RELAZIONE PROGETTO **ASSISTED LIVING**

Università degli studi di Torino

Corso di Laurea Magistrale in Informatica

Intelligenza Artificiale e Laboratorio 2015/2016

DOCENTE:

Prof. Piero Torasso

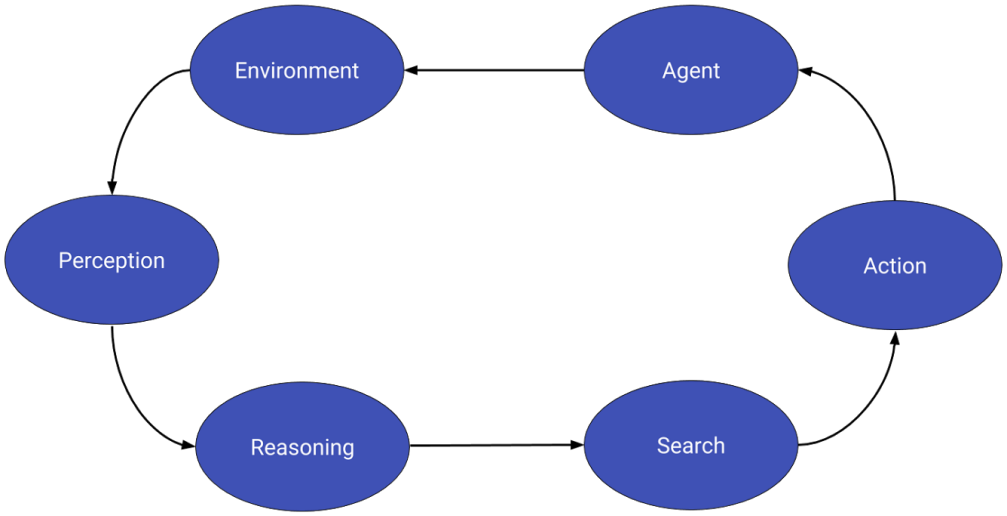
RELAZIONE A CURA DI:

823832 **Giovanni Bonetta**
762661 **Riccardo Renzulli**
763056 **Gabriele Sartor**

STRUTTURA GENERALE

IL CICLO PRINCIPALE

Per questo progetto abbiamo sviluppato un agente intelligente che combina sia aspetti deliberativi sia reattivi. Tale agente assume il ruolo di assistente mobile per soddisfare, se possibile, le richieste di cibo provenienti da persone anziane.

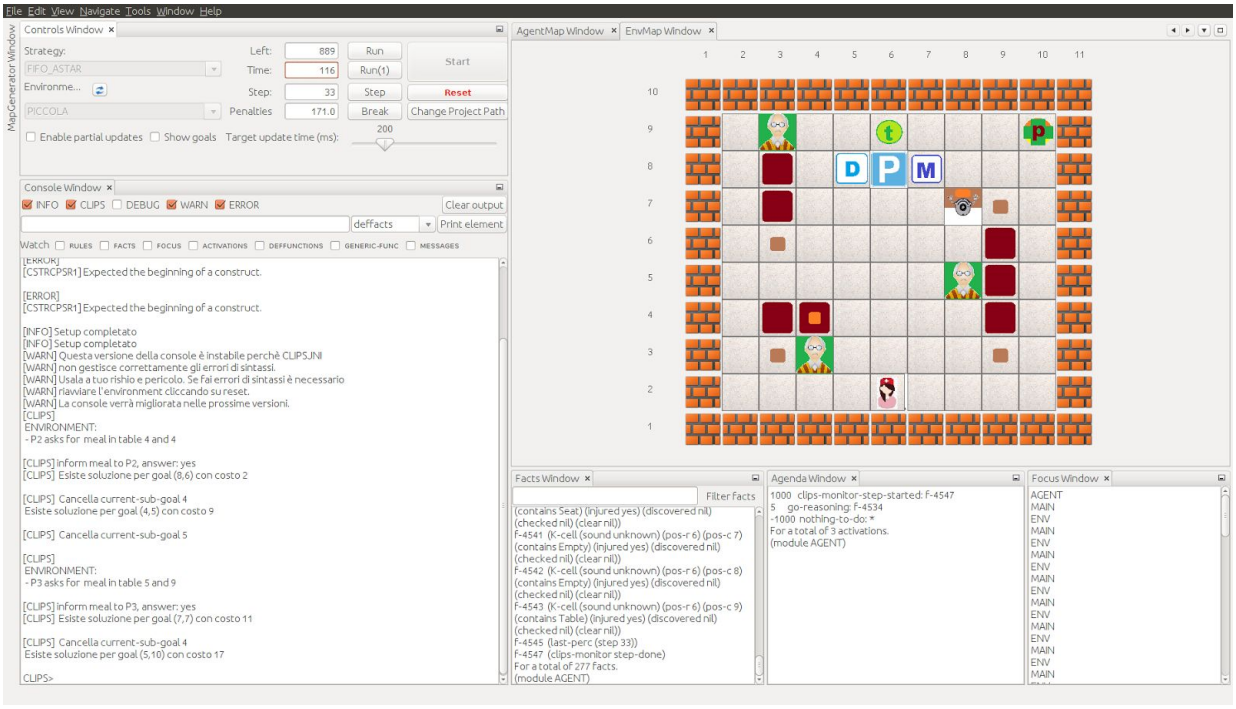


In linea generale il ciclo principale è il seguente: l'agente prima processa le percezioni appena attinte dal mondo, dopo delibera quale richiesta soddisfare e decide le azioni da eseguire per portarla a termine. Questo processo si ripete finchè ci sono ancora richieste in attesa di risposta, tutto ciò entro un tempo massimo prestabilito.

CLIPSMONITOR (GUI)

Come ambiente grafico abbiamo usato il software CLIPSMONITOR[1] sviluppato da Diego Rullo. CLIPSMONITOR è un IDE che ci è stato utile sia nella fase di sviluppo del nostro progetto sia in quella di debugging del medesimo.
Ora presentiamo i moduli cablati nella GUI.

[1] *https://github.com/diegorullo/ClipsMonitor*



I MODULI

Per quanto riguarda l'implementazione del progetto nel linguaggio CLIPS, sono stati aggiunti o completati questi moduli:

- AGENT
- PERCEPTION
- REASONING
- SEARCH (A-STAR e BFSEARCH)
- ACTION

AGENT

Questo modulo, fornito inizialmente in una versione embrionale, è stato sviluppato da noi attraverso la stesura di un insieme di regole aventi come scopo principale quello di aggiornare lo stato interno dell'agente ogni qualvolta viene eseguita un'azione. Abbiamo anche sfruttato la possibilità di creare nuove *deffunctions* in CLIPS per calcolare la nuova direzione corrente del robot.

```
(defrule turnLeft
    (declare (salience 10))
    (exec (step ?step) (action Turnleft))
    ?k <- (K-agent (time ?time) (step ?step) (direction ?dir))
    =>
    (modify ?k (time (+ ?time 2)) (step (+ ?step 1)) (direction (getDirectionTurnLeft ?dir)))
)

(defrule turnRight
    (declare (salience 10))
    (exec (step ?step) (action Turnright))
    ?k <- (K-agent (time ?time) (step ?step) (direction ?dir))
    =>
    (modify ?k (time (+ ?time 2)) (step (+ ?step 1)) (direction (getDirectionTurnRight ?dir)))
)

(defrule forward
  (declare (salience 10))
  (exec (step ?step) (action Forward))
  ?k <- (K-agent (time ?time) (step ?step) (pos-r ?r) (pos-c ?c) (direction ?dir))
  (perc-vision (step ?step) (time ?time) (perc2 Empty|Parking))
  =>
  (modify ?k (time (+ ?time 1)) (step (+ ?step 1)) (pos-r (getForwardR ?dir ?r)) (pos-c (getForwardC ?dir ?c)))
)
```

Inoltre sono state implementate ulteriori azioni, oltre a quelle di movimento, come *loadMeal*, *loadPill*. *loadDessert*, *deliveryMeal*, *informMeal* ecc.
Ad esempio, la regola *loadMeal* viene attivata quando si verificano le seguenti condizioni: il robot si trova davanti ad un *MealDispeser*, ha uno scompartimento libero ed esiste un fatto *exec (action LoadMeal)*, asserito dal modulo ACTION in base alle azioni pianificate dal modulo REASONING. In questo modo l'agente sarà carico (*loaded yes*) e potrà in seguito portare il cibo ad un paziente grazie all'azione *DeliveryMeal*.

```
(defrule loadMeal
  (declare (salience 10))
  (exec (step ?step) (action LoadMeal) (param1 ?y) (param2 ?x) (param3 ?type))
  ?k <- (K-agent (time ?time) (step ?step) (pos-r ?r) (pos-c ?c) (direction ?dir) (free ?free) (waste no) (content $?cont))
  (test (> ?free 0))
  (perc-vision (step ?step) (time ?time) (perc2 MealDispenser))
  (test (= ?x (getForwardC ?dir ?c)))
  (test (= ?y (getForwardR ?dir ?r)))
  =>
  (modify ?k (time (+ ?time 15)) (step (+ ?step 1)) (loaded yes) (free (- ?free 1)) (content (insert$ $?cont 1 ?type)))
)
```

PERCEPTION

Nel modulo PERCEPTION il robot gestisce le percezioni dal mondo esterno in un certo *step* grazie ai fatti *perc-vision* provenienti dal modulo ENV. L'agente quindi, ad

ogni *step* e in base ai nuovi *perc-vision*, aggiorna, tramite delle regole *update*, i fatti *K-cell*. Quest'ultimi sono utili all'agente per conservare la propria conoscenza del mondo esterno in un certo momento.

REASONING

Il modulo REASONING è il cuore centrale della fase di pianificazione dell'agente e perciò ha come scopo principale quello di gestire le varie richieste provenienti dai pazienti.
Ogni richiesta avrà un fatto *main-goal* associato con i relativi *sub-goals* e viene designata come richiesta corrente quella a cui viene data una priorità maggiore. Nel nostro progetto abbiamo dato sempre priorità alle richieste più vecchie nel tempo.
In tal modo viene prodotto un piano per portare a termine la richiesta corrente grazie ai fatti *plan-action*. Una volta eseguito verrà selezionata la prossima richiesta corrente.
All'interno di questo modulo viene gestito anche il replanning del percorso nel caso in cui ad esempio un anziano si sia spostato davanti all'agente intralciando il suo passaggio.
In realtà abbiamo creato un modulo REASONING diverso per ogni strategia diversa adottata; le strategie possono essere suddivise in due macro gruppi: quelle che sfruttano entrambi gli slot del piano di carico dell'agente e quelle no. Esse verranno descritte nel dettaglio nei prossimi paragrafi.

SEARCH

Il modulo SEARCH si occupa della pianificazione dei percorsi che l'agente deve seguire per portare a termine un'azione. Nel nostro progetto il modulo SEARCH corrisponde a due moduli distinti: BFSEARCH e A-STAR. Il primo trova il percorso migliore (in termini di minor numero di azioni di movimento necessarie) attraverso una ricerca in ampiezza, il secondo invece si basa sulla strategia di ricerca informata *A** dove come euristica abbiamo impiegato la distanza di Manhattan.

```
(deftemplate node
   (slot ident)
   (slot gcost)
   (slot fcost)
   (slot father)
   (slot pos-r)
   (slot pos-c)
   (slot direction)
   (slot open)
)
```

In entrambe le strategie di ricerca il nodo ha la stessa struttura, con la differenza che in *A** abbiamo tenuto traccia anche dello slot *fcost*, ossia il costo per espandere il nodo (*gcost*) sommato al costo stimato per arrivare dal nodo corrente al nodo goal.

```
(defrule achieved-goal
  (declare (salience 100))
    (current ?id)
    (temp-goal ?r ?c)
    (node (ident ?id) (pos-r ?pos-r) (pos-c ?pos-c) (direction ?direction) (gcost ?g))
    (test (= ?r (getForwardR ?direction ?pos-r)))
    (test (= ?c (getForwardC ?direction ?pos-c)))
    ?current-sub-goal <- (current-sub-goal (id ?ident) (pos-r-target ?pos-r-target) (pos-c-target ?pos-c-target)
                         (action-to-do ?action-to-do) (param1 ?param1) (param2 ?param2) (param3 ?param3))
    ?sub-goal <- (sub-goal (id ?ident))
    =>
    (printout t " Esiste soluzione per goal (" ?pos-r "," ?pos-c ") con costo "  ?g crlf)
    (assert (plan-action-aux (plan-step (+ ?g 1)) (action ?action-to-do) (param1 ?param1) (param2 ?param2) (param3 ?param3)))
    (assert (stampa ?id))
)
```

Il goal viene raggiunto quando l'agente si trova in una delle caselle adiacenti all' obiettivo ed è rivolto verso di esso. L'output sarà quindi un certo numero di azioni pianificate dette *plan-action-aux*. Esse verranno poi trasformate in *plan-action*.
All'interno di questo modulo abbiamo gestito anche il caso di fallimento del goal grazie all'asserzione del fatto *failure*: se in un certo step non è possibile raggiungere il nodo goal allora verrà eseguita un'azione di wait.

ACTION

Il modulo ACTION si occupa della definizione delle azioni da eseguire traducendo i fatti *plan-action* in fatti *exec* che verranno poi gestiti dal modulo ENV. In particolare ad ogni step viene tradotta l'azione con *plan-step* minore.

```
(defrule make-exec-action
   ?plan-action <- (plan-action (action ?action) (plan-step ?ps)(param1 ?p1) (param2 ?p2) (param3 ?p3))
   (status (step ?s))
   (not (and
           (plan-action (plan-step ?ps2))
           (test (< ?ps2 ?ps))
        )
   )
   (not (exec (step ?s)))
   =>
   (assert (exec (action ?action) (step ?s) (param1 ?p1) (param2 ?p2) (param3 ?p3)))
   (retract ?plan-action)
)
```

STRATEGIE FIFO

DESCRIZIONE GENERALE

La strategia FIFO (First In First Out) è la base da cui siamo partiti per gestire l'ordine delle richieste da servire. Ogni qualvolta arriva una richiesta il robot risponde allo step successivo con una *inform*.
Nello specifico le nostre strategie si basano tutte sul servire prima le richieste più vecchie. Abbiamo optato per questa scelta con l'obiettivo di minimizzare il numero di step in cui prendiamo penalità a causa di consegne non ancora effettuate, ma a cui abbiamo già risposto in modo affermativo. Dal punto di vista dell'agente robotico ogni richiesta diventa un task da portare a termine e ne viene gestito uno solo alla volta. Il task corrente è memorizzato grazie al fatto *main-goal*. Come da specifiche del progetto i pazienti possono ordinare due tipi di pietanze: meal e dessert. Questo significa che l'agente deve gestire due tipi di main goal.

Ognuno di essi poi è diviso in sotto goal grazie ai fatti *sub-goal*, i quali scompongono il task principale in sotto task più semplici secondo il ben noto principio divide et impera.

Tutti i sotto goal condividono una struttura precisa e indicano l'intenzione di:

- raggiungere un luogo nella mappa;
- eseguire in quel punto un'azione.

Inoltre ognuno di essi ha due identificativi: il proprio, salvato nello slot *id*, e quello del *main-goal* di appartenenza, salvato in *goal-ref*. Gli *id* inoltre hanno la funzione di specificare un ordinamento tra i vari sotto goal: viene eseguito sempre quello che ha *id* minore.

Per fare un esempio, nell'immagine di seguito vediamo come il *main-goal* *DeliveryDessert* viene scomposto in due *sub-goal*.

```
(defrule create-sub-goals-dessert-YES
  (declare (salience 50))
  ?main-goal <- (main-goal (id 1) (k-request ?x))
  (not (sub-goal))  ; controllo che non ci siano già dei sub-goal da portare a termine
  ?y <- (K-dessertstatus (arrivaltime ?arrivaltime) (requested-by ?requested-by) (tpos-r ?tpos-r) (tpos-c ?tpos-c)
                         (delivered ?delivered) (answer ?answer))
  (test (eq ?x ?y))
  ?prescription <- (prescription (patient ?requested-by) (dessert yes))
  ?dessertdispenser <- (K-cell (pos-r ?dessert_pos_r) (pos-c ?dessert_pos_c) (contains DessertDispenser))
  =>
  (assert (sub-goal (goal-ref 1) (id 2) (pos-r-target ?dessert_pos_r) (pos-c-target ?dessert_pos_c)
                    (action-to-do LoadDessert) (param1 ?dessert_pos_r) (param2 ?dessert_pos_c)))
  (assert (sub-goal (goal-ref 1) (id 3) (pos-r-target ?tpos-r) (pos-c-target ?tpos-c)
                    (action-to-do DeliveryDessert) (param1 ?tpos-r) (param2 ?tpos-c)))
)
```

Una volta che il robot avrà eseguito questi due sotto goal, il *main-goal* corrispondente sarà completato.

Esiste poi un ultimo tipo di *main-goal* che riguarda la pulizia dei tavoli. Il robot semplicemente raggiunge il tavolo, lo pulisce e infine scarica la spazzatura nel *trashbasket*. Questo *main-goal* viene eseguito solo nel caso in cui i pazienti abbiano avuto il tempo di mangiare e non ci siano altre richieste di pietanze da servire.

Nei paragrafi seguenti descriviamo le nostre 4 strategie FIFO di questo tipo che considerano un *main-goal* alla volta. Esse differiscono tra di loro per l'ordine di esecuzione dei *sub-goal* e per l'algoritmo di ricerca del cammino.

FIFO BFS

La strategia FIFO *breadth-first search* si basa sull'idea di creare un *main-goal* alla volta calcolando il tragitto da fare in ogni *sub-goal* utilizzando un algoritmo di ricerca in ampiezza. Esso calcola le celle in cui deve passare il robot espandendo ogni nodo con le tre azioni di movimento che possono essere eseguite (*forward*, *turnleft*, *turnright*), facendo così una ricerca su tutto lo spazio delle possibilità. Una volta trovato il percorso che porta dalla cella attuale in una di quelle adiacenti alla posizione target, l'algoritmo termina. Pur essendo computazionalmente costoso, l'algoritmo BFS restituisce la soluzione ottima in termini di numero di azioni effettuate ed è stata la base da cui siamo partiti per sviluppare le altre strategie.

FIFO A-STAR

La strategia FIFO con algoritmo di ricerca *A** è la nostra migliore soluzione per trovare i cammini da far percorrere all'agente. Nel nostro dominio *A** produce una soluzione ottima poichè abbiamo usato l'euristica di Manhattan (euristica consistente). Questo algoritmo ci permette prestazioni migliori in quanto espande meno nodi del precedente. Per quanto riguarda la gestione dei main e sub goal nulla cambia rispetto alla strategia FIFO BFS.

FIFO BFS_v2 E FIFO A-STAR_v2

Queste due strategie sono più "intelligenti" rispetto alle loro rispettive versioni iniziali perchè cercano di limitare viaggi inutili nella particolare situazione in cui l'agente deve eseguire il *main-goal DeliveryMeal* e il paziente ha bisogno delle pillole. Ad esempio, se la pillola deve essere somministrata prima del pasto, vengono eseguiti prima i *sub-goal LoadPill* e *LoadMeal*, successivamente i *sub-goal DeliveryPill* e *DeliveryMeal*. Ovviamente questo porta un vantaggio in termini di penalità nel caso in cui i vari dispenser siano vicini tra loro.

STRATEGIA DOUBLE_A-STAR

DESCRIZIONE GENERALE

Questa strategia mira a sfruttare entrambi gli slot del piano di carico dell'agente per soddisfare richieste pervenute da diversi pazienti.
Il robot, in presenza di più richieste, ne gestisce una coppia.
Ad esempio, se ad un certo istante abbiamo due richieste di *meal* pervenute dai pazienti P1 e P2, verranno eseguite due azioni di *Load* ed in seguito due azioni di *Delivery* (una *Load* e una *Delivery* per ogni richiesta).
Dato che i rifiuti non possono essere caricati sull'agente se esso ha del cibo o delle pillole a bordo, questa situazione è stata gestita come caso particolare. Infatti, le operazioni di *CleanTable* vengono accoppiate solamente tra di loro e seguite da una sola azione di *ReleaseTrash*.

DETTAGLI IMPLEMENTATIVI

In questa strategia, abbiamo modifiche significative nel modulo REASONING a causa della gestione di due *main-goal* in contemporanea. Durante l'esecuzione, avremo sempre al massimo due fatti di tipo *main-goal* con slot *id* diversi (nel nostro caso abbiamo solo 1 e 2). Nel momento in cui il robot riceve una richiesta e non esiste un fatto *(main-goal (id 1) ...)* o *(main-goal (id 2) ...)* ne viene creato uno per gestire quel determinato goal. Per gestire contemporaneamente i due *main-goal* sfruttiamo la presenza del fatto *last-current*, il quale ci indica l'ultima azione eseguita e a quale *main-goal* appartiene. In questo modo, ad ogni ciclo andiamo a gestire il primo *sub-goal* del *main-goal* trascurato nel passo precedente.
Inoltre, questa strategia tiene in considerazione la presenza di più dispenser. Nel caso in cui il robot deve muoversi verso un *MealDispenser, DessertDispenser o PillDispenser*, si dirige verso il più vicino. Lo stesso avviene per la scelta del *TrashBasket* per l'esecuzione di una *ReleaseTrash*. Nella seguente immagine

vediamo come viene scelto il *TrashBasket* più vicino nel caso in cui l'agente stia per eseguire una *ReleaseTrash*.

```
(defrule create-current-sub-goal-relase-OK-twotrashbasket
  (declare (salience 48))
  (not(plan-action))
  ?last-current <- (last-current ? ~ReleaseTrash)
  ?sub-goal <- (sub-goal (goal-ref ?goalid) (id ?id) (action-to-do ReleaseTrash) (param1 ?param1) (param2 ?param2) (param3 ?param3))
  ?trashbasket <- (K-cell (pos-r ?trash_pos_r) (pos-c ?trash_pos_c) (contains TrashBasket))
  ?trashbasket2 <- (K-cell (pos-r ?trash_pos_r2) (pos-c ?trash_pos_c2) (contains TrashBasket))
  (K-agent (pos-r ?r-agent) (pos-c ?c-agent))
  (test (neq ?trashbasket ?trashbasket2))
  (not (test (< (+ (abs (- ?trash_pos_r2 ?r-agent)) (abs (- ?trash_pos_c2 ?c-agent)))
                (+ (abs (- ?trash_pos_r ?r-agent )) (abs (- ?trash_pos_c ?c-agent))))))
  =>
  (retract ?last-current)
  (assert (last-current ?goalid ReleaseTrash))
  (assert (current-sub-goal (goal-ref ?goalid) (id ?id) (pos-r-target ?trash_pos_r) (pos-c-target ?trash_pos_c)
                            (action-to-do ReleaseTrash) (param1 ?trash_pos_r) (param2 ?trash_pos_c) (param3 ?param3)))
  (focus A-STAR)
)
```

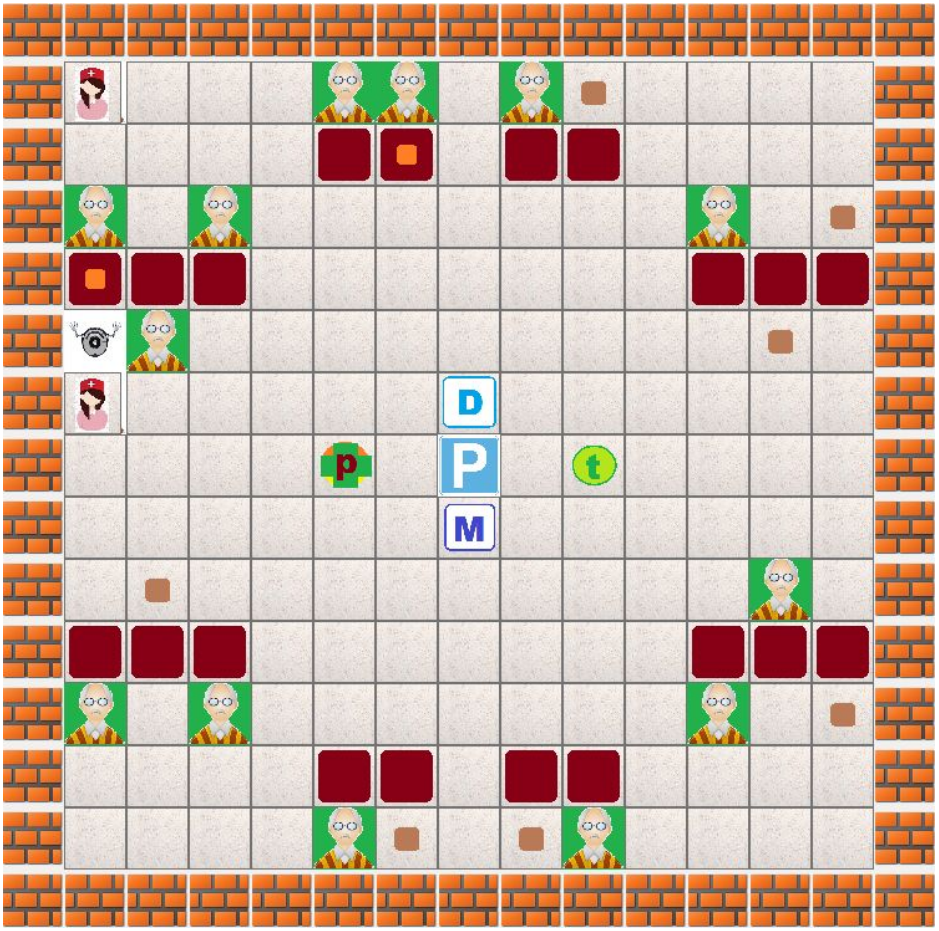
Come possiamo notare nel codice, dopo aver trovato un *Trashbasket* in posizione *(?trash_pos_r, ?trash_pos_c)* con distanza minima dal robot, l'algoritmo richiama il modulo A-STAR per generare un piano per raggiungerlo.
Per la scelta del nuovo *current-sub-goal*, abbiamo sfruttato la *salience* per dare priorità assoluta alle operazioni di *CleanTable* ed una leggermente più bassa per quelle di *ReleaseTrash*. Dopodiché abbiamo le regole che catturano le situazioni residuali. Così facendo, non è possibile avere un'azione di *Load* tra le azioni di gestione dei rifiuti, di conseguenza evitiamo le penalità derivanti dalla presenza di cibo e spazzatura contemporaneamente a bordo del robot.

REPLANNING E FALLIMENTO

Dopo aver scelto un *sub-goal* da gestire, il nostro agente crea un piano per soddisfarlo. Fino a quando non porta a termine tutte le azioni pianificate *(plan-action)*, il robot continua ad eseguire le operazioni senza utilizzare il modulo REASONING. Tuttavia, data la natura dinamica dell'ambiente, in alcuni casi potrebbe accadere che una persona si trovi davanti al robot durante l'esecuzione del piano. In quel momento, l'agente percepisce l'impossibilità di proseguire sul cammino pianificato in precedenza e cancella l'intero piano.

```
(defrule action-unfeasible
  (declare (salience 80))
  (not (replanning))
  (status (time ?time))
  (perc-vision (step ?step) (time ?time) (perc2 PersonStanding))
  (plan-action (plan-step ?ps) (action Forward))
  (not (plan-action (plan-step ?plan-step&:(< ?plan-step ?ps))))
  =>
  (printout t "Non puoi eseguire Forward!" crlf)
  (assert (replanning))
)
```

Non avendo cambiato il *current-sub-goal*, non serve altro che passare nuovamente al modulo SEARCH per trovare il nuovo cammino.
In alcuni casi è possibile che il *goal* sia impossibile da raggiungere ad un certo istante. Per esempio, questo può essere causato dalla presenza di una persona davanti alla nostra destinazione e ci impedisce di raggiungerla. Il nostro robot, se non trova nessun cammino che porta al raggiungimento del *goal*, decide di eseguire un'azione di *Wait*, con la speranza che l'ambiente si evolva in modo tale da poter rendere di nuovo raggiungibile il target.



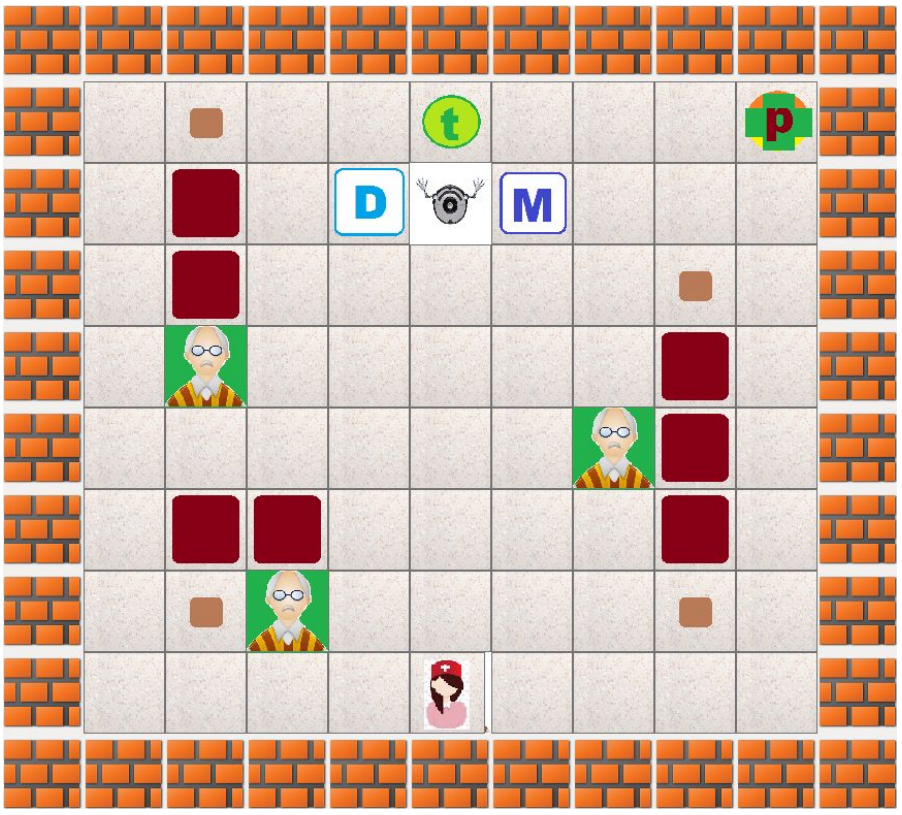
Quando si verifica una situazione di fallimento nel modulo SEARCH, viene asserito un fatto *failure* e viene pianificata un'azione di *Wait*. Il fatto *failure* ci serve come flag per cancellare le asserzioni create durante la ricerca del piano e per riportare il REASONING alla scelta del *current-sub-goal*. Dopodiché l'agente si ritrova nel suo regolare flusso di esecuzione.

SPERIMENTAZIONE

Al fine di fare un confronto significativo tra strategie, esse sono state simulate su mappe di diverse dimensioni. Di seguito vediamo i risultati, ossia le penalità, ottenuti su mappe piccole (10x11), medie (15x15) e grandi (20x20).

MAPPE PICCOLE

Per il test del programma su mappe di piccole dimensioni, abbiamo usato una sola mappa con diverse sequenze di richieste.



MAPPA HISTORY_CONTEMPORANEE

In questo esempio, tutti e tre gli anziani richiedono un *meal* e un *dessert* nei primi step della *history*. Di conseguenza, nei primi istanti abbiamo già ricevuto tutte le richieste.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
1231	1157	1231	1157	1284

MAPPA DEFAULT

In questo esempio, tutti e tre gli anziani richiedono un *meal* ma la richiesta del paziente P1 allo *step* 30 viene rifiutata in quanto non era seduto in quell'istante. Anche la richiesta di *dessert* di P1 viene rifiutata per via della sua dieta, mentre quella di *dessert* per P2 viene soddisfatta.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
562	586	562	586	628

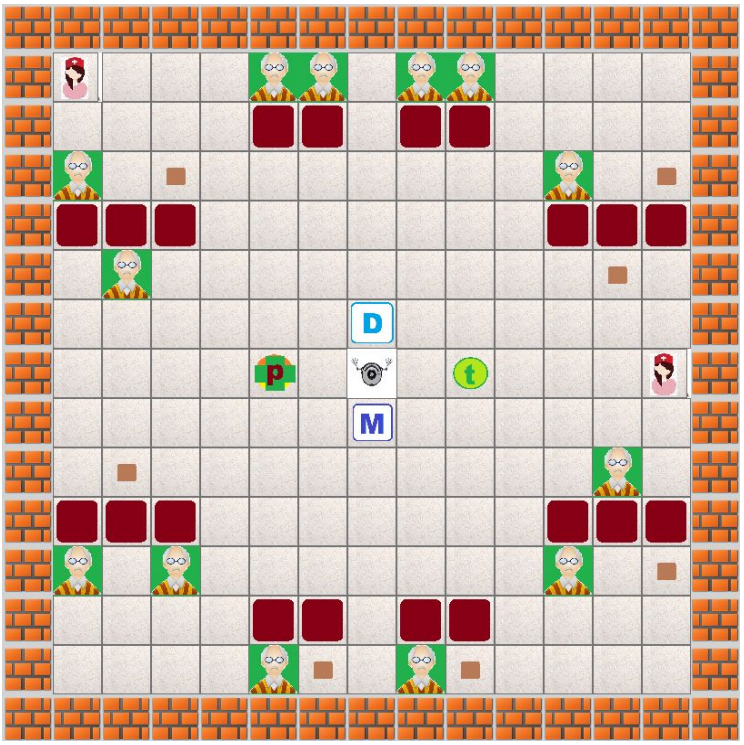
MAPPA SIMPLE

Qui vengono fatte tre richieste di *meal* e tre richieste di *dessert*. Sono tutte soddisfacibili tranne quella di *dessert* del paziente P1.

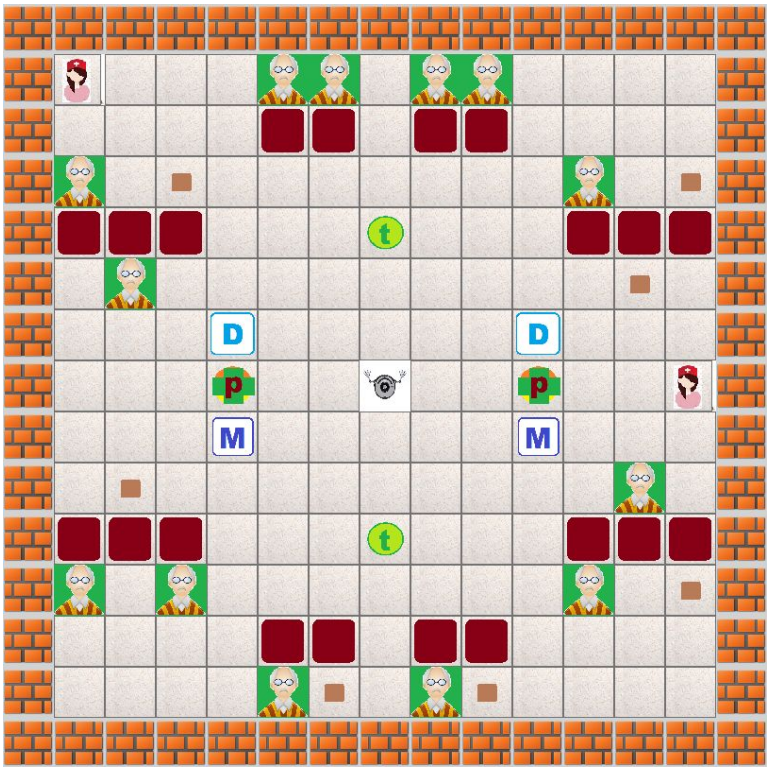
BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
743	549	743	549	791

MAPPE MEDIE

Per il test del programma su mappe di medie dimensioni, abbiamo usato due mappe con diverse richieste.
Le mappe "media fallimento", "media history distanze1" e "media_history sequenziale" condividono la stessa seguente struttura:



mentre nella mappa “media history dispenser” la configurazione dei dispenser è diversa rispetto alla precedente:



MAPPA HISTORY_DISTANZE1

In questo esempio gli anziani fanno 8 richiste di *meal* e 4 di *dessert* e sono tutte portate a termine. La particolarità è che le richieste avvengono a gruppi di 3 alla volta e si lascia scorrere molto tempo tra un gruppo e l'altro. Per esempio agli step 1,2,3 vengono fatte tre richieste di meal. Poi le seguenti 3 agli step 200,201,202... e così via.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
3750	3496	3750	2944	3565

MAPPA HISTORY SEQUENZIALE

in questo esempio le richieste vengono fatte lasciando trascorrere 100 time tra una e l'altra. vengono soddisfatte tutte.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
659	668	596	668	596

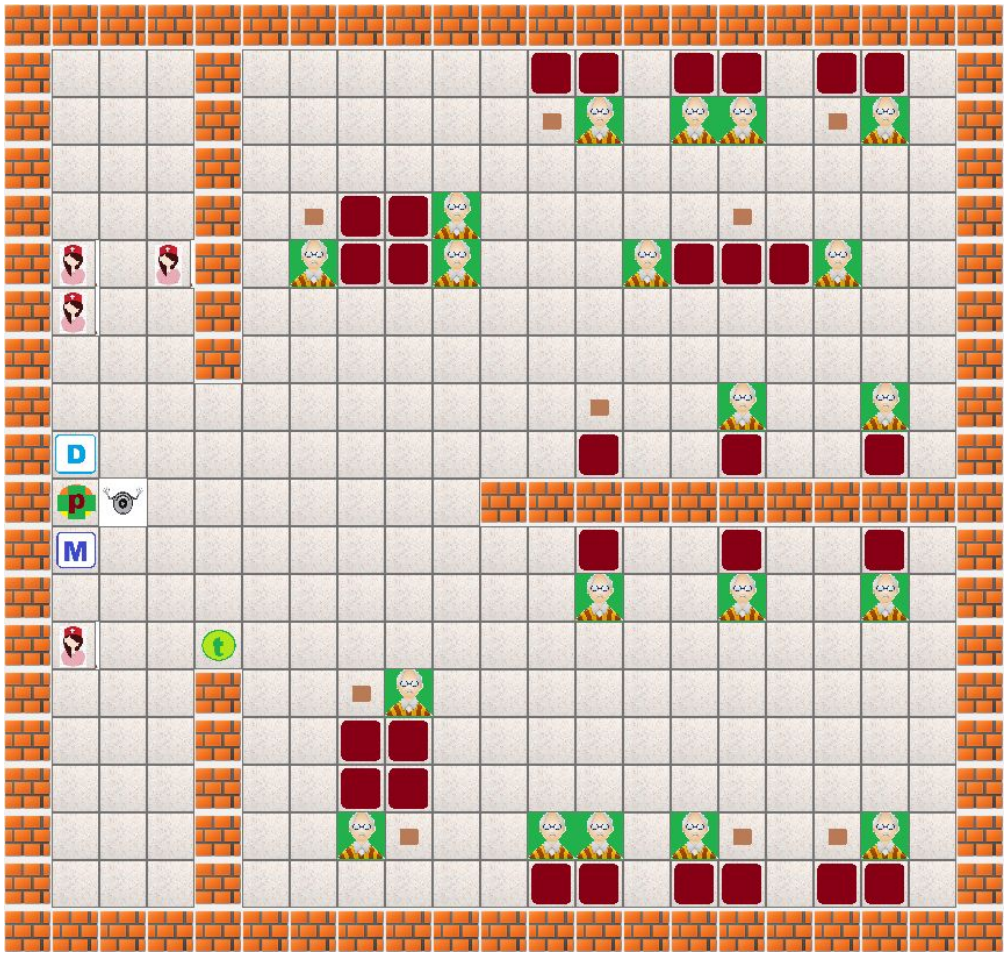
MAPPA HISTORY DISPENSER

In questo esempio le richieste sono uguali a quelle in media history distanze1.
La particolarità è che nella mappa sono presenti due dispensers dello stesso tipo e due trashbaskets. Questo ci è servito per sviluppare una strategia che scegliesse gli obiettivi con distanza più piccola.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
3750	3637	4338	3503	3716

MAPPE GRANDI

Per evidenziare le caratteristiche delle diverse strategie, abbiamo testato il comportamento del nostro agente su mappe di dimensioni maggiori. Tutti gli esempi condividono la mappa qui sotto riportata e le differenze tra loro sono date dalla natura e sequenza delle richieste.



MAPPA HISTORY_SEQUENZIALE

Nella history di questo esempio abbiamo 10 richieste di *meal* e 5 di *dessert*. Le richieste risultano inviate con un certo ritardo tra di loro in modo tale da poter essere potenzialmente soddisfatte in modo sequenziale dall'agente.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
6426	4214	6399	4169	5100

MAPPA HISTORY_DISTANZE

In questo caso, abbiamo 6 richieste di *meal* e di *dessert*. Con intervallo abbastanza regolare, il robot riceve insiemi di 3 richieste vicine nel tempo.

BFS	BFS_v2	ASTAR	ASTAR_v2	DOUBLE_ASTAR
5530	3433	5560	3433	4896

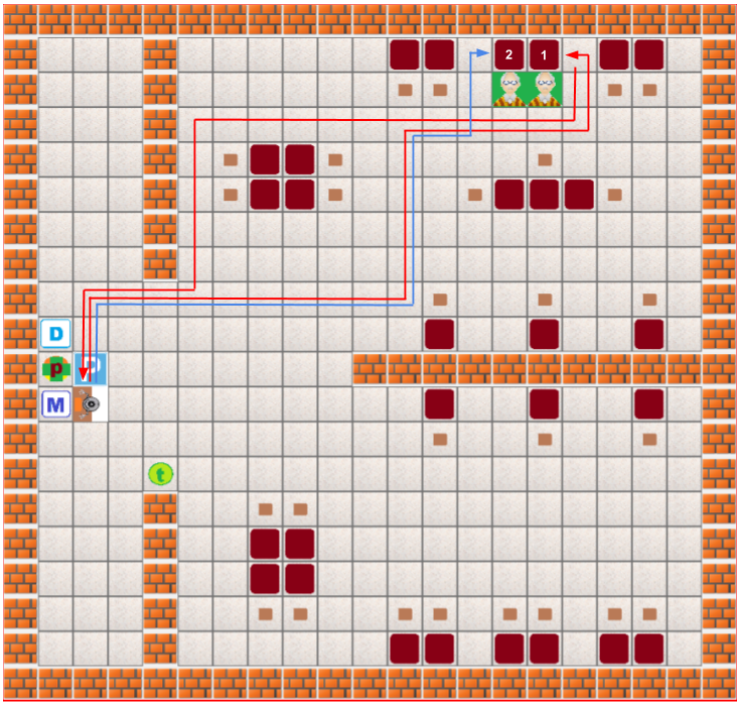
CONCLUSIONE

Per quanto riguarda le strategie FIFO semplici con *breadth-first search* e *A**, possiamo notare come le penalità tipicamente non siano molto diverse poiché trovano entrambe soluzioni ottime. Ci sono casi in cui esse differiscono: per uno stesso target possono esistere più soluzioni ottime e non è detto che le strategie trovino esattamente la stessa. Questo avviene ad esempio nella mappa *media_history_dispenser* dove con FIFO BFS otteniamo 3750 punti di penalità mentre con FIFO A-STAR 4338; tale differenza è dovuta alla dinamicità dell'ambiente in quanto è possibile che seguendo un certo cammino ottimo l'agente si imbatta in un anziano mentre in un altro no.

Nelle strategie FIFO BFS_v2 e FIFO A-STAR_v2, come ci aspettavamo, otteniamo delle perfomance nettamente migliori rispetto a quelle descritte prima proprio per il fatto che in certe situazioni riusciamo a sfruttare il doppio slot diminuendo il cammino totale che deve percorrere l'agente.

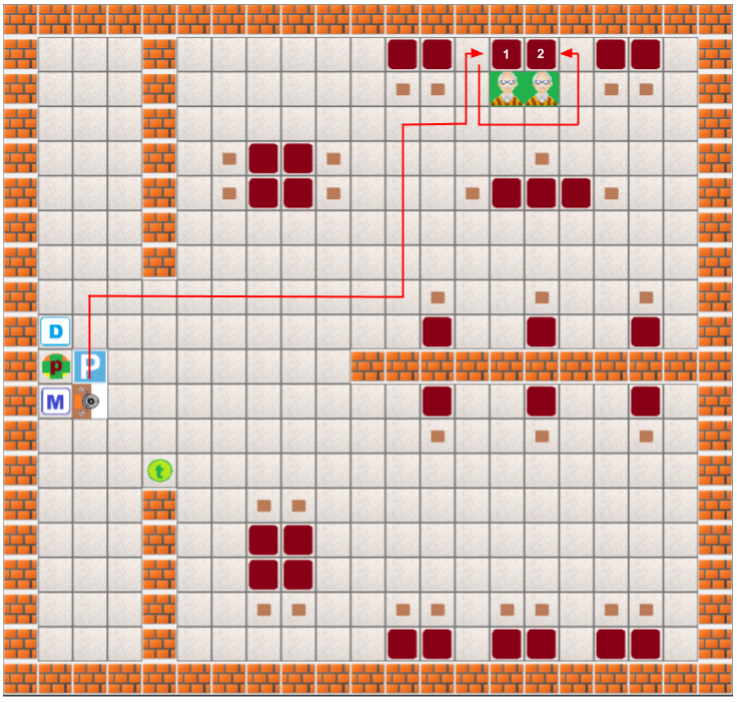
Un confronto più interessante può essere quello tra le strategie FIFO semplici e la FIFO DOUBLE_A-STAR. Come scritto precedentemente, la strategia FIFO DOUBLE_A-STAR cerca sempre di parallelizzare le varie richieste in modo da poter sfruttare entrambi gli slot del piano di carico dell'agente. I vantaggi portati da questa strategia però sono relativi: essi dipendono sia dalla tipologia delle richieste sia dalle

distanze tra vari dispenser nella mappa. Infatti, se ci sono molte richieste di *meal* che includono anche la pillola, questa strategia è meno efficace rispetto alle strategie FIFO BFS_v2 e FIFO A-STAR_v2 in quanto i pazienti che stiamo servendo contemporaneamente potrebbero anche non essere seduti in tavoli vicini. Se invece le richieste non includono anche la pillola e gli anziani sono vicini tra loro la strategia FIFO DOUBLE_A-STAR si comporta meglio. Quest'ultimo caso può essere osservato nelle seguenti immagini.



Richieste di *meal* eseguite sfruttando la strategia FIFO A_STAR_v2.
Penalità: 657.

.



Richieste di *meal* eseguite sfruttando la strategia FIFO DOUBLE_A_STAR.
Penalità: 537.

Vediamo come la penalità della strategia double sia minore rispetto a quella della strategia A-STAR_V2 (e di tutte le altre).