACOLO MOBILE
 - Non deve essere all'esterno dell'ambiente
 - Non deve essere sulla traiettoria di NESSUN OSTACOLO MOBILE.

3.3.3 Inserimento di un Ostacolo Fisso a Run-Time

Questa forma di interazione con il sistema prevede la possibilità in OGNI STEP di esecuzione di poter modificare l'ambiente a RUN-TIME attraverso l'introduzione casuale di un ostacolo FISSO. Attraverso l'inserimento di opportune coordinate.

Anche per gli OSTACOLI FISSI, vengono eseguiti alcuni controlli per l'inserimento:

-
- Non deve avere una posizione già occupata da un altro OSTACOLO FISSO
 - Non deve avere una posizione già occupata da un'altro OSTACOLO MOBILE
 - Non deve essere all'esterno dell'ambiente
 - Non deve avere la posizione attuale dall'Agente (neo)
 - Non deve avere una la stessa posizione del TARGET

Capitolo 4.

Esecuzione del sistema

4.1 Panoramica

In questo capitolo si procederà alla spiegazione, anche e soprattutto in via pratica, di come avviene la computazione, facendo frequente riferimento all'implementazione.

Verranno esposti i diversi moduli (file.pl) e il ruolo che occupano all'interno del sistema.

Alla fine di tutto verranno esposte delle considerazioni e proposte idee per sviluppi futuri.

4.2 Strutturazione moduli

Il Sistema prevede l'utilizzo di diversi moduli, in precisione 9, quello di START e quelli funzionali.

Modulo start:

- Start.pl: questo modulo, come si evince dal nome, viene utilizzato per avviare la computazione di tutto il sistema, e la regola che viene utilizzata è *start*.

In questo modulo oltretutto esiste un'altra regola, anch'essa a zero argomenti *flush* utilizzato per pulire la memoria del Sistema da tutte le assunzioni e fatti della base di conoscenza(scenario) utilizzati e che il sistema durante la computazione, ha aggiunto e/o modificato.

Moduli funzionali, citati in ordine alfabetico e non di chiamata:

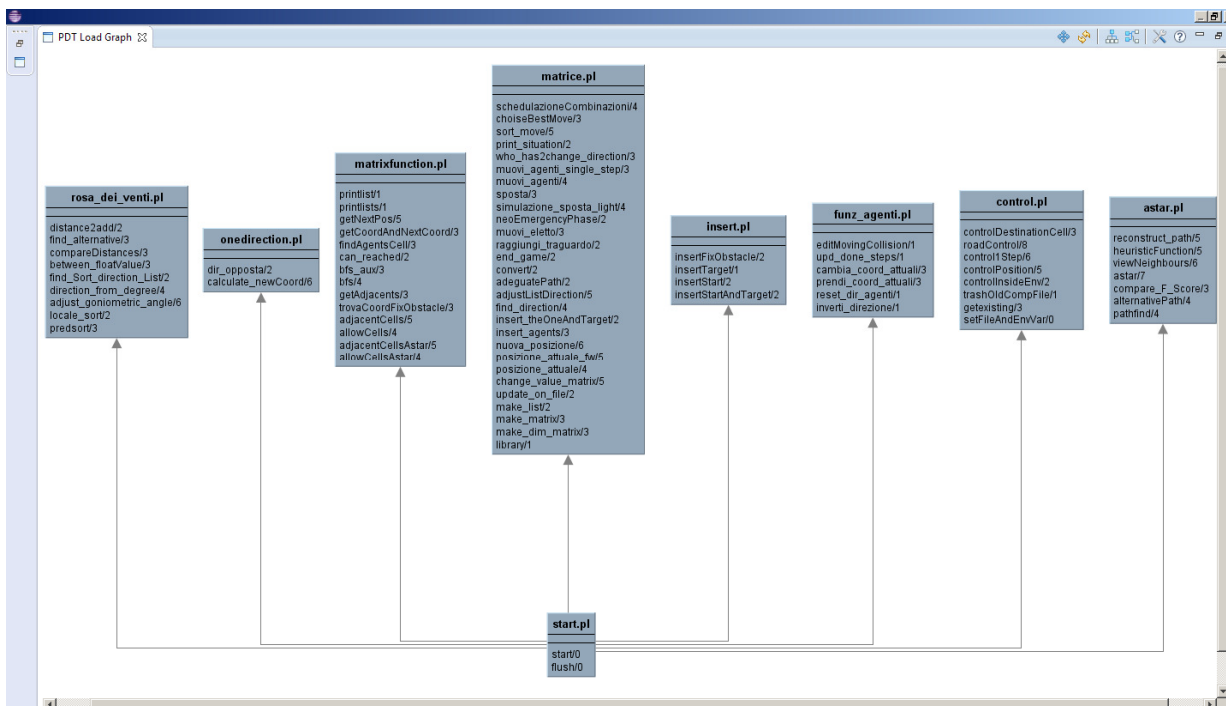
- astar.pl: utilizzato come modulo al cui interno sono inseriti le funzionalità di pathfinding (algoritmo A*) e tutti i metodi/regole ad esso correlati e utilizzati al suo interno; inoltre è presente la regola che permette la ricostruzione del Path (iniziale) che dovrà eseguire l'agente per arrivare al target, e la funzione EURISTICA;
- control.pl: sono presenti alcune regole di controllo come: verifica sui eventuali file di output presenti derivanti da precedenti computazioni e regole di gestione di questi a seconda della presenza o meno, regole per controllare

-
- (chiunque chiami questa regola) che le coordinate (argomenti in input) siano interni all'ambiente/matrice, controlli sulle posizioni all'interno dell'ambiente/matrice
- `funz_agenti.pl`: contiene le regole che permettono la manutenzione e l'aggiornamento delle entità/oggetti della base di conoscenza, e regole per ottenere informazioni (posizione, passi, etc.), regole per controllare eventuali collisioni tra agenti/ostacoli MOBILI
 - `insert.pl`: contiene le regole che consentono l'immissione di entità come (Agente(neo), TARGET, e ostacoli FISSI) e aggiornare l'ambiente/matrice
 - `matrice.pl`: questo è il modulo PIÙ importante di tutti, al suo interno sono presenti le regole di computazione ed esecuzione dell'intero sistema, QUI viene creata la matrice/ambiente e viene asserito un fatto con il quale si esplicitano le sue proprietà (dimensioni), sono presenti regole che permettono la modifica/aggiornamento dell'ambiente al procedere della computazione e di prendere informazioni sull'ambiente stesso (posizione in una determinata coordinata) e simulazioni di spostamenti, sono presenti regole che permettono un "buon intendersi" nei movimenti tra ostacoli MOBILI, sono presenti le regole che permettono il **movimento** degli OSTACOLI MOBILI e dell'agente NEO, è presente la **REGOLA DI GOAL**, e meta predicato di scrittura su file.
 - `matrixFunction.pl`: modulo di funzionalità operative, usato come modulo funzioni accessorie (realizzato in un secondo momento), contiene la regola BFS (`can_reached`) per capire se ha senso effettuale la computazione, regole per l'ottenimento di insiemi di coordinate (ostacoli fissi e mobili), regole di supporto all'algoritmo A* per il ritrovamento di adiacenti. Contiene anche regole per permettere una stampa a video su terminale

dell'ambiente, per poter vedere a Run-Time come evolve la situazione.

- **oneDirection.pl:** contiene una sorta di CONOSCENZA A PRIORI che il sistema possiede, in forma di fatti e regole, dove abbiamo la specifica delle direzioni e delle rispettive direzioni opposte sottoforma di fatti, e una regola che sviluppa il movimento in una specifica direzione.
- **rosa_dei_venti.pl:** questo modulo realizza in buona parte il primo approccio da me seguito, quello di RISOLUZIONE DEI TRIANGOLI, anch'esso possiede della CONOSCENZA A PRIORI espressa sottoforma di fatti (intervalli angolari per la sezione di una direzione, con angolo secco), sono presenti delle regole di trasformazione degli angoli, e la regola che permette di “rankizzare” le direzioni in funzione degli angoli (adequati) con l'angolo TARGET

Si riporta per completezza la schermata grafica restituita dal Plugin di eclipse – PROLOG DEVELOPMENT TOOL con lo schema dei diversi moduli, con all'interno le regole presenti e la relativa ARIETÀ:



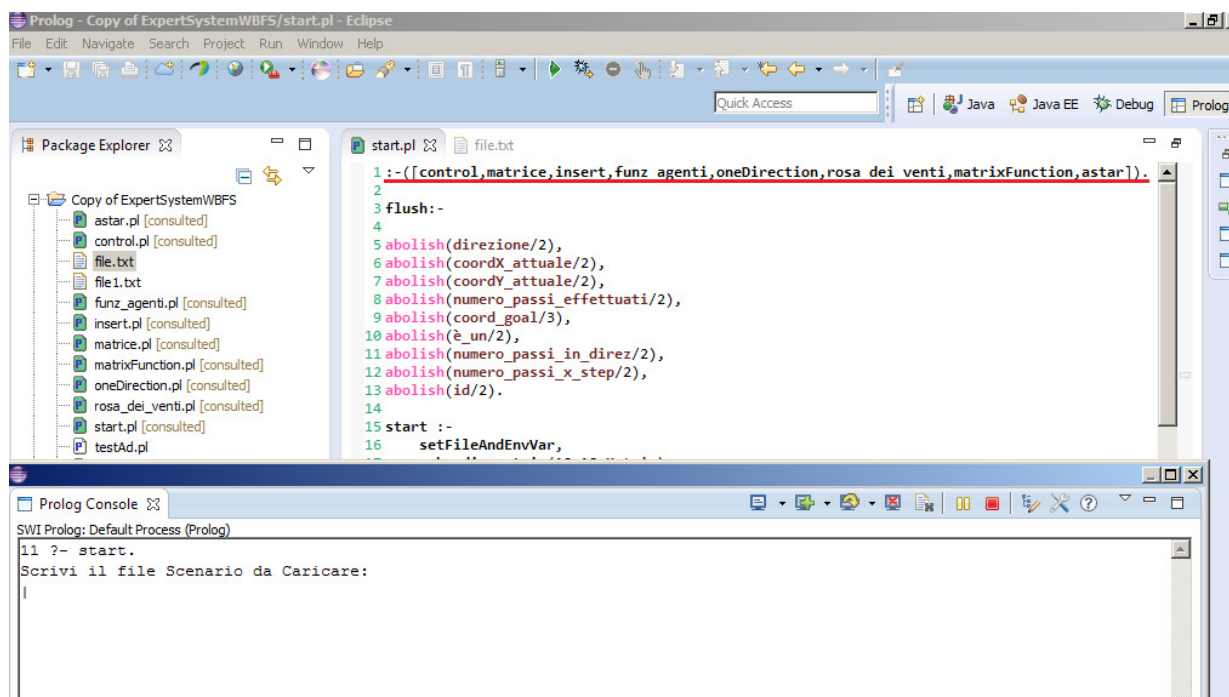
4.3 Computazione

Come detto già prima verrà utilizzato il terminale di SWI-Prolog all'interno di eclipse, fornito dal plugin PDT.

Nel paragrafo precedente si è detto che tutto parte con il modulo START.pl, e in precisione lanciando da terminale la regola START.

Dando invio, produce queste azioni:

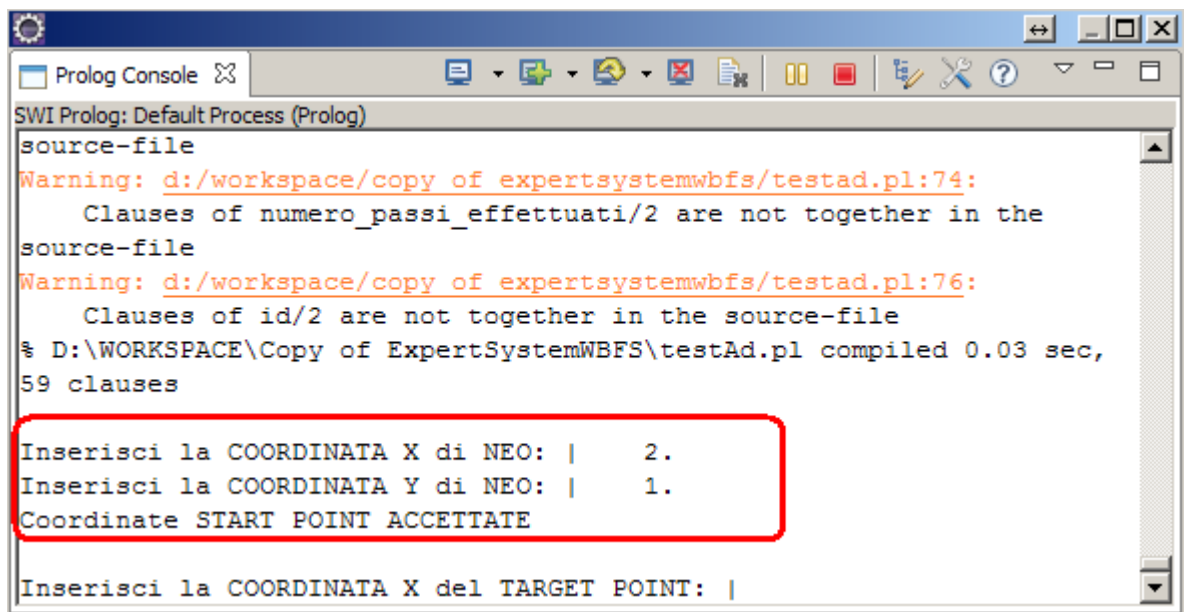
- CONSULT DI TUTTI I MODULI
- Ritrovamento e cancellazione di file derivanti da altre precedenti computazioni
- Inizializzazione di un nuovo file di testo (file.txt)
- Costruzione dell'ambiente/matrice
- Salvataggio dell'ambiente costruito con scrittura sul file di testo
- Richiesta del absolute path con la BASE DI CONOSCENZA (scenario) da caricare.



La base di conoscenza (scenario) ha un absolute path:

'D:\\WORKSPACE\\ExpertSystemWBFS\\testAd.pl'.

Dopo aver fatto il CONSULT del file passato, e caricato gli OSTACOLI FISSI E MOBILI, e loro posizioni interne all'ambiente, il sistema avvierà la seconda fase di interazione con l'utente chiedendo di inserire le COORDINATE DI START e TARGET.



```
SWI Prolog: Default Process (Prolog)
source-file
Warning: d:/workspace/copy of expertsystemwbfs/testad.pl:74:
    Clauses of numero_passi_effettuati/2 are not together in the
source-file
Warning: d:/workspace/copy of expertsystemwbfs/testad.pl:76:
    Clauses of id/2 are not together in the source-file
% D:\WORKSPACE\COPY of ExpertSystemWBFS\testAd.pl compiled 0.03 sec,
59 clauses

Inserisci la COORDINATA X di NEO: |    2.
Inserisci la COORDINATA Y di NEO: |    1.
Coordinate START POINT ACCETTATE

Inserisci la COORDINATA X del TARGET POINT: |
```

Se le coordinate inserite non saranno ritenute valide, o perché la cella è occupata o perché esterne dall'ambiente o perché (come nel caso del TARGET) è sulla strada di percorrenza di un agente, verrà chiesto se si desidera REINTRODURRE le coordinate al fine di poter cominciare una computazione (a seconda dell'entità a cui si riferiscono le coordinate, verrà chiesta la reintroduzione; se non vengono accettate quelle dell'Agente(neo) saranno richieste solo le sue, in caso riguardino le coordinate del TARGET, verranno richieste solo le Sue senza elidere quelle precedenti). Si mostra un esempio di coordinate sbagliate (caso in cui si indichi il target occupante la posizione [1,7] con una occupata da un ostacolo fisso).

```
SWI Prolog: Default Process (Prolog)

% D:\WORKSPACE\Copy of ExpertSystemWBFS\testAd.pl compiled 0.03 sec,
59 clauses

Inserisci la COORDINATA X di NEO: |    2.
Inserisci la COORDINATA Y di NEO: |    1.
Coordinate START POINT ACCETTATE

Inserisci la COORDINATA X del TARGET POINT: |    1.
Inserisci la COORDINATA Y del TARGET POINT: |    7.
Coordinate del TARGET POINT NON CORRETTE. Vuoi inserirne di
nuove? (y/n)
|    Y.

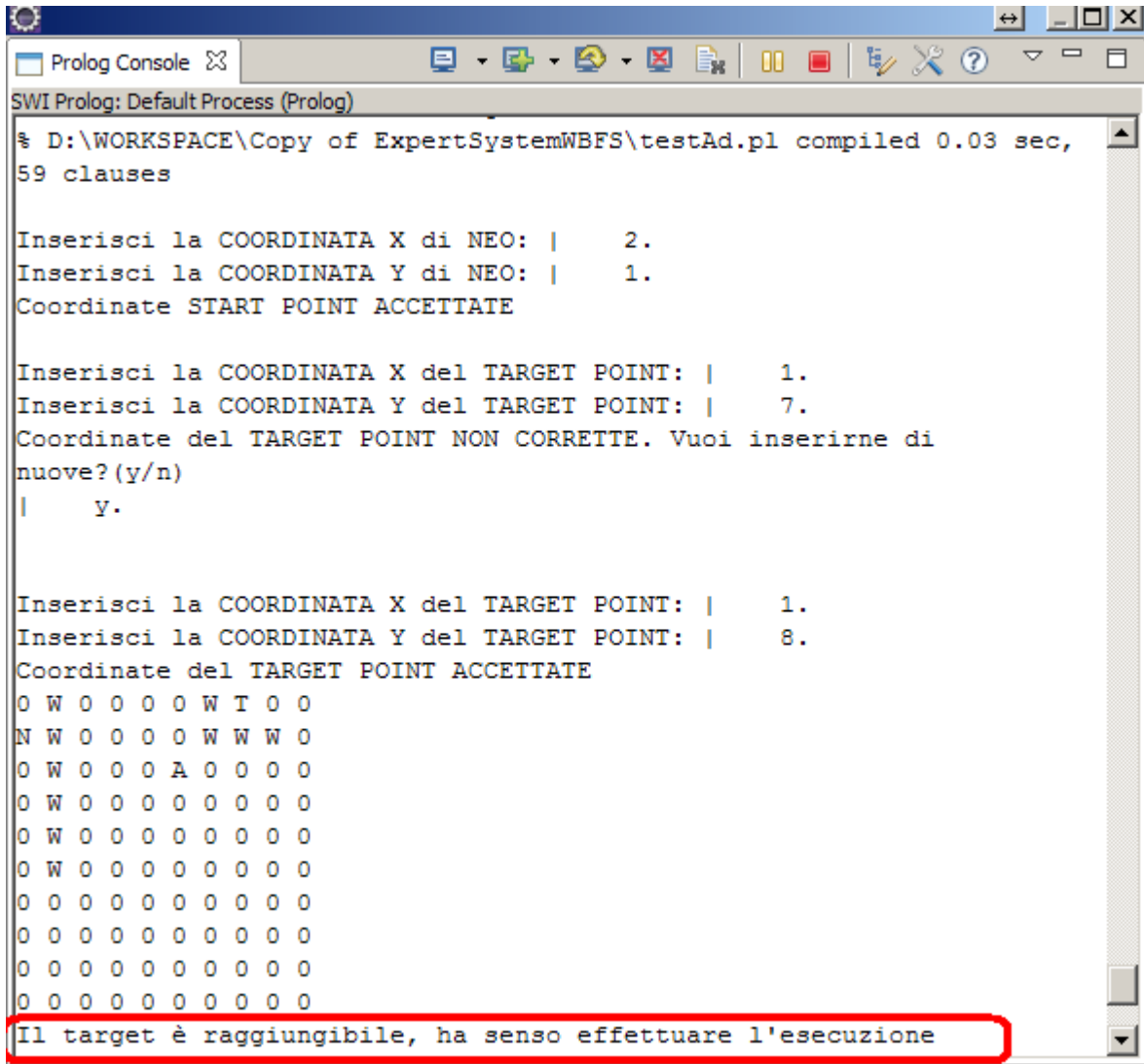
Inserisci la COORDINATA X del TARGET POINT: |    1.
Inserisci la COORDINATA Y del TARGET POINT: |    8.
Coordinate del TARGET POINT ACCETTATE

O W O O O O W T O O
N W O O O O W W W O
O W O O O A O O O O
O W O O O O O O O O
O W O O O O O O O O
O W O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
```

Dopo aver reintrodotta le coordinate SOLO DEL TARGET, il sistema inserisce nell'ambiente (già precedentemente ha effettuato l'introduzione nell'ambiente di OSTACOLI FISSI & MOBILI(agenti)) viene stampata a video (su CONSOLE di SWI PROLOG) l'ambiente/matrice.

A questo punto procede ad aggiornare i FILE di OUTPUT necessari, e come controllo viene innescata la REGOLA DI RAGGIUNGIBILITA' necessaria per sapere se ha senso effettuare la computazione, ovvero se:ESISTE (e NON qual è) un percorso con cui l'Agente (neo) riuscirà ad arrivare al TARGET, questo è dato dal predicato *can_reached(neo,target)*, che matchando con una regola di controllo, eseguirà il BFS riuscendo in pochissime iterate (ma purtroppo visitando moltissime celle) a capire se esiste una via percorribile per cui abbia senso effettuare la computazione.

Alla fine viene restituito un FEED a video nel quale si comunica il SUCCESSO o l'INSUCCESSO della procedura.



```
SWI Prolog: Default Process (Prolog)
% D:\WORKSPACE\Copy of ExpertSystemWBFS\testAd.pl compiled 0.03 sec,
59 clauses

Inserisci la COORDINATA X di NEO: |    2.
Inserisci la COORDINATA Y di NEO: |    1.
Coordinate START POINT ACCETTATE

Inserisci la COORDINATA X del TARGET POINT: |    1.
Inserisci la COORDINATA Y del TARGET POINT: |    7.
Coordinate del TARGET POINT NON CORRETTE. Vuoi inserirne di
nuove?(y/n)
|    y.

Inserisci la COORDINATA X del TARGET POINT: |    1.
Inserisci la COORDINATA Y del TARGET POINT: |    8.
Coordinate del TARGET POINT ACCETTATE
O W O O O O W T O O
N W O O O O W W W O
O W O O O A O O O O
O W O O O O O O O O
O W O O O O O O O O
O W O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
Il target è raggiungibile, ha senso effettuare l'esecuzione
```

Fatto ciò il passo immediatamente successivo è quello di dotare l'Agente(neo) di un Path, primario, da eseguire, che sia ottimo.

Questo path viene ricavato attraverso il lancio del predicato *pathfind/4* che, attraverso l'algoritmo A*, restituirà il path ottimo e lo fornirà in input alla procedura che permetterà di avviare TUTTA LA COMPUTAZIONE; tale procedura ha nome *muovi_agenti/4* e corrisponde alla regola più importante di tutto il sistema perché scandisce gli step; ogni sua completa esecuzione stabilisce una soluzione intermedia.

MOVIMENTO degli OGGETTI

La prima cosa che questa regola (*muovi agenti*) esegue è lo spostamento di tutti gli OSTACOLI MOBILI (agenti), ma non può farlo casualmente, ma non può farlo in modo diretto perché possono venir a verificarsi determinate situazioni, quali:

- Esaurimento dei passi percorribili in una direzione
- Collisione NON FRONTALE tra 2 agenti
- Collisione FRONTALE tra 2 agenti
- Collisine tra agente e OSTACOLO FISSO

Per ognuna di queste situazioni esiste un apposita procedura di comportamento che riguarda gli OSTACOLI MOBILI.

4.3.1 Esaurimento dei passi percorribili in una direzione

Questo è il primo controllo che viene effettuato su TUTTI gli OSTACOLI MOBILI, esso estrae la lista di tutti gli agenti presenti e verifica con una regola *who_has2change_direction* che il numero di passi compiuti in direzione sia uguale al numero di passi compiuti in quella direzione, in questo caso attraverso la conoscenza a priori del sistema sulle coordinate cartesiane (*rosa_dei_venti.pl*) ritratta il fatto caratterizzante la direzione dell'agente preso in considerazione allo stato attuale e asserisce la sua direzione in quella opposta alla precedente, resettando il numero di passi compiuti (*inverti_direzione*).

4.3.2 Collisione NON FRONTALE OSTACOLI MOBILI (agenti)

In questo caso il controllo che si fa è quello di vedere se 2 o più agenti cercano di occupare la stessa casella, ma aventi direzioni che non si sovrappongono tra loro ma si INTERSECANO, quindi scontri non FRONTALI.

La necessità di gestione di questo tipo di collisioni è consigliata, per una miglior chiarezza visiva.

La regola che è utilizzata per questo tipo di controllo è *choiseBestMove*, che attraverso una lista di tutte le possibili permutazioni della lista agenti, riesce a restituire una lista ordinata, di QUALE AGENTE FAR MUOVERE PER PRIMO, quale per secondo e così via.

Il criterio di ordinamento sugli spostamenti degli Ostacoli mobili è data dalla quantità di agenti che riescono a compiere uno spostamento. Più chiaramente verrà preferita quella permutazione con la quale si potranno spostare il maggior numero di ostacoli MOBILI (agenti).

Regole utilizzate: *choiseBestMove*, *schedulazione Combinazioni*, *sort_move*, *simulazione_sposta_light*.

4.3.3 Collisione FRONTALE tra 2 OSTACOLI MOBILI (agenti)

Questa situazione prevede, rispetto alla precedente, una condizione in più; ovvero, che 2 ostacoli mobili abbiano parte del loro percorso in comune, e cioè si muovono in direzioni tra loro opposte.

Questo comporta un controllo ulteriore, e nello specifico (regola *controlDestinationCell*) sapere attraverso un osservazione dell'ambiente, se la potenziale cella di destinazione di un ostacolo mobile è già occupata da un altro ostacolo mobile. E se ciò risultasse vero CONTROLLARE che l'ostacolo fisso in rotta di collisione abbia una direzione opposta alla propria; si hanno due situazioni:

- L'agente in rotta di collisione HA DIREZIONE OPPOSTA alla mia
- L'agente in rotta di collisione HA ALTRA DIREZIONE

nel secondo caso si riporterebbe alla situazione espressa nel precedente 4.3.3; nella primo caso invece, comporta l'applicazione di una "procedura di emergenza" che attraverso delle regole vada a modificare il comportamento degli ostacoli mobili (*editMovingCollision*), correggendo il numero di passi effettuabili in una direzione al massimo consentito, ovvero gli attuali compiuti, per entrambi gli agenti.

4.3.4 Collisine tra agente e OSTACOLO FISSO

Questa è una casistica particolare della precedente, ed è appunto la situazione in cui la cella che potenzialmente si vuole occupare è già presidiata da un OSTACOLO FISSO.

In tal caso la procedura e la regola è identica alla precedente, solo che in primis si verifica, osservando l'ambiente che si stia collidendo con un OSTACOLO FISSO, in secundis si adegua il numero di passi effettuabili nella direzione attuale a quelli effettivamente compiuti.

Allo spostamento di un SINGOLO AGENTE, viene salvato su di un file (MOVEDetails.txt) tutto l'ambiente, al fine di poter vedere come le SINGOLE ENTITA' si sono mosse.

Una volta effettuata lo spostamento di tutti gli ostacoli mobili presenti nell'ambiente, sarà la volta di MUOVERE L'AGENTE(NEO) attraverso l'applicazione della regola MUOVI_ELETTA.

Ma prima di poter effettuare il corretto spostamento, è necessario dapprima compiere una fondamentale operazione: l'Agente(neo) deve poter capire in che direzioni muoversi, in modo da poter effettuare il MIGLIOR spostamento.

Ricordiamo quanto detto precedentemente, che tra gli input di questo metodo, c'era anche un Path ricavato in fase iniziale dalla regola *start*.

La prima cella del path verrà data in input a una regola *find_direction* che restituirà una lista ordinata delle direzioni, a seconda dell'angolazione che le potenziali celle (ciascuna per ogni direzione) hanno rispetto al target e della differenza tra queste angolazioni e quella tra l'agente e il target.

La lista sarà ottenuta con i seguenti steps:

- ✓ rank delle celle nei diversi punti cardinali, con il metodo della RISOLUZIONI DEI TRIANGOLI.
- ✓ si prende la cella passata in INPUT (la prima posizione da raggiungere) ottenuta con il metodo dell'A*,
- ✓ si confronta se la direzione in cui si trova la cella A* è identica alla prima direzione presente nella lista ordinata di RISOLUZIONE DEI TRIANGOLI.

Nel caso in cui le direzioni COINCIDANO, il sistema restituisce la *FINALDIRECTIONLIST*, caso contrario, con l'uso della regola *adjustListDirection*, preleva all'interno della lista ordinata delle direzioni (Ris.Trinagoli) l'elemento-Direzione corretto e lo pone in testa alla LISTA. Cosicché l'algoritmo A*(calcolato senza ostacoli MOBILI *NORMALPHASE*) possa avere maggior peso decisionale sulla direzione da intraprendere.

Fatto tutto ciò, adesso abbiamo la lista delle direzioni ordinate, con cui l'Agente(Neo) potrà muoversi. Ecco le fasi salienti per quanto riguarda il suo comportamento di spostamento:

- asserzione di un fatto, con il quale si stabilisce che NEO si muove nella prima direzione (testa della lista precedentemente computata),
- simulazione di esecuzione dello spostamento dell'Agente lungo la direzione, con composizione di un Ambiente/matrice virtuale/FINTA.
- Dato che MUOVONO PRIMA GLI OSTACOLI MOBILI e POI l'AGENTE (NEO) simulo lo spostamento successivo che sarà fatto dagli OSTACOLI MOBILI.
- Controllo che NESSUN OSTACOLO MOBILE, riesca a scontrarsi con l'agente(NEO), attraverso l'uso di una variabili (neo_intercept)

Due situazioni si possono creare sulla variabile NEO_INTERCEPT:

1. il valore rimane FALSE, l'eletto non sarà MAI INTERCETTATO:

- reset delle direzioni originarie degli OSTACOLI MOBILI
- compirà lo spostamento "REALE",
- Aggiornamento dell'AMBIENTE,
- scrittura su 2 FILE (decisionDir.txt e movedetails.txt) dell'ambiente Nuovo/Aggiornato
- cancellazione della direzione attualmente intrapresa dall'Agente(NEO)

2. il valore diventa TRUE, l'eletto sarà INTERCETTATO:

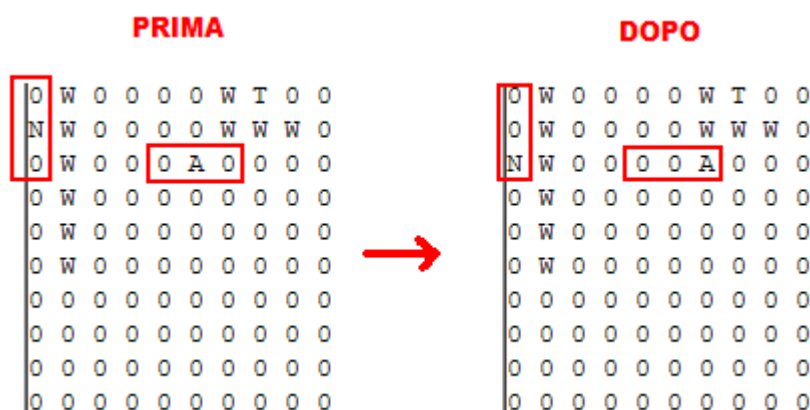
- reset delle direzioni originarie degli OSTACOLI MOBILI
- attivazione della procedura di emergenza *neoEmergencyPhase*, la quale cancella la direzione attualmente intrapresa dall'Agente(NEO) e attraverso una NUOVA ESECUZIONE dell'algoritmo A* **questa volta**

calcolato considerando ANCHE GLI OSTACOLI MOBILI (AGENTI),

- riordinamento della restante LISTA DIREZIONI data in input alla regola MUOVI_ELETTTO, senza ovviamente la direzione che ha portato a questa situazione,
- prima direzione alternativa UTILE è quella restituita dalla direzione in cui si trova la Prima Posizione restituita dall'A* (*EmergencyPath*)
- ri-esecuzione (ricorsiva) della regola *muovi_eletto*.

Compiuto finalmente lo spostamento dell'AGENTE(NEO), la regola *muovi_agent* procede, attraverso un controllo, quello se le coordinate (e quindi la direzione) ricevute in input, sono le stesse in cui attualmente si trova l'Agente(NEO); in caso di non corrispondenza vuol dire che si è resa necessaria l'esecuzione di una procedura di emergenza, e che quindi si rende obbligatoria la riscoperta del cammino MIGLIORE, attraverso una riapplicazione dell'A* in situazione di NORMALPHASE, ovverossia SENZA CONSIDERARE gli OSTACOLI MOBILI; dovesse esserci ancora un intercettazione si potrà riapplicare ALL'OCCORRENZA la procedura di emergenza.

Finito ciò si procede alla scrittura sul File (file.txt) della *situazione dell'ambiente intermedia*, e congiuntamente della stampa a video, per poter vedere da terminale cosa è successo, e in che modo si è evoluta la situazione.



Ultimo step di questa computazione (INTERMEDIA) è l'ultima fase di INTERAZIONE CON L'UTENTE, dove gli si dà possibilità di interagire attivamente nella modifica dell'ambiente; questo attraverso l'introduzione a piacere di un ostacolo FISSO con la regola *insertFixObstacle*

```
Prolog Console
SWI Prolog: Default Process (Prolog)
O W O O O O W T O O
O W O O O O W W W O
N W O O O O A O O O
O W O O O O O O O O
O W O O O O O O O O
O W O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O

Vuoi inserirne un nuovo OSTACOLO FISSO? (y/n)
|    y.

Inserisci la COORDINATA X del nuovo OSTACOLO FISSO: |
```

Viene esplicitamente chiesto se si desidera introdurre un nuovo Ostacolo fisso, optando per il sì il sistema, sull'esempio dell'Agente(NEO) e TARGET, chiede le coordinate in cui andare a posizionare l'Ostacolo FISSO, questo sarà subito inserito, andando a modificare istantaneamente l'ambiente.

Oltretutto viene effettuato un controllo sulla raggiungibilità dell'Agente (NEO) rispetto al target.

In caso di coordinate ERRATE, la computazione procede come se si avesse rifiutato la possibilità di modificare il sistema.

Se ne riporta un esempio, da Prolog Console:

```

Prolog Console
SWI Prolog: Default Process (Prolog)
O W O O O O W T O O
O W O O O O W W W O
N W O O O O A O O O
O W O O O O O O O O
O W O O O O O O O O
O W O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O

Vuoi inserirne un nuovo OSTACOLO FISSO? (y/n)
|   y.

Inserisci la COORDINATA X del nuovo OSTACOLO FISSO: |   6.
Inserisci la COORDINATA Y del nuovo OSTACOLO FISSO: |   3.
E' stato INSERITO un OSTACOLO FISSO NELL'AMBIENTE.

O W O O O O W T O O
O W O O O O W W W O
O W O O O O O O A O
N W O O O O O O O O
O W O O O O O O O O
O W W O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O

```

Con l'introduzione o meno del NUOVO OSTACOLO FISSO, termina l'esecuzione della regola *muovi_agenti*, e con essa si ottiene la soluzione parziale *i*-esima, dove $0 < i < n$; con *n* naturale, indicante il numero di step temporali al raggiungimento della SOLUZIONE FINALE.

Si procede ricorsivamente, con la regola *muovi_agenti*, FINCHE', la CONDIZIONE DI TERMINAZIONE, cioè che l'Agente (NEO) sia giunto al TARGET, NON VIENE VERIFICATA.

4.4 Output Files

4.4.1 file.txt

In questo file, restituito in OUTPUT come risultato del Sistema esperto, è inserita tutta la computazione, e le “SOLUZIONI/AMBIENTI” PARZIALI e quella FINALE.

Presenta una struttura tipica di una matrice Prolog Domain, con righe del tipo:

```
[ [0,W,0,0,0,0,W,T,0,0], [0,W,0,0,0,0,W,W,W,0], [N,W,0,0,0,0,A,0,0,0], [0,W,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0] ]
```

Questa visualizzazione, si ricorda che questa riportata è una sola riga, ossia UNA soluzione parziale.

Per poter ottenere una visualizzazione più “comoda”, ho scritto un visualizzatore in C++ di supporto appunto al sistema, che prende in input questo file e lo trasforma in una visualizzazione più consona agli scopi del sistema, vedendola come fosse un ambiente di Gioco.

4.4.2 decisionDir.txt

L’importanza della costruzione di questo file, risiede nel fatto che vengono riportati/stampati tutti le DIRECTION LIST dell’Agente NEO, indicando la LISTA INIZIALE, nella cui PRIMA POSIZIONE è sempre inserita la DIREZIONE SCELTA in primis, e immediatamente A CAPO, viene specificata in quale direzione l’Agente(NEO) si è mosso, indicando quindi possibili procedure di emergenza che sono state applicate.

Questo File non è possibile passarlo in Input al Visualizzatore.exe perché ha carattere solo informativo.

Se ne riporta due slot.

1. [[SE,80.53767779197437], [NE,9.462322208025618], [E,35.53767779197438], [N,54.46232220802562], [NO,99.46232220802563], [S,125.53767779197437], [O,144.46232220802563], [SO,170.53767779197437]]

SE

2. [[NE,0.0], [N,45.0], [E,45.0], [SE,90.0], [NO,90.0], [S,135.0], [O,135.0], [SO,180.0]]

E

Si può ben notare come nel primo Slot, la prima direzione è stata presa dall'Agente (NEO), mentre nel secondo è stato necessario prendere una direzione diversa dalla prima.

Questo perché, è possibile che siano stati trovati in tale direzione Ostacoli FISSI o talvolta MOBILI.

4.4.3 moveDetails.txt

Similmente al file.pl, questo file al suo interno conserva gli Ambienti/Matrici derivanti da ogni singolo spostamento di qualsiasi agente.

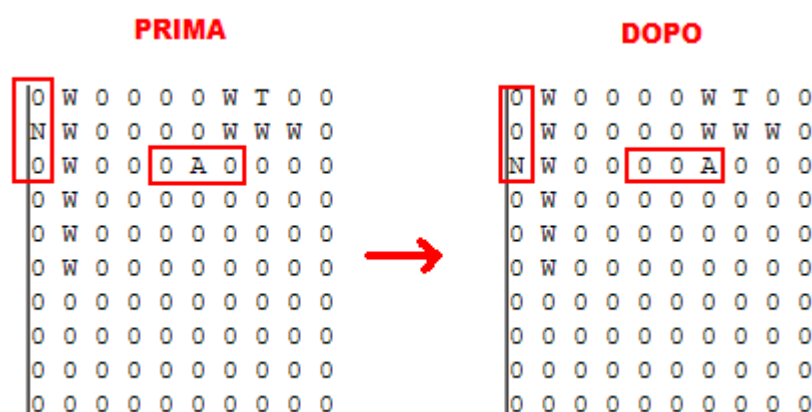
È una fotografia istantanea dell'ambiente, che permette anche di poter capire COME E QUALI OSTACOLI MOBILI si sono mossi e QUALI sono rimasti fermi (questo nel caso di eventuali collisioni).

Più righe di questo file CORRISPONDONO a UNA SOLA RIGA del file.pl

Questo File non è possibile passarlo in Input al Visualizzatore, mostrando uno spostamento di UNA SOLA ENTITA' per volta, COMPRESO L'AGENTE (NEO).

Per poter passare da una soluzione/Ambiente parziale ad un altro, vengono eseguite alcune consecutive righe del file moveDetails.txt

Si riportano righe di esempio di uno STEP, ad esempio quello riportato nel 4.4.1, e nello specifico passare da :



Prima viene spostato l'OSTACOLO FISSO :

`[[0,W,0,0,0,0,W,T,0,0],[N,W,0,0,0,0,W,W,W,0],[0,W,0,0,0,0,0,0,0,0],[0,W,0,0,0,0,0,0,0,0],[0,W,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]`

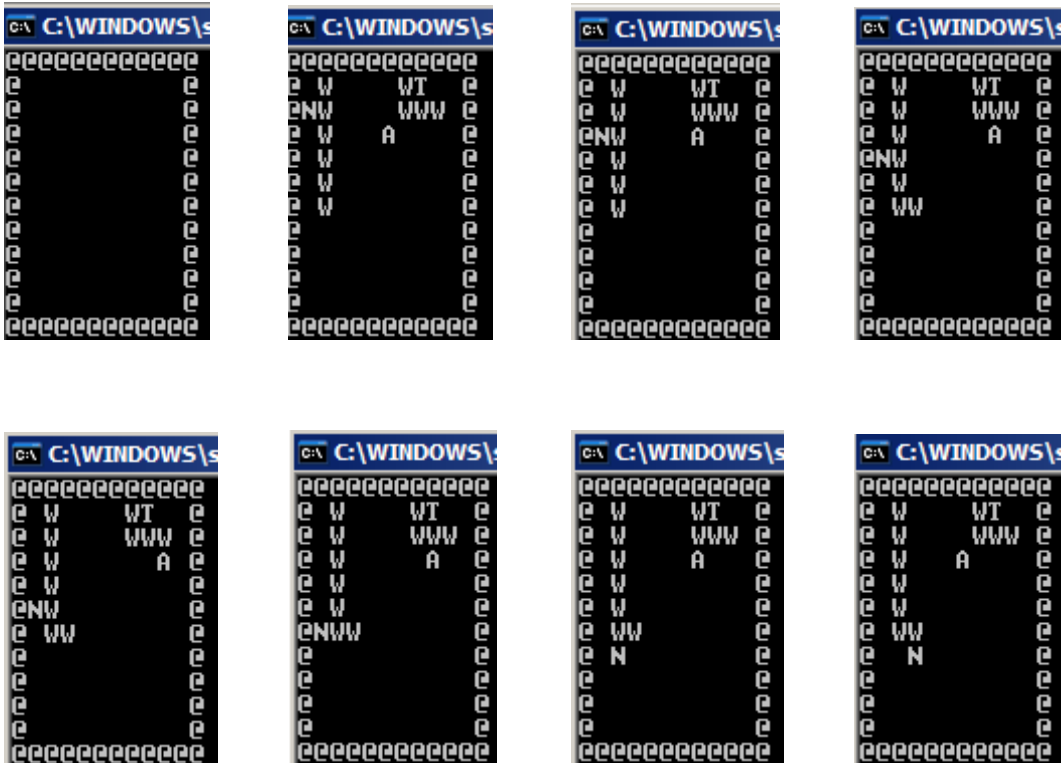
[[0,W,0,0,0,0,W,T,0,0], [N,W,0,0,0,0,W,W,W,0], [0,W,0,0,0,0,A,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0]]

e poi l'Agente(NEO):

[[0,W,0,0,0,0,W,T,0,0], [0,W,0,0,0,0,W,W,W,0], [N,W,0,0,0,0,A,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,W,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0]]

4.5 Esempio computazione su visualizzatore

Si riportano le schermate del visualizzatore, in merito a una computazione effettuata.



```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   e
e W       e
e W       e
e WWW     e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   e
e W   N   e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   e
e W   N   e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   e
e W   N   e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   AN  e
e W       e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   N e
e W       e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WT e
e W   WWW e
e W   A   e
e W       e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

```

C:\ C:\WINDOWS\s
#####
e W   WTN e
e W   WWW e
e W   A   e
e W       e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

GOAL:

```

C:\ C:\WINDOWS\s
#####
e W   WN  e
e W   WWW e
e W   A   e
e W       e
e W       e
e WW      e
e         e
e         e
e         e
e         e
#####

```

Capitolo 5.

Sviluppi Futuri

5.1 Finalità alternative del Progetto

Questo sistema oltre che per questo tipo di computazione, ben si presta anche alla risoluzione di labirinti.

Purché venga fornito in input la posizione di destinazione. Esso inoltre potrà dire se un eventuale labirinto può o meno essere risolto.

5.2 Work in progress

Diversi sono gli obbiettivi che conto di portare avanti per questo progetto, quali:

- rimozione della metodologia A^* , utilizzando esclusivamente quella di risoluzione dei triangoli.
- rimozione della verifica della raggiungibilità (can_reache) dell'obbiettivo TARGET
- miglior strutturazione degli OSTACOLI MOBILI
- upgrade delle velocità degli ostacoli MOBILI e Agente(NEO)
- facoltà di inseguimento degli OSTACOLI MOBILI verso l'Agente NEO.
- intelligenza degli OSTACOLI fissi per scopi collaborativi. Ad esempio chiudere l'unico corridoio di accesso al TARGET

Bibliografia

- *Intelligenza artificiale*. Un approccio moderno: 1 - Stuart J. Russell, Peter Norvig, F. Amigoni, S. Gaburri
- Intelligenza Artificiale - Nilsson N. J. APOGEO 2002
- N. Kasabov, Introduction: Hybrid intelligent adaptive systems. International Journal of Intelligent Systems, Vol.6, (1998) 453-454.
- Programmazione logica e *Prolog* - Luca Console, Evelina Lamma, Paola Mello
- Furlan F. & Lanzarone G.A.. *PROLOG*. Linguaggio e metodologia di programmazione logica
- P. E. Hart, N. J. Nilsson, B. Raphael: *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*, [SIGART](#) Newsletter, 37, pp. 28-29, 1972.
- N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980
- Slide del Corso di Intelligenza Artificiale, **Prof.ssa F.Esposito** - Cdl Magistrale Informatica – dipartimento di informatica – Università degli Studi di Bari
- Slide del Corso di Ingegneria e conoscenza dei Sistemi Esperti, **Prof.ssa F.Esposito** - Cdl Triennale Informatica – dipartimento di informatica – Università degli Studi di Bari
- Slide di Laboratorio di Prolog Intelligenza artificiale, **Prof. N.Di Mauro** - Cdl Magistrale Informatica – dipartimento di informatica – Università degli Studi di Bari
- Swi-Prolog Manuale: <http://www.swi-prolog.org/man/clpfd.html>