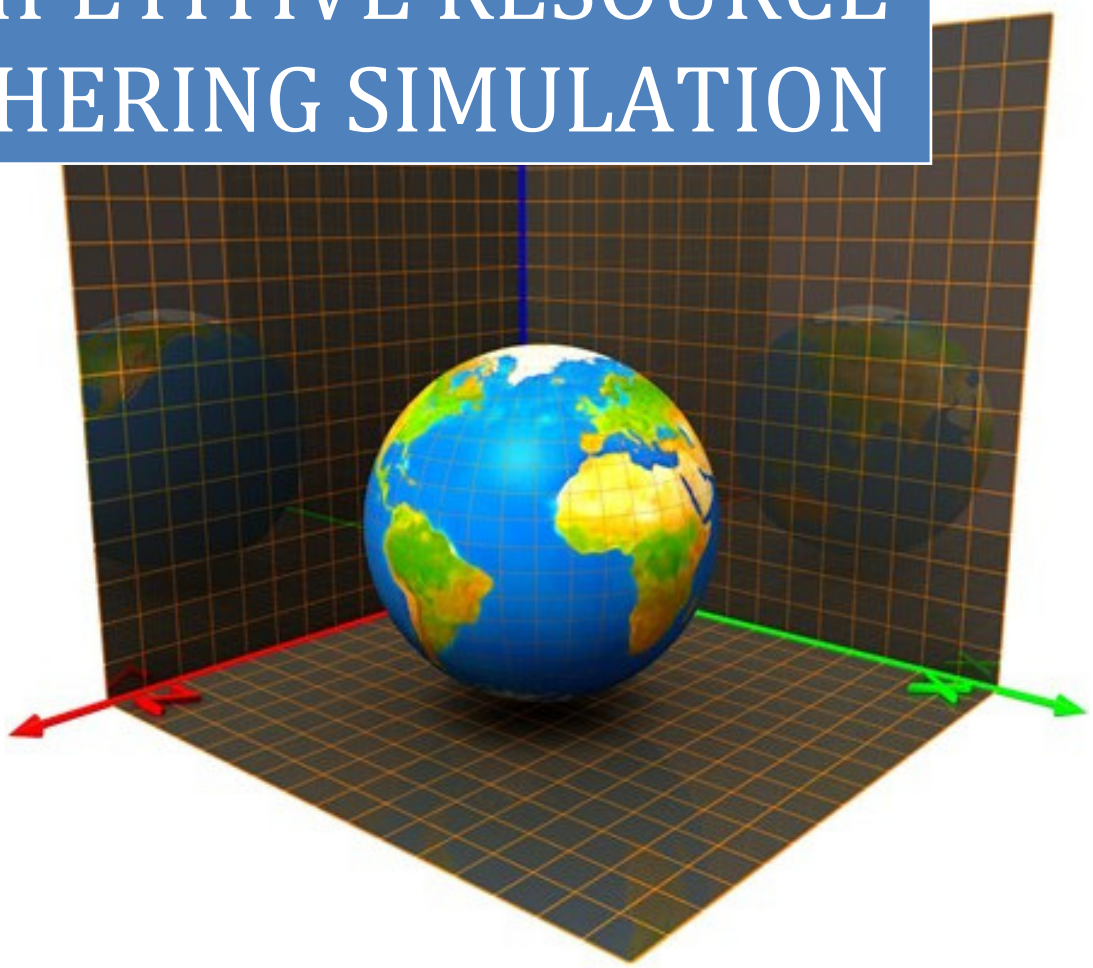
DALI PROJECT

2013/2014

COMPETITIVE RESOURCE GATHERING SIMULATION



Manuel Dell'Elce

Email: mh025r0x@gmail.com

Matricola: 230340

Università degli Studi dell'Aquila

18/07/2014

Indice

Introduzione

Il progetto presentato di seguito consiste nella **simulazione** di un ambiente composto da agenti e da risorse in esso distribuite, in cui gli agenti **competono** e combattono tra di loro al fine di recuperare tali **risorse**.

Il MAS è stato modificato in modo da consentirne la piena fruizione in sistemi **Linux** based, risolvendo anche alcuni dei problemi comuni che in esso si presentano.

L'architettura è stata estesa consentendo la definizione di **tipi** di agenti, i quali possono poi essere **istanziati** in base alle preferenze dello sviluppatore; inoltre, l'architettura è stata resa **modulare**, creeando una serie di moduli condivisi tra gli agenti e da essi importati, ad esempio contenenti quelle funzioni di utilità comuni.

Lo scambio di **messaggi** intra-agente, senza l'utilizzo dell'agente user, è stato raggiunto scrivendo direttamente sullo stream tcp del server.

Come vedremo, il mondo simulato è inoltre specificabile tramite file esterno, utilizzando un formato dichiarativo human-readable, il quale è poi inoltre riutilizzato nella parte presentazione degli agenti stessi per visualizzare la parte di mondo da essi osservata.

Traduzione del MAS per ambienti Linux

I file ***startmas.bat***, ***conf/makeconf.bat*** e ***conf/startagent.bat*** sono stati analizzati dettagliatamente e tradotti nelle loro controparti .sh, in modo da rendere il MAS completamente funzionale su Linux.

Per l'avvio di terminali separati per gli agenti, è stato utilizzato ***xterm***, terminale presente nella maggior parte delle distribuzioni linux, si prega di installarlo in caso in cui si ricevano errori sulla sua non esistenza, ad esempio utilizzando il comando *"apt-get install xterm"* in distribuzioni Debian based.

È inoltre necessario specificare il path di installazione della propria versione di **SICStus Prolog** nella variabile sicstus_home presente all'inizio del file, esempio:

```
sicstus_home=/usr/local/sicstus4.2.3
```

Una volta avviati tutti i processi necessari tramite lo startmas, questo rimane in attesa della pressione di un tasto, premuto il quale vengono terminati tutti i processi xterm e sicstus esistenti, chiudendo così il MAS.

Attenzione: se sono presenti ulteriori processi sicstus o xterm non relativi al MAS, questi verranno comunque terminati alla chiusura del MAS.

Lo startmas è stato inoltre modificato per consentire la suddivisione degli agenti in tipi ed istanze, la quale deve essere rispettata per il corretto funzionamento del MAS, consultare la relativa sezione per maggiori informazioni.

Notare inoltre che i file contenenti la configurazione dell'agente, creati dal makeconf e precedentemente salvati nella root della directory conf, sono ora memorizzati nell'apposita sottodirectory **conf/mas**, in modo da poter essere cancellati ad ogni riavvio.

Abilitare riconnessione immediata

Una volta terminato il MAS, il kernel linux impiega fino ad un minuto per rendere di nuovo disponibili le porte precedentemente chiuse, ciò renderebbe particolarmente scomodo l'utilizzo del MAS, specialmente in fase di sviluppo in cui è necessario chiuderlo e riavviarlo spesso.

A tal proposito, è stato creato un apposito file "fix address already in use error.sh" che permette di disabilitare tale comportamente **fino al prossimo riavvio**, abilitando il riciclo rapido delle connessioni tcp, impostando ad 1 il parametro "net.ipv4.tcp_tw_recycle" tramite il comando Linux/Unix "sysctl".

Estensioni al MAS

La struttura di base del MAS è stata modificata in modo da cosentire, tra l'altro, la suddivisione degli agenti in moduli, tipi ed istanze.

Tipi e Istanze

Aprendo la cartella mas nella root del progetto, si vedrà che non è più composta semplicemente dai file degli agenti, bensì è suddivisa in due sottocartelle: types and instances.

Spesso capita infatti di avere una serie di agenti con lo stesso comportamento generale. È il caso del progetto in analisi, in cui il tipo di agente "robot" è poi istanziato in un certo numero di agenti con gli stessi schemi di comportamento e stati iniziali, i quali vengono però modificati e prendono strade diverse in base alle esperienze individuali.

Nella cartella **types** vengono specificati i file degli agenti così come fatto normalmente con la vecchia struttura, questi sono però istanziabili nella cartella **instances**, creando un file txt che ha come nome quello dell'istanza e come contenuto il nome del file relativo al proprio tipo, comprensivo di estensione.

Ad esempio, nel progetto in analisi, nella cartella types sono presenti i due tipi:

- robot.txt
- world.txt

Mentre nella cartella instances abbiamo le istanze:

- robot1.txt con contenuto "robot.txt"
- robot2.txt con contenuto "robot.txt"
- robot3.txt con contenuto "robot.txt"
- world.txt con contenuto "world.txt"

L'agente world è un singleton: ne è presente una sola istanza.

Per utilizzare tale nuova struttura, lo startmas.sh è stato modificato ulteriormente, in modo da creare i file definitivi degli agenti.

A tal proposito, vengono passati in rassegna tutti i file presenti nella cartella mas/instances, recuperando per ciascuno di essi il relativo file di tipo in mas/types, tale file viene poi copiato rinominandolo con il nome dell'istanza nella nuova cartella build, che conterrà quindi gli agenti veri e propri utilizzati per la compilazione e l'avvio.

Moduli

Altra situazione che capita spesso nello sviluppo, è quella di avere del codice comune a più parti della propria applicazione, ad esempio per lo scambio di messaggi, configurazioni, etc.

Relativamente a ciò, è stata creata una nuova cartella **mas_modules** in cui è possibile specificare i moduli poi importati all'interno degli agenti con la direttiva built-in use_module, es. "import use_module('mas_modules/utils.txt')".

Inviare messaggi direttamente sul server TCP

Requisito del progetto è quello di inviare messaggi da agente ad agente senza dover inserire manualmente i messaggi tramite l'agente standard user, a tal proposito, considerata l'impossibilità di scambiare messaggi tramite l'azione message, la quale risulta ad oggi non funzionante, è stato analizzato il funzionamento dell'agente user stesso, in modo da individuare il metodo utilizzato per l'invio di messaggi.

Il metodo consiste nello scrivere direttamente sullo stream TCP del Linda Server, inviando il messaggio "message" con il predicato out.

Ne è stata così estratta un'azione "**message**", **specificata nel modulo utils.txt**, che mima il funzionamento dell'azione originale:

```
/* Send message by writing to the server. */
message(To, Message, From) :> getServer(I), out(message(I,To,I,From,italian,[],Message)).
```

Dove l'azione **getServer/1** è definita come segue:

```
/* Evaluates the server name I, used by the message utility. */
:-dynamic server/1.
getServer(I) :> server(I) ; open('../interpreter/server.txt',read,Stream,[]), read(Stream,I), close(Stream),
assert(server(I)).
```

Il file **interpreter/server.txt** è automaticamente generato dall'interprete ad ogni avvio e contiene l'indirizzo host:porta del server, ad es. *'manuel-hp':3010*.

La getServer legge il contenuto del file e lo memorizza tramite il predicato dinamico "server" appositamente creato, in modo da restituirlo subito nelle richieste successive, tale indirizzo del server è quindi utilizzato dall'azione message per individuare il server su cui scrivere il messaggio.

Aspetti generali degli agenti

È importante analizzare di seguito quegli elementi comuni a tutti gli agenti, quali metodologie di avvio, scambio e ricezione di messaggi, nonchè di gestione della loro identità .

Configurazione

All'inizio del file di ogni agente, è presente la sua **configurazione**, ovvero la dichiarazione dei predicati dinamici e i fatti dell'agente.

```
1 /**** Configuration *****/
2 worldFilename('conf/world.txt'). /* Note: the world in the file must be a square/rectangle! */
3 :- dynamic worldWidth/1.
4 :- dynamic worldHeight/1.
5 :- dynamic worldCell/3. /* x, y and type */
6 :- dynamic worldAgent/2. /* identity and properties (position and health) of agents that joined the world */
7 line_of_sight(4).
```

Configurazione dell'agente world

Main

Dopo la configurazione troviamo il **main**: l'entry point dell'applicazione, in cui vengono caricati tutti i moduli e le librerie necessarie, eseguite le inizializzazioni e le azioni iniziali.

L'ambiente DALI esegue l'azione "go" all'avvio dell'agente ed era utilizzata inizialmente per l'esecuzione del main, sfortunatamente però, è stato riscontrato che eseguirla impedisce la gestione degli eventi esterni, i quali vengono rilevati solo come "appresi", ma senza chiamare i relativi handler.

Per porre rimedio a tale inconveniente, è stata sfruttata la **reattività** offerta da DALI, combinata ai predicati dinamici:

```
 9 /***** Main *****/
10
11 :- dynamic started/1.
12 starting(this) :- not(started(this)).
13 startingI(X) :> main, assert(started(this)).
```

Il predicato started(this), inizialmente falso, rende vero starting(this), il quale causa tramite reattività l'esecuzione del l'azione main, terminata la quale impostando a vero started(this) si impedisce di chiamare l'azione una seconda volta al prossimo ciclo di controllo.
Dato che il ciclo di controllo viene avviato solo dopo che l'agente è pronto, ciò consente di simulare il go, ma senza lo svantaggio del non poter gestire gli eventi esterni.

Scambio di messaggi

Per quanto riguarda lo **scambio di messaggi**, notiamo che è necessario specificare non solo il nome del ricevente, ma anche quello del mittente, informazione che è tra l'altro utile per discriminare come gestire il messaggio stesso a livello di agente.
Tale informazione, non può però esser più specificata in modo statico nel codice direttamente quando si invia un messaggio, poichè come visto precedentemente, gli agenti che andiamo a definire sono in realtà tipi, poi istanziati da agenti con nomi specifici.
Fortunatamente, la configurazione di ogni agente è accessibile in quanto esportata dall'interprete DALI

stesso nel predicato “agente”, ciò ci consente quindi in modo dinamico, all’avvio dell’agente, di recuperare e memorizzare il nome dell’agente.
A tal proposito, è stata creata un’azione retrieveIdentity nel modulo utils, definita come segue:

```
3 :- dynamic identity/1. /* name of the agent, used to identfy it in the message exchange */
4 retrieveIdentity :> agente(N,Tee,S,F), assert(identity(N)).
```

L’azione estrae il nome N e lo memorizza con identity(N).

Tale azione è quindi chiamata da tutti gli agenti nel proprio main:

```
main :> use_module(library(random)),
        use_module('mas_modules/utils.txt'),
        use_module('mas_modules/world_objects.txt'),
        retrieveIdentity, /* action in utils */
        nl, write('Started'), nl,
        loadWorldFromFile.
```

Main dell’agente world: notare la call a retrieveIdentity

Per lo **scambio di messaggi**, gli agenti utilizzano una serie di **helper** definiti nel modulo utils, che offrono uno **shortcut** per il loro invio, ma che allo stesso tempo permettono l’utilizzo di un **formato standard** che offre una serie di vantaggi.

Le azioni di helper sono “**send(To, MessageName)**” e “**send(To, MessageName, ArgsList)**”, dove To è il nome dell’agente ricevente, MessageName è un atomo con il nome del messaggio, es. ‘join’ ed ArgsList è una lista opzionale contenente gli argomenti del messaggio, espressi come atomi, es. [now, please].

L’helper al suo interno estrae l’identità dell’agente e gli eventuali argomenti, inviando poi il messaggio tramite l’azione “message” vista precedentemente, utilizzando il formato:

send_message(received(From, MessageName, Arg1, … , ArgN), From)

E causando in fase di ricezione l’evento esterno (in fase di invio viene creato dinamicamente un funtore received che contiene tra l’altro gli elementi della lista come parametri separati):

receivedE(From, MessageName, Arg1, Arg2, ... , ArgN)

Ciò ha i seguenti **vantaggi**:

- Utilizzare un unico tipo di evento esterno permette di poter definire handler generali che riguardano tipi diversi di messaggi,
- Viene passato sempre automaticamente il mittente in From, permettendo una gestione più fine del messaggio.
- Dato che all’interno del tipo di messaggio (received), è possibile inviare solo atomi, e non ad esempio predicati complessi quali position(X,Y), la lista di argomenti permette di risolvere agevolmente tale problema, inviando ad. es. il messaggio come send(agent, position, [X,Y]) e ricevendolo come receivedE(agent, position, X, Y).

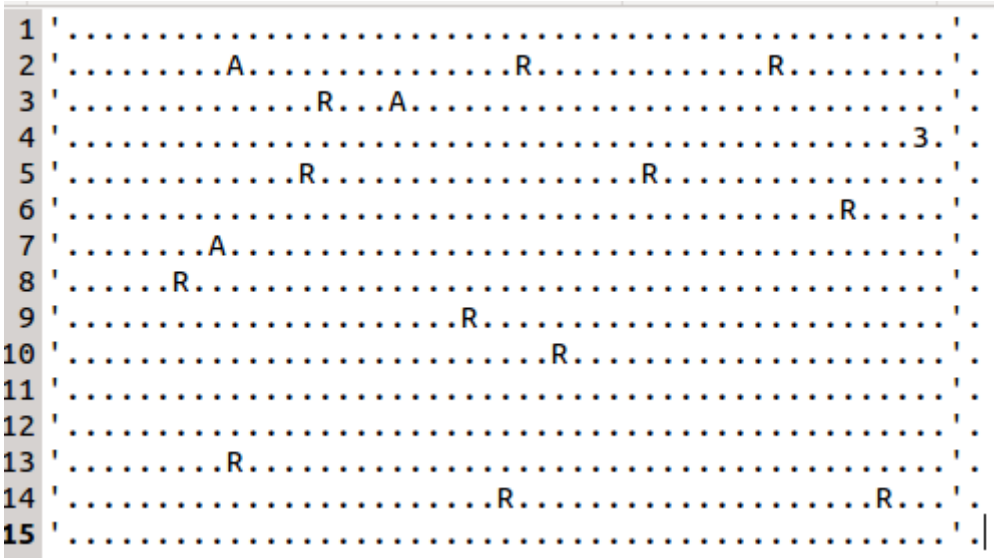
Agente World

Nel progetto in esame non è previsto l'utilizzo di un software di simulazione separato, pertanto, il simulatore è stato sviluppato tramite un particolare tipo di agente, il cosiddetto World.

La simulazione consiste in un mondo bidimensionale a griglia dalla forma rettangolare, ogni cella del mondo è occupata da uno ed un solo **oggetto**,il quale può comunque cambiare nel corso della simulazione e che ne definisce il suo tipo.

Definizione del mondo

Il mondo al suo stato iniziale è definito nel file **conf/world.txt**:



Come si può vedere dall'immagine, ogni riga è una stringa della stessa lunghezza, i cui caratteri definiscono il valore delle celle nella rispettiva "riga" del mondo.

La semantica dei caratteri è definita nel **modulo world_objects.txt**:

```
/* Specifies the possible cells of the world and their meanings in term of objects. */

worldObject('A', agent).
worldObject('R', resource).
worldObject('.', blank).

```

Con blank B si intende una cella di sfondo/terreno in cui è possibile muoversi liberamente, mentre le resource R sono risorse che gli agenti "agent" A sono interessati a recuperare.

Un nuovo mondo è quindi definibile semplicemente creando un nuovo file world.txt o modiicando quello esistente. Notare inoltre che è anche possibile cambiare il path del file del mondo modificando il parametro del predicato worldFilename/1 nella configurazione dell'agente.

Caricamento del mondo

Il primo vero compito dall'agente world consiste nella lettura e nel parsing del file sopracitato, operazione effettuata tramite l'azione **loadWorldFromFile** .
Per ogni carattere X con valore T di ogni linea Y del file, viene memorizzata la relativa cella del mondo worldCell(X,Y,T) ; inoltre l'altezza e la larghezza del mondo vengono calcolati come, rispettivamente, il numero di linee (meno quella di End Of File), e il numero di caratteri in una linea del file.

Registrazione degli agenti

Il mondo è ora pronto a ricevere visitatori: gli agenti invieranno un messaggio "**join**" per richiedere di poter essere registrati nel mondo.
A tale richiesta, si reagisce cercando una cella del mondo contenente un oggetto "agente" (quì inizialmente funzionante come "segnaposto", o spawn point) e non ancora occupata, registrando il nuovo utente del mondo nella posizione trovata.
Infine, il processo è completato impostando i punti salute dell'agente, utilizzati in fase di competizione, e inviando un messaggio di risposta all'agente stesso, passandogli tra l'altro anche informazioni relative al mondo, quali larghezza ed altezza, da lui utilizzate per la sua esplorazione:

```
receivedE(From, join) :>
    nl, write('Received join request from '), write(From), nl,

    /* find available spawn point for the new agent */
    worldObject(AgentCellType, agent), worldCell(X, Y, AgentCellType), \+ worldAgent(AnotherAgent, position(X, Y)),
    assert(worldAgent(From, position(X, Y))),
    /* set health */
    assert(worldAgent(From, health(100))),

    write('Sending acceptance...'), nl,
    worldWidth(WorldWidth), worldHeight(WorldHeight),
    send(From, joined, [X, Y, WorldWidth, WorldHeight]).
```

Invio delle informazioni ambientali agli agenti

Come si vedrà nella sezione apposita, una volta che un agente ha effettuato l'accesso, inizierà a guardarsi intorno, compiendo il **sensing** dell'ambiente relativo alla sua posizione corrente, ciò viene effettuato inviando un messaggio "sense" all'agente world.

Fare il sensing corrisponde ad ottenere i valori delle celle riguardanti una certa porzione di mondo, nello specifico, quelle che si estendono per una certa lunghezza nei quattro punti cardinali con centro la posizione dell'agente; tale lunghezza è il "line_of_sight" , valore configurabile, con valore di default pari a 4.

Pertanto il world reagisce al messaggio di sense di un agente recuperandone la posizione, informazione inoltre ad esso inviata e che gli consente di orientarsi, calcolando poi il rettangolo con centro tale posizione e span pari al line_of_sight ed inviando quindi un messaggio "cell"con X,Y e Tipo di cella per ogni cella del rettangolo individuato.

Notare che ciò comporta l'invio di un certo numero di messaggi per completare il sense, il che rallenta l'operazione rispetto all'invio di un unico messaggio contenente tutte le informazioni; quest'ultima strada è stata tentata, ma senza successo, creando una lista di argomenti con tutti i dati richiesti, in quanto sembra esserci un limite alla lunghezza dei messaggi scambiabili, superato il quale questi non vengono più spediti.

Nota: alla ricezione del messaggio di sense, prima di gestirlo, l'agente chiede all'utente di premere invio per proseguire, ciò è stato aggiunto per facilitare l'analisi di ciò che sta succedendo, permettendo all'utente di avanzare quando desidera.

Modificare il mondo

Il mondo mette a disposizione una serie di azioni che permettono agli agenti di modificarlo a loro piacimento, il tutto ovviamente rispettando le regole da esso dettate, anche se nella versione corrente non sono state implementate severe politiche di controllo e ci si affida al buon senso degli agenti. Nello specifico, le versioni possibili sono: muovere un agente, consumare un oggetto o attaccare un altro agente.

Muovere un agente

Azione accessibile da un agente tramite l'invio di un messaggio "move" contenente posizione di partenza (FromX, FromY) e posizione di arrivo (ToX, ToY). Viene effettuata scambiando i valori delle celle di partenza e di arrivo, ma sotto il vincolo che quest'ultima sia effettivamente una cella blank, ovvero una in cui è possibile spostarsi liberamente.
Effettuato tale spostamento dell'oggetto "agent" con l'oggetto "blank", viene aggiornata la posizione dell'agente inteso come utente del mondo.

```
receivedE(From, move, FromX, FromY, ToX, ToY) :>
    nl, write('Received move request from '), write(From), nl,
    moveObject(FromX, FromY, ToX, ToY),
    /* update agent position */
    retract(worldAgent(From, position(_, _))), assert(worldAgent(From, position(ToX, ToY))).
moveObject(FromX, FromY, ToX, ToY) :>
    worldCell(FromX, FromY, FromCellType),
    worldCell(ToX, ToY, ToCellType),
    worldObject(ToCellType, ToObject),
    ToObject = blank,
    /* swap from and to */
    write('Moving object...'), nl,
    retract(worldCell(FromX, FromY, FromCellType)), assert(worldCell(FromX, FromY, ToCellType)),
    retract(worldCell(ToX, ToY, ToCellType)), assert(worldCell(ToX, ToY, FromCellType)).
```

Azione move

Consumare un oggetto

Accessibile tramite il messaggio "consume", rimuove un oggetto dal mondo, utilizzato ad esempio dagli agenti per "consumare" le risorse, ovvero recuperarle.

Il messaggio richiede inoltre gli argomenti X e Y relativi alla posizione dell'oggetto da rimuovere, e viene effettuata modificando il tipo della cella in tale posizione con un blank.

```
receivedE(From, consume, X, Y) :>
    nl, write('Received consume request from '), write(From), nl,
    consumeObject(X, Y).
consumeObject(X, Y) :>
    worldObject(BlankCellType, blank),
    write('Consuming object...'), nl,
    retract(worldCell(X, Y, CellType)), assert(worldCell(X, Y, BlankCellType)).
```

Azione consume

Attaccare un agente

L'ultima azione effettuabile nel mondo, nonchè la più complessa, consiste nell'attaccare un altro agente. È accessibile tramite il messaggio "attack" e richiede i parametri X e Y della cella in cui è presente l'agente da attaccare.

Viene effettuata recuperando l'agente alla posizione data e, se presente, diminuendo i suoi punti salute di una quantità casuale compresa tra 0 e 100.

Nel caso in cui i punti salute raggiungano o vadano sotto lo zero in seguito all'attacco, l'agente è considerato morto e viene rimosso dal mondo, sostituendo il valore della sua cella con una blank. Inoltre viene ad esso mandato un messaggio "dispose" per consentirgli di terminare la sua esecuzione.

```
receivedE(From, attack, X, Y) :>
    worldAgent(Agent, position(X, Y)),
    nl, write('Received attack request from '), write(From), write(' versus '), write(Agent), nl,
    /* decrease agent health */
    random(0, 100, Damage),
    worldAgent(Agent, health(Health)), NewHealth is Health - Damage,
    write('Inflicted damage = '), write(Damage), nl,
    retract(worldAgent(Agent, health(_))), assert(worldAgent(Agent, health(NewHealth))),
    (NewHealth =< 0, /* remove agent */
        write(Agent), write(' died!'), nl,
        worldObject(BlankCellType, blank), retract(worldCell(X, Y, _)), assert(worldCell(X, Y, BlankCellType)),
        retractall(worldAgent(Agent, _)), send(Agent, 'dispose')
    ; /* else */
        write('Remaining opponent health = '), write(NewHealth), nl).
```

Agente Robot

L'agente robot è un utente del mondo, si muove in esso al fine di recuperare il maggior numero possibile di risorse, competendo al tempo stesso con gli altri robot esistenti ed attaccandoli al fine di distruggerli.

Particolarità importante è il concetto di **tempo**, il comportamento dell'agente è infatti diviso in step, ciascuno dei quali identificato con un istante temporale incrementato di volta in volta ed utilizzato per annotare le osservazioni compiute.

L'agente infatti non solo osserva la situazione corrente del mondo, ma si ricorda anche di quelle precedenti e le usa a suo vantaggio in caso di necessità, ad esempio, se nel suo processo di recupero di risorse finisce per trovarsi in un'area già completamente ripulita, potrà sfruttare la sua memoria per muoversi verso una zona lontana della mappa in cui però ricorda di aver visto delle risorse.

Entrare nel mondo

La prima vera operazione dell'agente robot consiste nel mandare un messaggio "join" all'agente world, aspettandosi in risposta un messaggio "joined" con la sua posizione e con le informazioni del mondo, ovvero larghezza ed altezza.

```
/* Notify the world it want to join, obtaining in response info about the world, such as its width and height. */
joinWorld :-
    nl, write('Joining world...'), nl,
    send(world, join).
receivedE(world, joined, SpawnX, SpawnY, WorldWidth, WorldHeight) :-
    write('Joined!'), nl,
    write('Spawned at ('), write(SpawnX), write(', '), write(SpawnY), write(')'), nl,
    write('World has width = '), write(WorldWidth), write(' and height = '), write(WorldHeight), nl,
    assert(worldWidth(WorldWidth)), assert(worldHeight(WorldHeight)),
    newStep. /* Begin living! */
```

```
Started

Joining world...
Joined!
Spawned at (9, 1)
World has width = 49 and height = 15
```

Nuovo step temporale

Il robot è ora pronto a vivere nel mondo, incrementa il contatore con l'istante temporale corrente, nel main impostato a -1, e inizia a guardarsi intorno effettuando il sense del mondo.

Tale operazione è svolta con l'azione **newStep** e viene richiamata ogni volta dopo aver eseguito un azione, dando luogo così al loop di comportamento "sensing -> action -> sensing".

```
newStep :-
    time(CurrentTime), retractall(time(X)), NewTime is CurrentTime+1, assert(time(NewTime)),
    identity(Agent), nl, write(Agent), write(': begin new step, time '), write(NewTime), nl,
    nl, write('Sensing environment...'), nl,
    send(world, sense).
```

```
robot1: begin new step, time 0

Sensing environment...
```

Osservare il mondo

Dopo aver inviato il messaggio di sense, come visto nell'agente world, il robot si aspetta in risposta la sua posizione ed una serie di messaggi riguardanti il valore delle celle da lui visibili.

Formando queste celle un rettangolo, l'agente le stampa a schermo riga per riga, in modo da permettere all'utente di rendersi conto della posizione corrente dell'agente e di ciò che ha intorno.

Ogni informazione sulle cella ricevute e sulla propria posizione viene memorizzata annotandola con l'istante temporale corrente.

Al termine dell'operazione di sense, l'agente world invierà un messaggio senseEnd, al cui il robot reagirà eseguendo la fase successiva del suo comportamento, ovvero agendo, ed in seguito iniziando un nuovo step temporale.

```
/***** Sense world *****/

receivedE(world, position, X, Y) :>
    nl, write('My position is ('), write(X), write(','), write(Y), write(')'), nl,
    time(Time), assert(position(X, Y, Time)).

receivedE(world, senseNewY) :> nl. /* Break line to print the next row aligned with the previous one */

receivedE(world, cell, X, Y, Type) :>
    write(Type),
    time(Time), assert(cell(X, Y, Type, Time)).

receivedE(world, senseEnd) :>
    nl, nl, /* rows printing finished */
    act, /* Do something with the world */
    newStep. /* start next iteration */
```

```
Sensing environment...

My position is (9,1)

.........
....A....
.........
.........
........R
.........
```

Come si può vedere nell'immagine, la porzione di mondo visibile dal robot comprende una risorsa.

Funzioni d'utilità

Prima di trattare le azioni vere e proprie eseguite dall'agente, è utile passare in rassegna quei metodi d'utilità da esso utilizzati, in modo da avere una visione più chiara di come il mondo viene analizzato e manipolato.

Recall

Permette di recuperare le informazioni che si hanno su una cella del mondo, basandosi sull'osservazione più recente fatta su di essa. Può inoltre essere utilizzata anche per ricordare celle in cui sono presenti nemici e posizioni del mondo di cui non si ha alcuna informazione.

```
/* Evaluates information about the cell at the latest time it has been seen. */
recallCell(X, Y, enemy) :- /* to recall cell where there is an enemy, i.e. an agent different from us. */
    time(TimeNow), position(MyX, MyY, TimeNow),
    worldObject(AgentCellType, agent),
    worldWidth(WorldWidth), worldHeight(WorldHeight), XUpper is WorldWidth-1, YUpper is WorldHeight-1,
    between(0, XUpper, X), between(0, YUpper, Y),
    recallCell(X, Y, AgentCellType), (X \= MyX ; Y \= MyY).
recallCell(X, Y, void) :- /* to recall unknown cell of the world */
    worldWidth(WorldWidth), worldHeight(WorldHeight), XUpper is WorldWidth-1, YUpper is WorldHeight-1,
    between(0, XUpper, X), between(0, YUpper, Y),
    voidCell(X, Y).
recallCell(X, Y, Type) :-
    cell(X, Y, Type, Time), \+ (cell(X, Y, _, OtherTime), OtherTime > Time).

/* a cell is void if never seen. */
voidCell(X, Y) :- \+ cell(X, Y, _, _).
```

Distance

Usata per valutare la distanza tra una posizione del mondo ed un altra, in termine di spostamenti orizzontali, verticali o diagonali nella griglia del mondo richiesti per raggiungere la posizione di arrivo.

Tale tipo di distanza corrisponde alla Chebyshev distance [1], anche detta chessboard distance, ed è pari al massimo delle differenze delle due coordinate.

```
/* Evaluates chessboard distance between two points. */
distance(FromX, FromY, ToX, ToY, Distance) :-
    Distance is max(abs(FromX-ToX), abs(FromY-ToY)).
```

Closest

Valuta la posizione della cella più vicina di un dato tipo di cui l'agente si ricorda. Utilizza la recall e la distance:

```
/* Evaluates information about the closest cell. */
closest(X, Y, CellType) :-
    time(TimeNow), position(MyX, MyY, TimeNow),
    recallCell(X, Y, CellType), distance(X, Y, MyX, MyY, Distance),
    \+ (recallCell(OtherX, OtherY, CellType), distance(OtherX, OtherY, MyX, MyY, OtherDistance), OtherDistance < Distance).
```

Move towards

Usata per spostarsi nel mondo, data la posizione d'arrivo, calcola il tipo di spostamento (di una cella) richiesto: orizzontale, verticale o diagonale e la sua direzione, inviando poi il relativo messaggio al mondo.

```
moveTowards(X, Y) :>
    time(TimeNow), position(MyX, MyY, TimeNow),
    XDistance is X-MyX, YDistance is Y-MyY,
    (XDistance > 0, XDelta is 1 ; XDistance = 0, XDelta is 0 ; XDistance < 0, XDelta is -1),
    (YDistance > 0, YDelta is 1 ; YDistance = 0, YDelta is 0 ; YDistance < 0, YDelta is -1),
    MyNewX is MyX+XDelta, MyNewY is MyY+YDelta,
    send(world, 'move', [MyX, MyY, MyNewX, MyNewY]).
```

[1] http://en.wikipedia.org/wiki/Chebyshev_distance

Azioni

L'utente può svolgere una serie di azioni nel mondo, queste sono inoltre prioritizzate, nel senso che l'agente cercherà sempre di compiere quelle a priorità più alta prima di tentarne una con priorità minore. Tali azioni sono, ordinate per priorità:

1. Attaccare un nemico
2. Muoversi verso un nemico
3. Consumare una risorsa
4. Muoversi verso una risorsa
5. Muoversi verso una cella sconosciuta
6. Attendere

Attaccare un nemico

Se l'agente è adiacente ad un nemico, ovvero si trova a distanza 1 da esso, cercherà di attaccarlo inviando al mondo il relativo messaggio attack con le coordinate dell'agente nemico.

Notare che l'attacco potrebbe fallire se nel frattempo l'agente si è spostato verso una cella più distante.

```
/* Attack adjacent enemy */
act :>
    time(TimeNow), position(MyX, MyY, TimeNow),
    closest(EnemyX, EnemyY, enemy),
    distance(EnemyX, EnemyY, MyX, MyY, Distance),
    Distance is 1, /* is adjacent */
    write('Attacking adjacent enemy at ('), write(EnemyX), write(','), write(EnemyY), write(')'), nl,
    send(world, attack, [EnemyX, EnemyY]).
```

```
robot1: begin new step, time 3

Sensing environment...

My position is (8,4)

.. ......
.........
.........
.........
....A....
.....A...
.........
..R......
.........

Attacking adjacent enemy at (9,5)
```

World:

```
Received attack request from robot1 versus robot3
Inflicted damage = 1
Remaining opponent health = 99
```

Muoversi verso un nemico

Se l'agente si ricorda di aver visto dei nemici in una data posizione, deciderà di muoversi verso quello più vicino al fine di poterlo attaccare negli step successivi tramite l'azione precedente.

```
/* Move towards closest enemy to attack it */
act :>
    closest(EnemyX, EnemyY, enemy),
    write('Moving towards closest enemy at ('), write(EnemyX), write(','), write(EnemyY), write(')'), nl,
    moveTowards(EnemyX, EnemyY).
```

Consumare una risorsa

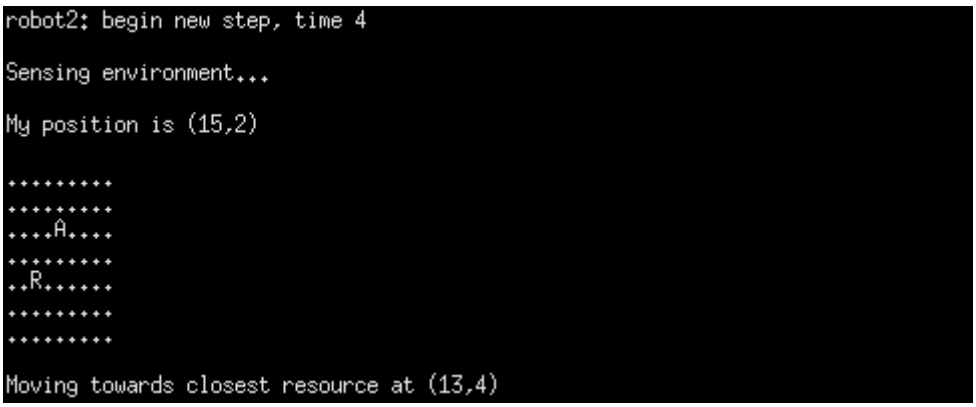
Simile all'azione dell'attacco, applicata se l'utente è adiacente ad una risorsa, invia al mondo il relativo messaggio consume.

```
/* Consume adjacent resource */
act :>
    worldObject(ResourceCellType, resource),
    closest(ResourceX, ResourceY, ResourceCellType),
    time(TimeNow), position(MyX, MyY, TimeNow),
    distance(ResourceX, ResourceY, MyX, MyY, Distance),
    Distance is 1, /* is adjacent */
    write('Consuming adjacent resource at ('), write(ResourceX), write(','), write(ResourceY), write(')'), nl,
    send(world, consume, [ResourceX, ResourceY]).
```

Al passo successivo, la risorsa non è più visibile in quanto consumata:

Muoversi verso una risorsa

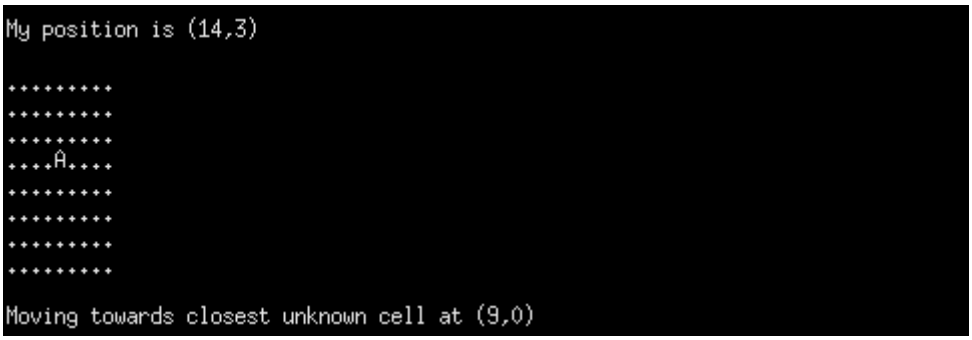
Se non ci sono risorse adiacenti, ma l'agente ne vede una più lontana o comunque si ricorda di una cella in cui ne aveva vista una, utilizzerà la moveTowards per spostarsi verso di essa.



Muoversi verso una cella sconosciuta

Se l'agente non ha di meglio da fare, esplorerà il mondo muovendosi verso quelle celle di cui non ha ancora alcuna informazione, al fine di trovare nuove risorse e nemici.

```
/* Explore closest unknown cell of the world. */
act :>
    closest(X, Y, void),
    write('Moving towards closest unknown cell at ('), write(X), write(','), write(Y), write(')'), nl,
    moveTowards(X, Y).
```



Attendere

Se le azioni precedenti non sono applicabili, ovvero l'agente non vede nè si ricorda di nemici o risorse vicine o lontane ed ha esplorato tutto il mondo, semplicemente attenderà senza fare nulla, aspettando il passo successivo.

Dispose

Come visto nella sezione relativa al world, il robot riceverà un messaggio "dispose" nel caso in cui sia stato ucciso da un altro agente, a tale messaggio, il robot reagirà uscendo dal MAS.

```
receivedE(world, dispose) :>
    write('My life ended, goodbye...'), nl,
    halt. /* close agent */
```

```
Received attack request from robot3 versus robot1
Inflicted damage = 61
robot1 died!
```

robot1:

```
robot1: begin new step, time 8

Sensing environment...
My life ended, goodbye...
```