PRESENTAZIONE DEI PROGETTI TIPO
PARTE I, uso delle librerie.

I progetti tipo utilizzano principalmente quanto visto sulla architettura di un agente intelligente e sugli algoritmi di ricerca, in particolare A*.

Nella prima parte di questa presentazione descriviamo l'uso della libreria attraverso alcuni "snippet" relativi al suo utilizzo in un progetto.
Nella seconda parte si dà un'idea di alcuni progetti tipo.

Parte I. Agenti e snippet.

1.1. Agenti reattivi.

Lo schema di base per tali agenti è il seguente:

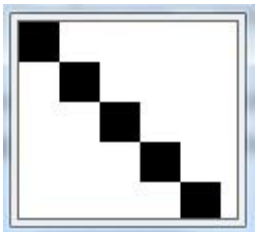
```
type(open:obiettivo).
  %  l'obiettivo generale dell'agente
type(open:azione).
  %  le azioni dell'agente, che cambiano lo stato del mondo+agente
pred(agente(obiettivo)).
  % agente(Goal) :  genera ed esegue la sequenza di azioni dell'agente in base alla implementazione dei predicati aperti
decidi_azione ed esegui
  %  MODO (+)  genera un comportamento
open_pred(fine(obiettivo)).
  %  fine(G) :  l'obiettivo G è raggiunto (e l'agente termina).  MODO  (+) semidet
open_pred(decidi_azione(obiettivo, azione)).
  %  decidi_azione(G, A) : con goal G l'agente decide di fare l'azione A
  %  MODO:   (+,-) det (nondet per agenti non deterministici)
open_pred(esegui(azione)).
  % esegui(A) : l'agente esegue l'azione A.  MODO  (+) det

agente(Obiettivo) :-
        fine(Obiettivo),!
        ;
        decidi_azione(Obiettivo, Azione),!,
        esegui(Azione),
        agente(Obiettivo).
```

Questo agente è situato in un mondo esterno e lo ***"stato esterno"*** del mondo + agente è rappresentato da predicati dinamici; la proprietà statiche del mondo e le regole generali che lo governano sono rappresentati da clausole statiche.

Un agente reattivo ***reagisce agli eventi***, cioè ai cambiamenti dello *stato esterno*, decidendo ogni volta una singola azione da eseguire ed eseguendola; *la decisione è determinata da ciò che l'agente percepisce riguardo allo stato esterno e dall'obbiettivo dell'agente.* Nel caso del modello di agente qui considerato gli eventi sono esclusivamente il risultato delle azioni dell'agente, ovvero il mondo non genera eventi asincroni. Dunque il comportamento consiste in una sequenza di azioni decise ed eseguite man mano dall'agente; dopo ogni evento, che consiste nella esecuzione di una azione, si ha una nuova decisione. Per simulare l'accadimento degli eventi nel tempo e mostrare l'esecuzione

passo-passo, il predicato **esegui** sarà concluso con una interazione con l'utente, come mostrato dal seguente codice che "implementa" i predicati aperti del modello generale.

Esempio di agente reattivo "Aggira Ostacoli". Si tratta di un agente che si muove in un mondo come illustrato in Fig. 1 (in nero, gli ostacoli), e deve raggiungere una posizione goal a partire dalla posizione iniziale.



Trattandosi di un ambiente geometrico, usiamo la geometria piana data nel modulo **geometria.pl**, dove i punti sono pensati come quadretti in una quadreattatura dello spazio bidimensionale. Definisce i tipi point (i punti) e direzione (le direzioni della rosa dei venti), lo sposamento di un quadretto in una data direzione, la rotazione antioraria, ecc.; vedi le specifiche nel codice. Vediamo ora una rappresentazione del mondo.

Tipi:

```
type([va_in(direzione,point)]:azione).
   %  unica azione, andare nel prossimo quadretto nella direzione data
type([mossa(point,point)]:evento).
   %  mossa(P1,P2): evento che consiste in uno spostamento daP1 a P2 dell'agente
type([target(point)]:obiettivo).
   %  target(P) : l'agente deve raggiungere il punto P
```

Predicati che rappresentano la parte statica del mondo e le leggi che lo governano:
Oltre alla geometria, usiamo i predicati world/2 e content/2 per rappresentare il mondo di Fig.1 come segue:

```
%  mondo 5 x 5 con ostacoli sulla diagonale:  per I compreso fra 1 e 5,  il punto (I,I) contiene un ostacolo
world(5, 5).
content(point(I,I), ostacolo) :-  between(1,5,I).
```

Predicati che rappresentano la parte dinamica del mondo. Qui ci basta un predicato in/2 per gestire posizione e direzione di movimento:

```
in(Dir,P)
% l'agente si trova in P e Dir è la direzione con cui è arrivato in P
```

Con questa rappresentazione dello stato del mondo+agente, implementiamo i predicati aperti come segue:

```
fine(target(G)) :-    in(_,G).
      %fine se l'agente ha raggiunto la posizione target
decidi_azione(target(G), va_in(NewDir,R)) :-
       % come nuova direzione decido la prima in senso antiorario che si avvicina al target G e porta a una posizione
       %   libera calcolata con il predicato leftmost, che applica la strategia di aggiramento antiorario degli ostacoli
          in(Dir,P),
          leftmost(G, P, NewDir, R)
```

```
esegui(va_in(Dir2,P2)) :-
      retract(in(_, P1)),     % cancella lo stato attuale del  (mondo+agente), ricavando la posizione attuale P1
      step(Dir2, P1, P2),     %  trova la posizione P2 adiacente a P1 in direzione Dir2
      assert(in(Dir2,P2)),    %  asserisce che ora l'agente si trova in P2 in direzione Dir2
      simula(mossa(P1,P2)).   %simula l'accadimento dell'evento mossa(P1,P2)
```

Per fare in modo di scandire gli eventi nella visualizzazione del comportamento dell'agente si può implementare simula come interazione con l'utente. Qui sotto si fornisce una versione minimale di simula, che si limita a stampare un evento e richiedere all'utente di dare una risposta arbitraria (ad es. INVIO) e consentendo di abortire la simulazione digitando "a".

```
simula(Evento) :-  writeln(Evento),  readln(Risp),
            ( Risp = [a|_] -> abort ; true ).
```

Infine è necessario attivare l'agente a partire da uno stato iniziale. Il seguente codice fa partire il nostro agente da una posizione iniziale P0, con target G:

```
start(P0, G) :-
        retractall(in(_,_)),
        %  cancello il contenuto della base dati dinamica dai risultati di eventuali esecuzioni precedenti
        assert(in(nord:nord,P0)),  %agente inizialmente ha direzione nord:nord
        % parte l'esecuzione dell'agente
        agente(empty, target(G)).
```

Come detto nei commenti, il predicat leftmost applica la *strategia di aggiramento antiorario* degli ostacoli, cioè se il punto target è in direzione Dir a partire dalla posizione corrente del robot, leftmost parte da Dir e procede per rotazioni a sinistra (ad es. da nord:est a nord:nord a nord:ovest …) fino alla prima direzione che porta in un passo ad una posizione libera da ostacoli. Con questa strategia è garantito l'aggiramento degli ostacoli convessi, mentre ostacoli concavi potrebbero intrappolare l'agente, che seguiterà a girare su uno stesso percorso ciclico. Si vedano il codice e gli esempi suggeriti dal comando `aggiratore_help` di **aggiratore.pl**.

NOTA. OGNI modulo ha un suo help che indica come usarlo.

1.1.1. Usare un repository di mondi (modulo **mondi.pl** e file **repository.pl**).

Il codice precedente usa un unico mondo, quello di Fig.1. Nel progetto si chiederà di analizzare il comportamento dell'agente in più mondi; allo scopo conviene salvare in un file i mondi di interesse ed eventualmente disporre di una generazione random dei mondi. Nel modulo mondi.pl si danno esempi di predicati per la generazione random, per la rappresentazione di mondi mediante mappe di caratteri e per la gestione di un repository di mondi. I predicati di rappresentazione del mondo sono ternari perché il primo argomento è il nome del mondo, necessario avendo a che fare con più mondi:

```
pred(world(any, integer, integer)).
  %   world(W, NCol, NRow) : W è un mondo caricato in memoria dinamica con NCol colonne e NRow righe
  %  MODO  (?,?,?) nondet
 :- dynamic(world/3).
pred(content(any, point, contenuto)).
  %  content(W, P, C) :  il punto P di W contiene C        MODO (?,?,?) nondet
 :-  dynamic(content/3).
```

I predicati world/3 e content/3 sono dinamici sia per supportare la generazione random, sia perché alcuni tipi di contenuto di un quadretto possono variare dinamicamente; ad esempio se in un mondo w1 si ha content(w1, point(3,4), agente) e l'agente si sposta in (3,5), bisogna cancellare content(w1, point(3,4), agente) ed asserire content(w1, point(3,5), agente).

1.1.2. Usare la grafica fornita dalla libreria draw_world.pl.

Nelle librerie lib è stato inserito un semplice modulo draw_world.pl, che consente di disegnare un mondo in una quadrettatura dello spazio bidimensionale. I predicati sono:

```
pred(new_picture(atom, integer, integer, integer)).
  %  new_picture(PicName, NCol, NRow, Size) :  apre una nuova immagine in cui i punti sono quadretti di lato Size,
  %   con NCol colonne e NRow righe, identificata dal nome PicName
  %  MODO (++,++,++,++) det
pred(draw_fig(atom, fig, point)).
  %  draw_fig(Pic, Fig, point(Row, Col)):  Disegna sulla immagine Pic la figura Fig nel punto point(Row, Col)
  %  MODO (++,++,++) semidet (fallisce se PicName non è stata aperta con new_picture )
pred(del_fig(atom, fig, point)).
  %  del_fig(PicName, Fig, Point) :  cancella la figura Fig in posizione Point
  %  MODO (++,??,++) semidet
pred(move_fig(atom, fig, point, point)).
%  move_fig(Pic, Fig, P1, P2) : cancella la figura Fig in posizione P1 di Pic e la ricopia in posizione P2
%  MODO  (++, ??, ++, ++)  semidet
```

I tipi di figura disegnabili supportati al momento sono:

```
type([box(integer,list(filling)), % box(Size, Fill) : quadrato con lato lungo Size e filling Fill
     circ(integer, list(filling))     % circ(Size, Fill) : cerchio con diametro lungo Size e filling Fill
    ]: fig).
type([col(colour), text(atom)]:filling).
%  Riempimento di una figura; per ora solo colore e testo
```

Ad esempio, dopo aver caricato la libreria draw_world, provate nell'ordine le query:

```
?- new_picture(prova, 10, 10, 20).
?- draw_fig(prova, box(20, [col(yellow),text(a)])).
?- draw_fig(prova, circ(10, [col(green)]), point(5,7)).
```

Vedere anche draw_loaded_world e draw_content nel modulo **mondi.pl.**

1.2. Agenti reattivi con stato interno.

Un agente puramente reattivo non ha stato, ovvero a fronte dello stesso ***stato esterno*** = situazione del mondo + agente, decide sempre la stessa azione, indipendentemente dalla storia precedente. Un agente reattivo con stato possiede anche uno stato interno, che dipende dalla storia passata; l'azione decisa non dipende solo dallo stato esterno, ma anche da quello interno. Lo schema di base per tali agenti è il seguente:

```
type(open:stato_interno).
%  stato_interno: stato interno dell'agente, dipende alla storia precedente e influisce sul comportamento
type(open:obiettivo).
%   l'obiettivo generale dell'agente
type(open:azione).
%  le azioni dell'agente, che cambiano lo stato del mondo+agente

pred(agente(stato_interno, obiettivo)).
%  agente(S, Goal) :  genera ed esegue la sequenza di azioni dell'agente a partire dallo stato_interno S e dallo stato
%  del mondo+agente, in base alla implementazione dei predicati aperti decidi_azione ed esegui
%  MODO (+,+)
open_pred(fine(stato_interno, obiettivo)).
%  fine(S, G) :  l'obiettivo G è raggiunto (e l'agente termina)
%  MODO  (+,+) semidet
open_pred(decidi_azione(stato_interno, obiettivo, azione)).
%  decidi_azione(S, G, A) : nello stato_interno S e con goal G l'agente decide di fare l'azione A
%  MODO:   (+,+,-) det (nondet per agenti non deterministici)
open_pred(esegui(azione, stato_interno, stato_interno)).
% esegui(A, S1,S2) : l'agente esegue A e passa da stato_interno S1 a stato_interno S2
% MODO  (+,+,-) det  (in caso di azioni deterministiche)

agente(Stato, Obiettivo) :-
        fine(Stato, Obiettivo),!
        ;
        decidi_azione(Stato, Obiettivo, Azione),!,
        esegui(Azione, Stato, NuovoStato),
        agente(NuovoStato, Obiettivo).
```

Una implementazione di un agente reattivo che istanzia questo schema è data dall'agente "**Raccoglitore**" dato nel file **raccoglitore.pl.** "Raccoglitore" si muove in un ambiente con ostacoli, va a raccogliere un oggetto in una posizione e lo trasporta in un'altra posizione. Ha 3 stati interni, *presa*, *trasporto* e *fine*. Parte nello stato *presa* e vi rimane fino a quando raggiunge la posizione dell'oggetto da raccogliere; passa dallo stato *presa* allo stato *trasporto* quando raccoglie l'oggetto e inizia a trasportarlo; rimane nello stato *trasporto* fino a quando raggiunge la posizione in cui depositarlo; passa dallo stato *trasporto* allo stato *fine* quando ha raggiunto la posizione in cui depositare l'oggetto e lo deposita. Si veda il codice, che è documentato, nel file **raccoglitore.pl.** Sia nello stato *presa*, sia nello stato *trasporto*, l'agente raggiunge la posizione in cui prendere o depositare l'oggetto applicando la strategia di *aggiramento antiorario* degli ostacoli vista per l'agente puramente reattivo in **aggiratore.pl**. Di conseguenza ci aspettiamo ambienti privi di ostacoli concavi.

1.3. Agente Pianificatore Puro.

Si tratta di un agente che deve raggiungere un obiettivo. Calcola un piano per raggiungere l'obiettivo e poi esegue il piano calcolato. Se non esiste nessun piano fallisce.
Il piano può essere calcolato applicando direttamente un algoritmo di ricerca in uno spazio di ricerca. *I **nodi di pianificazione*** sono i nodi del grafo che costituisce lo spazio di ricerca e rappresentano i possibili *stati esterni* (stati del sistema agente + mondo). Nel grafo di ricerca, due nodi di pianificazione S1, S2 sono connessi da un arco se e solo se vi è una azione eseguibile dall'agente che fa passare dallo stato esterno rappresentato da S1 a quello appresentato da S2. *Il costo dell'arco (S1,S2) è il costo della azione corrispondente nel mondo esterno.* Di conseguenza, un percorso nel grafo di ricerca corrisponde a una sequenza di stati effettivamente percorribile dall'agente nel mondo esterno attraverso una corrispondente sequenza di azioni, detta *piano*. Eseguendo il piano calcolato, l'agente raggiunge l'obiettivo.

La scelta della strategia di ricerca da applicare nel calcolo del piano è centrale. Se si cerca un piano ottimale, di norma converrà A* con potatura chiusi e con opportune euristiche. Nel caso di esplosione dello spazio di memoria, si userà IDA*.

Un esempio di agente pianificatore puro che usa direttamente la ricerca in un grafo è il "PathFinder" (file **path_finder.pl**), che si muove in un ambiente con ostacoli e cerca di raggiungere una posizione goal calcolando prima il percorso (o piano) e seguendo poi il percorso calcolato.
L'agente è caratterizzato dai seguenti tipi e predicati

```
type([va(point,point)]:action).
   % va(P1, P2): azione di spostamento di un quadretto da P1 a P2, in una direzione verticale, orizzontale o diagonale.
   % La posizione P2 deve essere "navigabile", cioè libera da ostacoli.
pred(in(point)).
   % in(P):  l'agente si trova nel punto P,  predicato di stato esterno (del mondo + agente)
   :- dynamic(in/1).
pred(esegui(va(point,point))).
   % esegui(va(P1,P2)) :  possibile se l'agente è in P1 e P2 è navigabile; sposta l'agente in P2 con retract e assert
   % MODO (+) det
pred(start(any,point,point)).
   % start(W, P,G) : calcola un cammino e lo segue, per andare da P a G nel mondo caricato W
   % MODO (++,++,++) semidet
pred(segui_cammino(list(point))).
  %  segui_cammino(Path) :  simula l'agente che segue il cammino Path pianificato
  %  MODO (++) det
```

Per l'implementazione e i dettagli si veda il codice in **path_finder.pl**. Qui presentiamo solo il predicato di avvio start e l'uso della library(search).

```
start(W, P, G) :-
        world(W,_,_),!,
    % pongo W come mondo corrente e lo disegno
        retractall(current_world(_)), assert(current_world(W)),
        draw_loaded_world(W,20),

    %  pongo l'agente nello stato  iniziale, cioè in posizione P, e lo disegno
        retractall(in(_)), assert(in(P)),
        draw_fig(W, circ(15,[col(green)]), P),
        draw_fig(W, circ(15,[col(green)]), G),

        %  calcolo il piano usando la libreria search
        get_path(P, G, Path),
        step(['Calcolato un piano per andare da ', P, ' a ', G,  ' \n']),
        % e lo eseguo
        segui_cammino(Path).
```

1.3.1. Uso della libreria search. In get_path uso la libreria search, come segue.
Innanzitutto definisco i nodi e gli archi del grafo di ricerca. Identifico con i punti navigabili da parte dell'agente, dal momento che in ogni stato possibile del mondo esterno si trova in uno di essi. Un arco connette due nodi P1, P2 se sono adiacenti, cioè se l'agente trovandosi in P1 può eseguire l'azione va(P1,P2). Per l'adiacenza uso step(Dir,P1,P2) dato dalla geometria.

```
type([{point}]:nodo_pianificazione).
  %  i punti navigabili sono i nodi dello spazio di ricerca, sioè le posizioni possibili dell'agente
pred(arc(nodo_pianificazione, nodo_pianificazione,number)).
  %   arc(P1,P2,C)  :  l'azione va(P1,P2) è ipoteticamente possibile, cioè P1, P2 sono punti navigabili adiacenti un una
  %   delle 8 direzioni (orizzontali,verticali,diagonali). Il costo è la lunghezza in quella direzione (1 in orizzontale e
  %   verticale, 1.42 in diagonale)
arc(P1, P2, C) :-
    step(Dir,P1,P2),
    current_world(W), navigable(W,P1), navigable(W,P2),
    length_step(Dir,C).
```

Poi implemento i predicate aperti di path_search, usato per la ricerca di cammini.

```
path_search:neighbours(S, V) :-
        setof(X, C^arc(S,X,C), V), !  ;  V=[].
path_search:cost(S1,S2,C) :-
        arc(S1,S2,C).
path_search:heuristic(P,H) :-
        % come euristica di P prendo la distanza diagonal del nodo P dal nodo target G
        target(G), distance(diagonal,P,G,H).
```

Infine in get_path uso solve fornito dalla libreria search. In una chiamata solve(StartPred, GoalPred, Solution) con MODO (+,+,--) nondet, si ha:

- StartPred è un predicato unario tale che call(StartPred, X) fornisce uno stato iniziale X. Un modo per dare come stato iniziale un dato nodo P è porre StartPred=start_node, dove

start_node/1 è dinamico, ed asserire start_node(P). Così, call(start_node, X) chiama start_node(X) che unifica X con lo stato P.

- GoalPred è un predicato unario tale che call(GoalPred, X) riconosce uno stato goal X. Anche qui per passare come nodo goal un dato nodo target G, possiamo usare un predicato dinamico unario.
- Solution viene restituito nella forma sol(G, Path, Cost), dove G è un nodo goal, Path è un cammino dallo start a G e Cost è il costo del cammino.

Il codice diventa il seguente. *Si osservi l'uso in solve dei predicati qualificati path_finder:start_node e path_finder:target, dove si premette al predicato il modulo nel quale è definito, per far in modo che la call cerchi il predicato nel modulo giusto.*

```
get_path(P,G,Path) :-
        retractall(start_node(_)), assert(start_node(P)),  % fisso il predicato di start
        retractall(target(_)), assert(target(G)), % fisso il predicato di goal
        solve(path_finder:start_node, path_finder:target, sol(_,Path,_Cost)).
```

1.3.1.1. Uso di strategie e statistiche con la libreria search.

Si lanci il pathfinder e si esegua:

?- load_world(r1).

true.

?- start(r1, point(14,14), point(2,1)).

Si attende e la risposta non arriva. Il programma non è in loop, ma la ricerca richiede troppo tempo. Meglio bloccare l'esecuzione.

Il fatto è che la strategia di default è astar ***senza potatura cicli*** e con euristica 0.

Si scelga la strategia astar ***con potatura*** e si attivino le statistiche con **ds_on**:

```
?- set_strategy(astar).
true.
?- set_strategy(ps(closed)).
true.
?- ds_on.
true.
?- start(r1, point(14,14), point(2,1)).
Livello 0; nuovo livello / RET :
Livello invariato
nodi espansi: 19185; iterazioni: 18813; profondita' soluzione: 18; costo: 20.36
b=1.7295866418156207
Calcolato un piano per andare da point(14,14) (verde) a point(2,1) (giallo)
|:
```

Come si vede, con potatura si passa da un tempo non accettabile ad un tempo accettabile e fattore di branching effettivo 1.73. Ora si scelga la distanza "diagonal" e si lanci la stessa esecuzione.

```
?- set_distance(diagonal).
true.
?- start(r1, point(14,14), point(2,1)).
Livello 0; nuovo livello / RET :
Livello invariato
nodi espansi: 198; iterazioni: 86; profondita' soluzione: 18; costo: 20.36
b=1.3415056637348357
Calcolato un piano per andare da point(14,14) (verde) a point(2,1) (giallo)
|:
```

Si ottiene un fattore di branching effettivo di 1.34, molto meglio di 1.73. Si vede anche che i nodi espansi passano da 19185 a 198. Si consiglia di sperimentare e confrontare le statistiche con le varie euristiche con start(r1, point(24,18), point(2,1)).

Con ds_on non si attivano solo le statistiche, ma anche la possibilità di debugging visualizzando i passi di espansione dei nodi ed eventualmente la frontiera. Si scelga ad esempio nuovo livello di visualizzazione 2 e si chieda di mostrare la frontiera con +f:

```
?- start(r1, point(24,18), point(2,1)).
Livello 0; nuovo livello / RET : 2
Ok, nuovo livello = 2
Step 1:0 s-kip/a-bort/t-race/n-otrace/l-ivello/+f-rontiera/-f-rontiera/ RET: +f
Inizio con predicati start path_finder:start_node e di goal path_finder:target
Nodo iniziale:
  point(24,18), f:g:h = 27.80287754891569:0:27.80287754891569
vuoi il cammino? s_i/ RET :
```

Avendo scelto +f, d'ora in avanti sarà mostrata la frontiera, ad esempio

```
Step 2:1 s-kip/a-bort/t-race/n-otrace/l-ivello/+f-rontiera/-f-rontiera/ RET:
Nuovo nodo da esplorare:
  point(23,17), f:g:h = 27.82075756488817:1.42:26.40075756488817
vuoi il cammino? s_i/ RET :
Nuova frontiera:
Frontiera fr(no_bound,...) con 6 nodi
[
  point(23,18), f:g:h = 28.018512172212592:1:27.018512172212592
  point(24,17), f:g:h = 28.202941017470888:1:27.202941017470888
  point(23,19), f:g:h = 29.07863337187866:1.42:27.65863337187866
  point(24,19), f:g:h = 29.42534080710379:1:28.42534080710379
  point(25,17), f:g:h = 29.437851452243798:1.42:28.0178514522438
  point(25,19), f:g:h = 30.62616373302047:1.42:29.206163733020468
]
```

Concludiamo elencando le strategie attualmente implementate, visualizzabili con il comando:

```
 ?- search:strategy_request(R).
R = 'Frontiera':depth_first ;
R = 'Frontiera':breadth_first ;
```

```
R = 'Frontiera':best_first ;
R = 'Frontiera':lowest_cost ;
R = 'Frontiera':astar ;
R = 'Pruning':ps(cycles) ;
R = 'Pruning':ps(closed) ;
R = 'Pruning':ps(no_cut) ;
R = 'Bounded search':bs(no_bound) ;
R = 'Bounded search':bs(depth, '<upper_bound>') ;
R = 'Bounded search':bs(cost, '<upper_bound>') ;
R = 'Bounded search':bs(f, '<upper_bound>') ;
R = 'Bounded search':bs(priority, '<upper_bound>') ;
R = 'ID':id(depth) ;
R = 'ID':id(cost) ;
R = 'ID':id(f) ;
R = 'ID':id(priority) ;
R = 'ID limitato':id(depth, '<upper_bound>') ;
R = 'ID limitato':id(cost, '<upper_bound>') ;
R = 'ID limitato':id(f, '<upper_bound>') ;
R = 'ID limitato':id(priority, '<upper_bound>')
```

Frontiera: riguarda le strategie di gestione della frontier.
Pruning: riguarda le strategie di potatura
Bounded search: di solito si lavora senza limiti allo spazio di ricerca (no_bound); però in alcuni casi conviene stabilire un limite superiore (a depth, oppure cost, …) oltre il quale l'espansione dei nodi viene bloccata.
ID: iterative deepening su depth, oppure cost, … senza limiti sullo spazio di ricerca
ID limitato: oltre a ID si pone un limite allo spazio di ricerca.

1.3.1.2. Uso di forward_planner e strips_planner.

Invece di usare direttamente la ricerca nel grafo di ricerca che rappresenta lo spazio degli stati dell'agente, si possono usare forward_planner e strips_planner. Sono due pianificatori in avanti che fanno uso di A* con potatura chiusi come strategia di ricerca di default. Con questi due pianificatori l'uso di A* rimane nascosto. Vediamo l'uso di forward_planner nel caso dell'agente PathFinder e di strips_planner nel caso dell'agente Raccoglitore con pianificazione.

Per usare forward-planner bisogna definire:

a) Tipi: il tipo **planning_state** i cui elementi rappresentano gli stati esterni (di mondo+agente) a livello di calcolo del piano; il tipo **action** che rappresenta le azioni possibili.
b) Il predicato **do_action**(A, St1, St2, Cost) che, data un'azione A e un planning state St1, calcola il planning state St2 ottenuto eseguendo a in St1 (più precisamente, le azioni sono quelle dell'agente e l'arco (St1,St2,Cost) rappresenta a livello di pianificazione una transizione dallo stato esterno rappresentato da St1 a quello rappresentato da St2, che sarebbe ottenuta dall'agente eseguendo A nel mondo reale, con costo Cost). I modi sono: MODI (++,++,--,--) nondet, (--,++,++,--) det
c) L'euristica **h**(+S, -H) che calcola il valore euristico H = f(S).

forward_planner esporta il predicato plan(+StartPred, +GoalPred, -Path, -Plan, -Cost), dove call(StartPred, S) deve fornire uno stato iniziale S, call(GoalPred, G) deve riconoscere i nodi goal

G, Path è il cammino dal nodo iniziale al nodo goal, Plan è la corrispondente sequenza di azioni (il piano) e Cost è il costo.
Ad esempio la seguente parte del codice in **path_finder_forward_planner.pl** è:

```
type([in(point)]:planning_state).
   % in(P) rappresenta lo stato esterno  in(P), in cui l'agente si trova nel punto P
type([va(point,point)]:action).
   % va(P1,P2) : l'agente si sposta da P1 a P2
forward_planner:do_action(va(P1,P2), in(P1), in(P2), Cost) :-
      step(Dir,P1,P2),
     current_world(W),
     navigable(W,P1),
     navigable(W,P2),
     length_step(Dir,Cost).
forward_planner:h(in(P),H) :-
     target(in(G)),
     distance(diagonal,P,G,H).
% per ottenere il piano uso plan/5 esportato da forward_planner:
:- dynamic(start_node/1).
:- dynamic(target/1).
get_plan(P,G,Plan, Cost) :-
          retractall(start_node(_)), assert(start_node(in(P))),  % fisso il predicato di start
          retractall(target(_)), assert(target(in(G))), % fisso il predicato di goal
          plan(path_finder:start_node, path_finder:target, _,Plan,Cost).
```

Per usare strips_planner bisogna definire:

- Tipi. Il tipi fluent, action richiesti da strips (si ricordi che gli stati di strips sono liste ordinate di fluenti).
- Predicati aperti. Il predicato add_del(Act, Add, Del, Cost) (con il significato di Strips: eseguendo Act, Add diventano veri e Del diventano falsi). L'euristica h(+StripsState, -H) che fornisce il valore euritico degli stati di strips.

strips_planner esporta il predicato plan(+StartPred, +GoalPred, -Path, -Plan, -Cost), dove call(StartPred, S) deve fornire uno stato iniziale S, call(GoalPred, G) deve riconoscere i nodi goal G, Path è il cammino dal nodo iniziale al nodo goal, Plan è la corrispondente sequenza di azioni (il piano) e Cost è il costo.

Ad esempio, parte del codice dell'agente Raccoglitore con pianificazione, che deve raccogliere degli oggetti in varie posizioni e depositarli in un deposito è riportato qui sotto.

```
type([in(point), deve_prendere(punto), deposito(punto), fine]:fluent).
   %  in(P):  l'agente si trova in P;     deve_prendere(X): l'agente deve prendere un oggetto in X
   %  deposito(Q):  deve depositare gli oggetti in Q;    fine:  ha finito
type([va(point,point), raccoglie, deposita]:azione).
   %  va(P1,P2) : avanza da P1 a P2;   raccoglie:  raccoglie l'oggetto;   deposita: deposita gli oggetti

% implemento i predicate aperti
strips_planner:add_del(va(P1,P2), St, [in(P2)], [in(P1)], Cost) :-
          member(in(P1), St),  not(member(deve_prendere(P1),St)),
          step(Dir,P1,P2), % P2 raggiungibile in un passo in una qualche direzione Dir
```

```
        current_world(W), navigable(W,P2),
        length_step(Dir,Cost).  %distanza nella direzione di spostamento
strips_planner:add_del(raccoglie, St, [], [deve_prendere(P)], 0.5) :-
        % raccoglie in P e quindi poi non vale più deve_prendere(P)
        member(in(P), St), member(deve_prendere(P), St).
strips_planner:add_del(deposita, St, [fine], [deposito(P)], 0.5) :-
        % finisce quando deposita e non c'è più nulla da prendere
        member(in(P), St), member(deposito(P), St), not(member(deve_prendere(_), St)).

strips_planner:h(St, 0) :-
        member(fine,St), !.
strips_planner:h(St, H) :-
        %  conto il numero di oggetti da raccogliere; euristica da migliorare
        setof(X, member(deve_prendere(X), St), Pos) ->  length(Pos,H)
        ;   H = 0.
```

Qui abbiamo dato solo l'implementazione dei predicati aperti. Si veda il codice completo in **raccoglitore_strips.pl**.

1.4. Agente con decisioni e pianificazione.

Questo tipo di agenti prende man mano delle decisioni. Una decisione può richiedere una semplice esecuzione di una azione, oppure il calcolo di un piano che viene poi mandato in esecuzione. Lo schema generale di questo tipo di agenti è il seguente.

```
type(open:stato_interno).
  %  stato_interno:   lo stato interno dell'agente, che dipende  dalla storia precedente e influisce sul comportamento
type(open:obiettivo).
  %   l'obiettivo generale dell'agente
type(open:decisione).
  %  le possibili decisioni dell'agente
type(open:azione).
  %  le azioni dell'agente, che cambiano lo stato del mondo+agente

pred(agente(stato_interno, obiettivo)).
  %  agente(S, Goal) :  schema di comportamento di un agente con decisioni e pianificazione
  %  Genera ed esegue la sequenza di decisioni e azioni dell'agente a partire dallo stato_interno S e dallo stato
  %  iniziale del mondo+agente, in base alla implementazione dei predicati aperti
  %  MODO (+,+) genera e visualizza un comportamento
pred(esegui(list(azione), stato_interno, stato_interno)).
  %   esegui(P, S1,S2) : esegue il piano P, passando dallo stato S1 allo  stato S2.     MODO  (+,+,-) det
open_pred(fine(stato_interno, obiettivo)).
  %   fine(S, G) : l'obiettivo G è raggiunto (e l'agente termina).    MODO (+,+) semidet
open_pred(decidi(stato_interno, obiettivo, decisione)).
  %  decidi(S, G, D) : nello stato_interno S e con goal G l'agente prende la decisione D.   MODO:   (+,+,-) det
open_pred(pianifica(stato_interno, decisione, list(azione))).
  %  pianifica(S, D, P) : nello stato_interno S l'agente calcola il piano per portare a termine la decisione D
  %  MODO: (+,+,-)  det
open_pred(esegui(azione, stato_interno, stato_interno)).
  % esegui(A, S1,S2) : l'agente esegue A e passa da stato_interno S1 a stato_interno S2.   MODO  (+,+,-) det
```

```
agente(Stato, Obiettivo) :-
        fine(Stato, Obiettivo),!
        ;
        decidi(Stato, Obiettivo, Decisione),
        pianifica(Stato, Decisione, Piano),!,
        esegui(Piano, Stato, NuovoStato),
        agente(NuovoStato, Obiettivo).

esegui([], Stato, Stato).
esegui([A|Piano], Stato1, Stato2) :-
        esegui_azione(A, Stato1, Stato),!,
        esegui(Piano, Stato, Stato2).
```

Un agente di questo tipo è l'agente Delivery (file **delivery.pl**). Riportiamo solo la parte di codice utile a comprendere come implementare i predicati aperti. I tipi rilevanti sono action, fluent, stato_interno, decisione, obiettivo:

```
type([consegna(luogo,integer), davanti(porta)]:fluent).
  %  consegna(L,P1): deve consegnare un plico di P1 missive in L;
  %  davanti(L) : si trova davanti a una porta di ingresso a una stanza
type([preleva, va(luogo,luogo), consegna(porta)]:action).
  %   preleva : si trova davanti a una casella e preleva la posta;   va(L)  :  va davanti a L
  %   consegna : ha un plico da consegnare nella stanza davanti a cui si trova
type([carica_0,      %l'agente all'inizio deve caricare da c0
      caricata_0,    % l'agente ha caricato c0
      carica_1,      % l'agente ha caricatoc0 e deve caricare da c1
      distribuzione %  l'agente ha ultimato il carico e inizia a distribuire o sta distribuendo
      ]:stato_interno).
type([decido(action), distribuisco]:decisione).
   %  decido(A):  ho dciso di eseguire l'azione A;  distribuisco:   ho deciso di distribuire la posta caricata
type([posta_consegnata]:obiettivo).
  %  c'è un solo obiettivo, aver cosegnato tutta la posta
```

L'implementazione dei predicati aperti è la seguente. Si osservi che esegui_azione può cambiare lo stato interno.

```
  fine(_,consegne_eseguite) :-
        % ha finito quando è tornato a c0 e non c'è più posta da prelevare o consegnare
        davanti(c0), not(posta(_,_,_)), not(consegna(_,_)),  simula(fine).
  decidi(carica_0, _, decido(preleva)).
  decidi(carica_1,_, decido(preleva)).
   % se deve caricare da una casella, decide di prelevare dalla casella
  decidi(caricata_0,_,decido(va(c0,c1))).
    % prelevata la posta da c0, decide di andare alla casella c1
  decidi(distribuzione, _, decido(va(c1,c0))) :-
        % è davanti a c1 ma non ha consegne decide di tornare a casa
        davanti(c1), not(consegna(_,_)).
  decidi(distribuzione, _, distribuisco):-  consegna(_,_).
     %  una volta carico, decide di distribuire la posta
```

```
pianifica(_, decido(A), [A]).
    %  il piano per eseguire un'azione decisa A è [A]
pianifica(_, distribuisco, Piano) :-
        % il robot si trova davanti a c1 e ha caricato tutte le consegne da eseguire,
        %  rappresentate dal predicato dinamico consegna(L,K)
        setof(consegna(L,K), consegna(L,K), Consegne),
        % Consegne contiene le consegne da fare in questo momento, calcolo un piano per effettuarle tutte:
        pianifica_con_strips(Consegne, Piano),
        next_step(['Calcolato Piano ', Piano]).
esegui_azione(preleva, carica_0, caricata_0) :-
        davanti(c0), posta(c0,_,_),
        forall(posta(c0,L,K), carica_posta(c0,L,K)),!,
        (   consegna(_,_) -> next_step(['Caricata la posta da casella 0']) ;   next_step(['Non c''e'' posta in casella 0'])).
esegui_azione(va(c0,c1), caricata_0, carica_1) :-
        retract(davanti(c0)), assert(davanti(c1)),
        simula(va(c0,c1)), next_step(['Raggiunta casella 1']).
esegui_azione(preleva, carica_1, distribuzione) :-
        forall(posta(c1,L,K), carica_posta(c1,L,K)), consegna(_,_),!,
        next_step(['Caricata la posta da casella 1']).
esegui_azione(torna_a_casa, distribuzione, fine)  :-
        not(consegna(_,_)), !,
        retract(davanti(c1)), assert(davanti(c0)),
        simula(va(c1,c0)), next_step(['Non c''e'' posta, torno a casa']).
esegui_azione(va(L1,L2), distribuzione, distribuzione) :-
        retract(davanti(L1)), assert(davanti(L2)),
        simula(va(L1,L2)), next_step(['Raggiunta porta ', L2]).
esegui_azione(consegna, distribuzione, distribuzione) :-
        davanti(P), porta(P,L), retract(consegna(L,K)),
        simula(consegna(L)), next_step(['Consegnati ', K,' plichi in ', L]).
pianifica_con_strips(Consegne, Piano) :-
        %  uso plan/5 di strips_planner con predicati Start= start(Consegne) e Goal = goal, dove:
        % call(start(Consegne), S0)  fornisce lo stato iniziale,  call(goal, G)  riconosce il goal G
        plan(start(Consegne), goal, _, Piano, _).
```

1.5. Cenni su pianificazione gerarchica.

Se osserviamo il codice dell'agente delivery, osserviamo che il piano viene calcolato sul grafo i cui nodi le porte delle stanze e i cui archi sono le connessioni fra porte adiacenti. In particolare, l'azione va(L1,L2) va da una stanza ad un'altra:

```
esegui_azione(va(L1,L2), distribuzione, distribuzione) :-
        retract(davanti(L1)), assert(davanti(L2)),
        simula(va(L1,L2)),
        next_step(['Raggiunta porta ', L2]).
```

A livello di calcolo del piano l'azione viene implementata con "retract(davanti(L1)), assert(davanti(L2))", cioè è come se si avesse una "macro-mossa" che da davanti(L1) "salta direttamente" a davanti(L2). Nel mondo simulato l'agente non salta ma procede di quadretto in quadretto. Diremo che va(L1,L2) è una ***macro-azione***, cioè un'azione non direttamente eseguibile dall'agente, da realizzare attraverso una successione di ***azioni base***, cioè azioni effettivamente

eseguibili dall'agente. Nella implementazione dell'agente delivery, la macro-azione va(L1,L2) è "eseguita" passo-passo da un agente reattivo:

```
simula(va(St1,St2)) :-
        posto_davanti(St2, G),
        %  simulo l'esecuzione quadretto per quadretto, implementando un agente reattivo
        reattivo(G).
```

dove reattivo(G) ogni volta sceglie il quadretto adiacente più vicino a G e si sposta in esso.

Capita spesso di pianificare considerando macro-azioni la cui esecuzione richiede la esecuzione di una sequenza di azioni più elementari. Se questa sequenza viene trovata mediante pianificazione, si parla di ***pianificazione gerarchica***. L'uso della pianificazione gerarchica può risultare lo strumento da usare per rendere trattabili problemi altrimenti intrattabili, spezzando la ricerca di un piano molto lungo di sole azioni elementari in una gerarchia di piani più brevi. Ad esempio possiamo spezzare un piano lungo 100 in un piano di 10 macro-azioni eseguibili con piani lunghi 10; mentre con fattore di branching b la ricerca del piano di lunghezza 100 ha complessità dell'ordine di b^{100}, quelle dei piani di lunghezza 10 hanno complessità dell'ordine di b^{10} (anche assumendo che per ciascuno dei b^{10} macro-tentativi occorrano b^{10} tentativi elementari, in tutto avrei b^{20} tentativi << b^{100}). La pianificazione gerarchica definisce una gerarchia di azioni e uno schema di decomposizione di macro-azioni in azioni più elementari, governando in tal modo l'esplosione combinatoria. Possiamo vedere sperimentalmente la differenza con il seguente esempio.

Usando raccoglitore_strips.pl (che non è gerarchico) abbiamo:

```
25 ?- set_strategy(astar).
true.
26 ?- set_strategy(ps(closed)).
true.
27 ?- load_world(cw(2)).
true.
28 ?-  time(raccolta(cw(2), point(2,2),[point(10,19), point(4,12)],point(3,2),Pl,C)).
% 70,014,430 inferences, 48.984 CPU in 49.405 seconds (99% CPU, 1429323 Lips)
Pl = [va(point(2, 2), point(3, 3)), va(point(3, 3), point(4, 4)), va(point(4, 4), ….],
C = 44.90000000000002
```

Come si vede, il calcolo del piano ha richiesto 49 secondi. Usando raccoglitore_gerarchico abbiamo:

```
9 ?- set_strategy(ps(closed)).
true.
10 ?- load_world(cw(2)).
true.
11 ?- raccolta(cw(2), point(2,2),[point(10,19), point(4,12)],point(3,2)).
Avvio con strategia astar e potatura chiusi
|:
Calcolato il piano
[va_da_a(point(2,2),point(4,12)),raccoglie,va_da_a(point(4,12),point(10,19)),raccoglie,
va_da_a(point(10,19),point(3,2)),deposita]  con costo 44.9
```

Qui la risposta è immediata e abbiamo ottenuto un piano lungo 8 con macro-azione va_da_a. Il pianificatore usato è un pianificatore a 2 livelli privo di schemi di decomposizione. Il codice è dato in **two_level_planner.pl**. Essendo basato su strips_planner, usa i tipi di strips, in particolare fluent, action, strips_state. Usa un predicato dinamico level, per gestire il livello corrente di pianificazione: level(0) per i piani che possono contenere macro-azioni; level(1) per i piani di esecuzione delle

macro-azioni mediante azioni base. Implementa i predicati aperti di strips_planner (add_del ed euristica h) come segue:

```
strips_planner:add_del(Act, St, Add, Del, Cost) :-
        level(K),   add_del(K,Act, St, Add, Del, Cost).
strips_planner:h(St, H) :-
        level(K),  h(K,St,H).
```

I predicati add_del(K,Act, St, Add, Del, Cost) e h(K,St,H) sono aperti e vanno usati per definire le azioni di livello K e l'euristica da usare nella ricerca di un piano di livello K.

I predicati esportati dal planner sono:

```
pred(get_plan(list(any), list(action), number)).
%   get_plan(Input, Plan, Cost) :  Plan piano con macro-azioni di costo Cost per ottenere l'obiettivo dati in Input

pred(get_action_plan(strips_state, action, list(action), number)).
%   get_action_plan(St, Act, Plan, Cost) : Plan  piano che esegue la macro-azione Act  partendo dallo strips_state St
```

Il loro codice è:

```
%Livello 0: cerco un piano con azioni di livello 0
get_plan(ParameterList, Plan, Cost) :-
        starting_state(ParameterList, St0), % stato di starting livello 0
        set_level(0),
        retractall(action_plan(_,_,_,_)),
        plan( two_level_planner:start_with(St0),
              two_level_planner:goal_state(ParameterList), _, Plan, Cost).

%  Livello 1:  per le macro-azioni cerco un piano di esecuzione a meno se non già calcolato
get_action_plan(St, Action, Plan, Cost) :-
       action_plan(St, Action, Plan, Cost),!. %  piano già memorizzato
get_action_plan(St, Action, Plan, Cost) :-
       set_level(1),  % entro in livello 1
       action_starting_state(St, Action, St1), %starting state livello 1
       plan(   two_level_planner:start_with(St1),
               two_level_planner:action_final_state(St,Action), _, Plan, Cost),!,
        assert(action_plan(St, Action, Plan, Cost)),  % memorizzo il piano
        set_level(0).  % esco e rimetto il livello a 0
```

Il predicato plan esportato da strips_planner è usato da two_level_planner sia a livello 0, sia a livello 1. Mediante set_level si gestisce il predicato dinamico level(K); con K=0 strips_planner considererà le azioni definte da add_del(0,…) a livello 0, con K = 1 quelle a livello 1.

A livello 0, il predicato aperto starting_state(ParameterList, S0) deve fornire gli strips_states iniziali S0 del piano con macro-azioni, mentre goal_state(ParameterList, G) deve riconoscere quelli finali.

A livello 1, il predicato aperto action_starting_state(+St, +Action, -St1) deve fornire gli strips_states iniziali St1 del piano di esecuzione della macro-azione Action a partire dallo strips_state di livello 0 St, mentre action_final_state(+St, +Action, +G) deve riconoscere quelli finali G.

Gli strips_states di livello 0 e quelli di livello 1 non devono necessariamente essere gli stessi; *è però importante che le azioni base di livello 1 siano eseguibili nel mondo+agente*, dal momento che la esecuzione di una macro-azione di livello 0 viene "esplosa" in una sequenza di azioni di livello 1, ottenuta con get_action_plan. Il file **two_level_planner.pl** contiene anche una bozza di codice per la esecuzione di piani di 2 livelli che mostra come ciò avviene; il predicato di esecuzione di un piano

è esegui, quello di esecuzione delle azioni è esegui_azione, che per le macro-azioni di livello 0 calcola ed esegue un piano di livello 1 e per le azioni base invoca il predicato aperto esegui_azione_base. Il codice è il seguente:

```
esegui([]).
esegui([A|Piano]) :-
        esegui_azione(A),!,  esegui(Piano).

esegui_azione(A) :-  macro_azione(A),!,
        action_starting_state(A, St0),
        % se A è una macro-azione, calcolo lo strips_state St0 che rappresenta lo stato attuale di
        % mondo+agent e il PianoAzione per eseguire A a partire dallo stato attuale
        get_action_plan(St0,A,PianoAzione,_),
        %  e poi eseguo PianoAzione
        esegui(PianoAzione).

esegui_azione(A) :-
        %  se A non è una macro-azione è una azione base
        esegui_azione_base(A).
```

I predicati aperti per l'esecuzione di un piano a 2 livelli nel mondo+agente sono:

macro_azione(Act), che decide se Act è una macro-azione o meno.

action_starting_state(A, St0), che calcola lo strips_state St0 usato nella pianificazione di livello 1 e corrispondente allo stato attuale; viene usato per ottenere il PianoAzione da eseguire nello stato attuale tramite get_action_plan. Nella versione data qui tale piano è stato memorizzato in fase di pianificazione mediante il predicato dinamico action_plan e non viene ricalcolato; se per problemi di memoria si rinuncia a memorizzare i piani di livello 1, get_action_plan lo ricalcolerà.

esegui_azione_base(A) che esegue le azioni base di livello 0 (non le macro-azioni) e le azioni di livello 1 che, in una pianificazione a 2 livelli, sono tutte azioni base e non macro-azioni.

Vediamo ora come usare two_level_planner, implementandone i predicati aperti. Ciò è fatto nell'esempio del raccoglitore gerarchico (in **raccoglitore_gerarchico.pl**).

Innanzitutto si implementano i predicati aperti add_del e h in cui il primo parametro è il livello 0 o 1. Abbiamo:

```
%    implementazione AZIONI DI LIVELLO 0
two_level_planner:add_del(0, deposita, St, [fine], [deposito(P)], 0.5) :-
        % deposita viene eseguita se l'agente si trova in P,  deve depositare in P e non ha più nulla da
        %  prendere; è una azione base di livello 0 , non una macro-azione;  si rende vero fine
        not(member(deve_prendere(_), St)),
        member(deposito(P), St), member(in(P), St).

two_level_planner:add_del(0,va_da_a(P1,P2), St, [in(P2)], [in(P1)], Cost) :-
        %  se l'agente non ha più nulla da prendere,si trova in P1 e deve depositare in P2 \= P1 allora  va_da_a(P1,P2) ;
        %  P1, P2 sono posizioni distanti, per cui si tratta di una macro-azione; nel piano di livello 0 si ha un "salto" da P1
        % direttmente a P2 (in(P1) è cancellato e in(P2) diventa vero).
        not(member(deve_prendere(_), St)),  member(deposito(P2), St),
        member(in(P1), St), P1 \= P2,
        %  per valutare il costo Cost della macro-azione usato da astar calcolo il piano  di livello 1
        % per eseguire va_da_a(P1,P2), usando get_action_plan
        get_action_plan(St, va_da_a(P1,P2), _Plan, Cost).
```

……

Tralasciamo le altre azioni di livello 0, per le quali rimandiamo al codice; qui ci basta aver mostrato una azione base e una clausole per la macro-azione va_da_a(P1,P2), di livello 0. La macro-azione viene pianificata a livello 1 usando l'azione base va(P1,P2), che va da P1 a un quadretto adiacente P2:

```
%   implementazione  AZIONI DI LIVELLO 1
two_level_planner:add_del(1,va(P1,P2), St, [in(P2)], [in(P1)], Cost) :-
        member(in(P1), St),  step(Dir,P1,P2),
        % P2 raggiungibile in un passo e non contenente un ostacolo nel mondo corrente W dell'agente
        current_world(W),  not(content(W,P2,ostacolo)),
        length_step(Dir,Cost).  %lunghezza di un passo nella direzione di spostamento
```

Occorre poi chiudere il predicato aperto h a livello 0 e 1 (l'euristica usata nella ricerca dei piani di livello 0 e di livello 1):

```
%   implementazione EURISTICA DI LIVELLO 0
two_level_planner:h(0, St, 0) :-
        member(fine,St), !.
two_level_planner:h(0, St, H) :-
        %  conto il numero di oggetti da raccogliere; con la macro-azione va_da_a è un valore euristico sensato
        setof(X, member(deve_prendere(X), St), Pos) -> length(Pos,H)  ;   H = 0.

%   implementazione EURISTICA DI LIVELLO 1
two_level_planner:h(1, St, H) :-
        % a livello 1 di fatto si ha la pianificazione di un percorso (path-finding);  con movimenti
        % anche diagonali una euristica adatta è la distanza diagonale dalla posizione target
        member(target(P2), St), member(in(P1),St),
        distance(diagonal, P1,P2,H).
```

Il predicato get_plan per il calcolo dei piani di livello 0 usa come predicati di start e di goal nella ricerca del piano i predicati aperti starting_state e goal_state; la loro implementazione per il raccoglitore gerarchico è:

```
%   implementazione PREDICATI DI START E  GOAL LIVELLO 0
two_level_planner:starting_state([P1, PuntiRaccolta, P2],  [in(P1), deposito(P2)| FluentiRaccolta]) :
          setof(deve_prendere(X), member(X, PuntiRaccolta), FluentiRaccolta).
          %  [in(P1), deposito(P2) | FluentiRaccolta] sono i fluenti veri nello stato iniziale
two_level_planner:goal_state(_Param, State) :-
           % uno stato finale contiene il fluente fine, reso vero dall'azione "deposita"
           member(fine, State).
```

Il predicato get_action_plan per il calcolo dei piani di livello 1 usa come predicati di start e di goal nella ricerca del piano i predicati aperti action_starting_state e action_final_state; la loro implementazione per il raccoglitore gerarchico è:

```
%   implementazione PREDICATI DI START E GOAL LIVELLO 1
two_level_planner:action_starting_state(_St, va_da_a(P1,P2), [in(P1), target(P2)]).
%  nella pianificazione di va_da_a(P1,P2) mi basterebbe  in(P); target(P2) è usato nel calcolo dell'euristica

two_level_planner:action_final_state(_St, va_da_a(_,_), G) :-
          %  ho terminato il piano per va_da_a quando raggiungo il target
          member(target(P),G), member(in(P),G).
```

Infine, per eseguire il piano di livello 0 nel mondo+agente implemento i predicati aperti usati nella implementazione di esegui:

```
%  implementazione PREDICATI APERTI DI esegui
two_level_planner:macro_azione(va_da_a(_,_))   %  va_da_a è l'unica macro-azione

two_level_planner:esegui_azione_base(va(P1,P2)) :-
        retract(in(P1)), assert(in(P2)),
        simula(va(P1,P2)).

two_level_planner:esegui_azione_base(deposita) :- simula(deposita).

two_level_planner:esegui_azione_base(raccoglie) :-
        in(P), retract(deve_prendere(P)),
        simula(raccogli(P)).

two_level_planner:action_starting_state(va_da_a(P1,P2), [in(P1), target(P2)]).
%  l'esecuzione della macro-azione inizia nello stato in(P1); target(P2) è usato nel calcolo della euristica, data dalla
%  distanza diagonale fra la posizione corrente e il target P2
```

1.6. Agente con conoscenza incompleta e apprendimento.

Come ultimo esempio vediamo il caso di un agente che opera in un mondo che non conosce in modo completo. Costruiamo uno schema di agente un po' più complesso dell'agente con decisioni e pianificazione della sezione 1.4.

La differenza principale consiste nel fatto che quando l'agente prende una decisone o calcola il piano deve fare delle assunzioni sulla parte di mondo che non conosce. Può accadere che poi, nella esecuzione di un piano nel mondo reale, si venga a trovare in situazioni diverse da quelle che aveva previsto. In tal caso deve rivedere il piano e la decisione presa. Il modo più appropriato di trattare situazioni impreviste è l'uso delle eccezioni. In Prolog una eccezione viene sollevata con il predicato throw e viene catturata con il predicato catch (vedere il manuale).

Altra differenza, di minore importanza, è l'uso della pianificazione a 2 livelli.

Vediamo ora il codice, che si trova nel file **agente_ci.pl** (ci per "conoscenza incompleta"). I tipi principali sono i seguenti.

```
type(open:stato_agente). %  stato_agente:   lo stato interno dell'agente

type(open:obiettivo).    %gli obiettivi generali dell'agente

type(open:decisione_complessa). % le decisioni che richiedono un piano

type([failed(list(action), any), failed(list(action), any, list(action))]:exception).
  %  failed(Env, Cause) : interruzione azione, indica ambiente e cause
  %  failed(Env, Cause, PianoRestante) : interruzione piano, indica ambiente, cause e il piano che resta da eseguire.
  %  L'ambiente è uno stack di una o due azioni: Env = [Az] : eccezione sollevata da Az
  %  Env = [AzBase,Az] : eccezione sollevata da AzBase nel  corso della esecuzione di un piano per la macro-azione Az

type([esecuzione(action), piano(decisione_complessa)]:decisione).
  %  la decisione presa, in base a cui pianificare:
  %  "esecuzione(A)" ha un piano molto semplice, [A]
  %   "piano(D)" richiede il calcolo di un piano per portare a termine la decisione complessa D

type([{stato_agente},{exception}]:stato_interno).
  %  lo stato interno è uno stato "normale" dell'agente (stato_agente) o una situazione imprevista (exception) dovuta a
  % conoscenza  incompleta
```

Il codice è:

```
open_pred(fine(stato_interno, obiettivo)).
  %   fine(S, G) : l'obiettivo G è raggiunto e l'agente termina.    MODO (+,+) semidet

open_pred(decidi(stato_interno, obiettivo, decisione)).
  %  decidi(S, G, D) : nello stato_interno S e con goal G l'agente prende la decisione D. MODO:   (+,+,-) det

open_pred(macro_azione(action)).   % macro_azione(A) : A è una macro-azione.  Modo (+) semidet.

open_pred(esegui_azione_base(list(action),action, stato_interno, stato_interno)).
  % esegui_azione_base(Env, A, S1, S2) : l'agente esegue A e passa da stato_interno S1 a stato_interno S2; in caso di
  %  fallimento lancia una eccezione di fallimento; Env = [] per esecuzione a livello 0, Env=[M] per esecuzione a
  % livello 1 del piano per la macro-azione M.    MODO (+,+,+,-) det

open_pred(action_starting_state(action, strips_state)).
  %  action_starting_state(M, St) : M  è una macro-azione e St è lo strips_state iniziale di M corrispondente  allo stato
  %  attuale del mondo+agente
```

```
local_pred(esegui_azione(list(action),action, stato_interno, stato_interno)).
  % esegui_azione(Env, A, S1, S2) : l'agente esegue A e passa da stato_interno S1 a stato_interno S2; in caso di
  % fallimento solleva una eccezione di fallimento; Env = [] per esecuzione a livello 0, Env=[M] per esecuzione a
  % livello 1 del piano per la macro-azione M. MODO (+,+,+,-) det

local_pred(esegui(list(action), stato_interno, stato_interno)).
  % esegui(Piano, S1, S2) : l'agente esegue il Piano e passa da stato_interno S1 a stato_interno S2; in caso di fallimento
  % S2 è una exception. MODO (+,+,-) det

local_pred(pianifica(stato_interno, decisione, list(azione))).
  % pianifica(S, D, P) : nello stato_interno S l'agente calcola il piano per eseguire la decisione D. MODO: (+,+,-) det

pred(agente(stato_interno, obiettivo)).
  % agente(S, Goal) : schema di comportamento dell'agente. MODO (+,+)

agente(Stato, Obiettivo) :-
        fine(Stato, Obiettivo),!
        ;
        decidi(Stato, Obiettivo, Decisione),
        pianifica(Stato1, Decisione, Piano),!,
       catch( esegui([],Piano, Stato1, NuovoStato),  % in caso di eccezione il nuovo stato è l'eccezione stessa
              failed(EnvA, Causa, PianoA), NuovoStato = failed(EnvA, Causa, PianoA)),
        agente(NuovoStato, Obiettivo).

pianifica(_Stato, esecuzione(A), [A]).
pianifica(Stato,  piano(Decisione), Piano) :-  get_plan([Stato, Decisione], Piano, _Cost).

esegui(_Env, [], Stato, Stato).  %  Piano terminato con successo
esegui(Env,[A|Piano], Stato1, Stato2) :-  % esecuzione azione e prosecuzione
      catch( esegui_azione(Env, A, Stato1, Stato), failed(EnvA, Causa), throw(failed(EnvA, Causa, Piano))),
      % in caso di eccezione di fallimento, la rilancio includendo l'informazione sul Piano ancora da eseguire
      esegui(Env,Piano, Stato, Stato2).

esegui_azione(Env, A, Stato1, Stato2) :- macro_azione(A),!,
        action_starting_state(A, St0),
        get_action_plan(St0,A,PianoAzione,_),!,
        esegui([A|Env], PianoAzione, Stato1, Stato2).
esegui_azione(Env, A, Stato1, Stato2) :-
        esegui_azione_base(Env, A, Stato1, Stato2),
       (   Stato2=failed(Causa) ->  throw(failed([A|Env], Causa)) ; true).
       % lo stato di fallimento interrompe il processo di esecuzione
```

I predicati aperti fine e decidi hanno lo stesso ruolo e significato visti nel caso dello schema di agente della sezione 1.4. I predicati aperti macro_azione e action_starting_state hanno lo stesso ruolo e significato visto nell'esecuzione di un piano nel two_level_planner. Il predicato aperto esegui_azione_base cambia. Infatti può accadere che l'azione non risulti eseguibile a causa di una situazione del mondo imprevista; in tal caso, lo stato prossimo Stato2 diventa failed(Causa), dove Causa spiega le cause dell'impossibilità di esecuzione dell'azione. Se lo Stato2 è di fallimento, esegui_azione lancia un'eccezione contenente la causa e l'ambiente in cui si è verificata; l'eccezione viene catturata in esegui e rilanciata aggiungendo l'informazione sulla parte di piano ancora da eseguire. L'eccezione così arricchita viene catturata all'interno del ciclo di esecuzione di agente e passata come stato risultante dall'interruzione del piano. Al prossimo giro l'eccezione si troverà nello Stati di decidi(Stato, Obiettivo, Decisione) , che deciderà come trattarla.

Nel caso di conoscenza incompleta, la conoscenza del mondo può cambiare. Occorre dotare l'agente di un ***sistema di gestione della conoscenza***, che comprende i criteri con cui l'agente *fa assunzioni*

sulla parte di mondo che non conosce e le sue capacità di *apprendimento e di revisione della conoscenza* a fronte della sua esperienza presente e passata. Qui facciamo l'esempio di un agente che si muove in un mondo sconosciuto o conosciuto in parte. *Quando l'agente decide o pianifica lo fa in base alle conoscenze che ha e in base ad assunzioni sulla parte che non conosce*. Man mano che si muove, dalla posizione in cui si trova può osservare le posizioni circostanti e "*impara*" ricordando quanto ha osservato. Di conseguenza la conoscenza del mondo aumenta man mano che l'agente opera in esso. Uno schema di gestione della conoscenza in questo semplice caso è il seguente.

```
%  SCHEMA DI GESTIONE DELLA CONOSCENZA
type(open:osservabile). %  proprietà osservabile del mondo
type([{osservabile}, non(osservabile)]:osservazione).   %  una osservazione può essere positiva o negativa

open_pred(osserva(point, osservazione)).
  %  osserva(P, Oss)  : l'agente osserva che in P vale Oss.    MODO (++,++) det
open_pred(puo_assumere(point, osservazione)).
  % puo_assumere(P, Oss) : l'agente può assumere che in P valga Oss, a meno di conoscenze contrarie
  % MODO (+, ?) nondet.
pred(negazione(osservazione,osservazione)).
  %  NonOss è la negazione di Oss.  MODO (++,--) det
pred(sa(point, osservazione)).
  %  sa(P, Oss) :  l'agente sa che in P vale Oss.   MODO (++,--) nondet
  :- dynamic(sa/2).
pred(impara(point, osservazione)).
  %  impara(P, Oss) : non sa(P, Oss) e lo impara, cioè memorizza sa(P, Oss).  MODO (++,++) det
pred(sa_o_assume(point, osservazione)).
  %  sa_o_assume(P, Oss) : l'agente sa o assume che in P valga Oss, in assenza di conoscenza contraria
  %  MODO (++,--) nondet

negazione(A, non(A)) :- A \= non(_).
negazione(non(A), A) :- A \= non(_).

impara(P, Oss) :-   not(sa(P,Oss)), assert(sa(P,Oss)).

sa_o_assume(P, Oss) :-
        puo_assumere(P, Oss),
        negazione(Oss, NonOss), not(sa(P, NonOss)).

dimentica_tutto :- retractall(sa(_,_)).
```

I due predicati principali sono impara e sa_o_assume; con impara(P,Oss) si aggiunge nuova conoscenza (con assert(sa(P,Oss)), mentre sa_o_assume(P, Oss) riguarda le assunzioni che l'agente fa in fase di decisione o pianificazione, anche se non sa se vale Oss o meno. Nella implementazione del predicato aperto osserva va modellizzata la capacità di osservazione dell'agente, mentre puo_assumere deve indicare quali proprietà del mondo possono essere assunte per vere in caso di mancanza di conoscenze.

Vediamo come implementare i predicati aperti nella realizzazione dell'agente raccoglitore, che inzialmente non conosce la posizione degli ostacoli. In fase di pianificazione assume che non vi siano ostacoli e calcola il piano migliore per raccogliere gli oggetti in assenza di ostacoli. Poi, in esecuzione del piano, man mano che procede osserva le caselle circostanti e, usando il predicato dinamico sa, ricorda le posizioni libere e gli ostacoli che ha osservato. Se a un certo punto trova un ostacolo in P2 che gli impedisce di eseguire l'azione pianificata va(P1,P2), passa in stato di fallimento, indicando come causa l'ostacolo in P2. La decisione successiva sarà di aggirare l'ostacolo in P2. L'aggiramento viene realizzato con un comportamento reattivo basato sull'aggiramento degli ostacoli in senso

antiorario visto per l'agente reattivo aggiratore. Durante ogni esecuzione l'agente impara, cioè ricorda quanto ha osservato. Eseguendo più volte con obiettivi diversi alla fine raggiungerà una conoscenza completa del mondo. Il codice è il seguente.

I tipi aperti sono implementati come segue:

```
type([raccolto]:obiettivo).
   %  vi è un unico obiettivo, aver raccolto tutto
type([vai, fine]:stato_agente).
  %  vai : stato esecuzione normale;   fine : stato finale
type([in(point), deve_prendere(point), deposito(point)]:fluent).
  %   in(P):  l'agente si trova in P ;  deve_prendere(X): l'agente deve prendere un oggetto in X
  %   deposito(Q):  deve depositare gli oggetti in Q
type([va(point,point), va_da_a(point,point), raccoglie, deposita]:action).
  %  va(P1,P2) : avanza da P1 a P2 adiacente;  va_da_a(P1,P2) : va da P1 a P2 non adiacente, macro-azione
  %  raccoglie:  raccoglie l'oggetto; deposita: deposita gli oggetti
```

Lo stato dell'agente situato nel mondo è rappresentato usando i predicati dinamici che corrispondono ai fluenti:

```
pred(in(point)).
:- dynamic(in/1).

pred(deve_prendere(point)).
:- dynamic(deve_prendere/1).

pred(deposito(point)).
:- dynamic(deposito/1).
```

Con questa rappresentazione, il predicato che fa partire l'agente in un mondo W dallo stato iniziale in cui l'agente si trova in Start, deve raccogliere gli oggetti nei punti elencati nella lista PuntiRaccolta e deve depositarli in D è il seguente:

```
raccolta(W, Start, PuntiRaccolta, D) :-
      maplist(retractall, [in(_), deposito(_), deve_prendere(_)]),
      assert(in(Start)),  assert(sa(Start,non(ostacolo))), % non ci possono essere ostacoli
      assert(deposito(D)), assert(sa(D,non(ostacolo))), % nelle posizioni dell'agente, del deposito
      forall(member(P,PuntiRaccolta),                    % e nei punti di raccolta
        (assert(deve_prendere(P)), assert(sa(P,non(ostacolo))))),

      load_current_world(W),  %  disegno il mondo e il suo stato
      draw_agent_state(W),

     set_strategy(astar),   % fisso la strategia per la ricerca dei piani
     set_strategy(ps(closed)),

     osserva_in(Start),       % faccio le osservazioni nella posizione di start
     agente(vai, raccolto).  % e faccio partire l'agente con stato interno "vai" e obiettivo "raccolto"
```

In questo esempio la strategia viene scelta nel codice e non è lasciata all'utente. Le decisioni sono prese dall'agente come segue:

```
decidi(vai, _, esecuzione(deposita)) :-
        % se non c'è più nullada prendere e mi trovo nel punto di deposito, decido l'esecuzione dell'azione "deposita"
         not(deve_prendere(_)), in(Q), deposito(Q).

decidi(vai, _, piano(raccogli(P, OggettiDaPrendere, Q))) :-
        %  se mi trovo in P e ho da prendere e depositare degli oggetti, decido di trovare ed eseguire un piano per
        %  prendere e depositare gli oggetti
        in(P),  setof(deve_prendere(X), deve_prendere(X), OggettiDaPrendere), deposito(Q).

decidi(failed([va(P,Q),va_da_a(_P1,P2)], ostacolo_in(Q), _Piano), _, esecuzione(aggira_ostacolo(P,P2))).
        %  se nella esecuzione del piano per la macro-azione va_da_a(_P1,P2) un passo va(P,Q) incontra un ostacolo
        %  imprevisto, la decisione è di aggirarlo, andare in P2 e riprendere il piano da P2
```

Si osservi che ***nel caso di stato di fallimento*** si ha un ***aggiustamento*** del piano interrotto: l'esecuzione di aggira_ostacolo realizza un comportamento reattivo basato sull'aggiramento antiorario degli ostacoli che consente all'agente di completare [nell'ipotesi di ostacoli convessi] l'esecuzione della macro-azione va_da_a(_P1,P2). Raggiunto P2 termina la macro-azione e l'agente ***ripianifica a livello 0***, trovando un nuovo piano per proseguire la raccolta; il piano può essere diverso da quello interrotto, poiché nella pianificazione ora l'agente ha più conoscenze. In caso di ostacoli concavi, il comportamento reattivo può cadere in trappole, per cui bisognerà sostituirlo con una ***ri-pianificazione di livello 1*** del percorso da P a P2. Tale ri-pianificazione avviene nel nuovo stato di conoscenza dell'agente. Per sfruttare la conoscenza acquisita potrebbe anche convenire una ***ri-pianificazione di livello 0***, ovvero una nuova pianificazione della raccolta degli oggetti ancora da raccogliere.

L'unica macro-azione è va_da_a. L'esecuzione delle azioni base si basa sulla esecuzione dei passi elementari:

```
fai_un_passo(P,Q) :- % eseguo e simulo il passo da P a Q adiacente
       retract(in(P)), assert(in(Q)),  simula(va(P,Q)),
       osserva_in(Q).
```

Ad ogni passo l'agente osserva le posizioni adiacenti alla nuova posizione Q (osserva_in(Q)) e quindi, alla prossima mossa, sa già se ci sono ostacoli. Se sa che non ci sono ostacoli, esegue la mossa, altrimenti passa in stato failed(ostacolo_in(P2)), con causa "ostacolo_in(P2)". Il codice è:

```
esegui_azione_base(_Env, va(P1,P2), vai, vai) :-  % se so che non c'è ostacolo vado e resto in stato interno vai
     sa(P2, non(ostacolo)), fai_un_passo(P1,P2).

esegui_azione_base(_Env, va(P1,P2), vai, failed(ostacolo_in(P2))) :-
        % se so che c'è ostacolo, non mi muovo e passo in stato failed(ostacolo_in(P2))
        sa(P2, ostacolo),  simula(failed(va(P1,P2), ostacolo_in(P2))).
```

Le azioni base raccoglie e deposita sono:

```
esegui_azione_base(_Env, raccoglie, vai, vai) :-  % raccolgo nella posizione P in cui sono e resto in stato interno vai
        in(P), retract(deve_prendere(P)), simula(raccogli(P)).

esegui_azione_base(_Env, deposita, vai, fine) :- % deposito nella posizione in cui sono e passo in stato fine
         in(P), simula(deposita(P)).
```

L'azione aggira viene attivata in stato di fallimento e torna in stato vai; non si tratta di una azione elementare, ma nella realizzazione di un comportamento reattivo che completa l'esecuzione della macro-azione in corso (vedi codice di aggira_ostacolo):

```
esegui_azione_base(_Env, aggira_ostacolo(P,G), _Failed, vai) :-
        aggira_ostacolo(P,G).
```

Per la parte restante della implementazione dell'esecuzione di un piano rimandiamo al codice nel file **agente.ci.pl**.

Per quanto riguarda la pianificazione le azioni di livello 0 sono identiche a quelle viste per il raccoglitore_gerarchico. Cambia l'azione di livello 1 va(P1,P2), dal momento che l'agente pianifica in base a ciò che sa o assume sul mondo. Il codice è:

```
two_level_planner:add_del(1,va(P1,P2), St, [in(P2)], [in(P1)], Cost) :-
        member(in(P1), St),  step(Dir,P1,P2), % P2 raggiungibile in un passo in una direzione Dir
        sa_o_assume(P2, non(ostacolo)),  % sa che non c'è un ostacolo o assume che non ci sia
        length_step(Dir,Cost).
```

Se in pianificazione l'agente ha assunto che in una posizione P non ancora esplorata non ci sono ostacoli ma in fase di esecuzione del piano incontra un ostacolo eseguendo l'azione va, passa in stato di fallimento e si avvia il processo di interruzione che porta ad aggiustare il piano mediante aggira_ostacolo, come visto in precedenza.

Per quanto riguarda la gestione della conoscenza, i tipi e predicati aperti sono implementati come segue:

```
type([ostacolo]:osservabile).  %  l'agente è interessato solo agli ostacoli

osserva(P, Oss) :- current_world(W),
    % osserva se c'è un ostacolo o meno nel punto P del mondo corrente dell'agente
   (   content(W, P, ostacolo) ->  Oss=ostacolo   ;   Oss=non(ostacolo)).

puo_assumere(_, non(ostacolo)).  %  l'unica assunzione che questo agente può fare è che non ci sia un ostacolo

pred(osserva_in(point)).   %  osserva_in(P):  l'agente impara osservando tutte le posizioni adiacenti che non conosce
osserva_in(P) :- forall(step(_Dir, P, Q),
            ( sa(Q,_), !   %sa già le proprietà osservabili di Q
            ; osserva(Q,Oss), impara(Q,Oss), simula(impara(Q,Oss)))).  % oppure le impara
```

Siccome si tratta di un agente che impara, si è deciso che ad ogni esecuzione vengano puliti i predicati relativi allo stato iniziale dell'agente, ma venga conservata la conoscenza acquisita nelle esecuzioni precedenti. Pertanto nel codice del predicato di avvio

```
raccolta(W, Start, PuntiRaccolta, D) :-
      maplist(retractall, [in(_), deposito(_), deve_prendere(_)]),
      ……..
```

vengono azzerati in(_), deposito(_), deve_prendere(_,_), ma non sa(_,_) che conserva quanto imparato in precedenza.

Da ultimo, vediamo come lo stato di conoscenza dell'agente sia rappresentato e come tracciare l'esecuzione nella interfaccia utente. Per "disegnare" lo stato di conoscenza si usa draw_knowledge e per "disegnare" lo stato corrente dell'agente nel mondo si usa draw_agent_state:

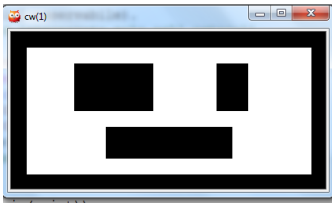
```
draw_knowledge(W, P, Size) :-                              % disegno la conscenza di una casella
     sa(P,non(ostacolo)) ->  true                                % caselle che sa libere bianche
    ;  (  sa(P,ostacolo) ->   draw_fig(W, box(Size,[col(black)]), P)    %  caselle che sa occupate nere
        ;  draw_fig(W, box(Size,[col(grey)]), P).                 %  caselle che non conosce grigie

draw_agent_state(W) :-     % "disegno" i predicati dinamici di stato del mondo+agente
       in(Pos), draw_fig(W, circ(15,[col(green)]), Pos), % cerchio verde l'agente
        forall(deve_prendere(PR), draw_fig(W, box(15,[col(blue)]), PR)), % quadrati blu gli oggetti da prendere
        deposito(Q),  draw_fig(W, circ(15,[col(yellow)]), Q).  % cerchio giallo il deposito
```

Con questa rappresentazione la simulazione visiva delle azioni è realizzata come segue:

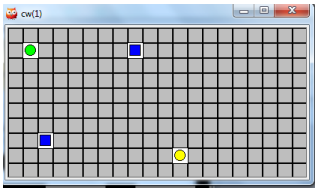
```
simula(va(P1,P2)) :- !,
        %  mossa di 1 casella del cerchio che rappresenta l'agente
        current_world(W), move_fig(W,circ(_,_),P1,P2), step([]).

simula(raccogli(P)) :-!,
        %  il quadrato che rappresenta l'oggetto raccolto viene cancellato
        current_world(W), del_fig(W, box(_,_), P), step(['Raccolto oggetto']).

simula(deposita(P)) :-!,
          %  con il deposito degli oggetti raccolti la simulazione finisce
          step(['Depositati gli oggetti in ',P]), current_world(W), destroy(W).

simula(impara(P,ostacolo)) :-!,
          %  la casella prima grigia (cioè non conosciuta) diventa nera (imparata la presenza di un ostacolo)
           current_world(W), del_fig(W, box(Size,[col(grey)]), P), draw_fig(W, box(Size,[col(black)]), P),
           step(['imparato ostacolo in ', P]).

simula(impara(P, non(ostacolo))) :-!,
          %  la casella prima grigia (cioè non conosciuta) diventa bianca (imparata l'assenza di un ostacolo)
          current_world(W),  del_fig(W, box(_,[col(grey)]), P),  step(['imparato ', P, ' libera']).
```

Per ulteriori dettagli sul codice si veda **agente.ci.pl**. Qui concludiamo con alcune schermate che mostrano come viene simulato graficamente il comportamento dell'agente con query iniziale

?- raccolta(cw(1), point(2,2),[point(2,9), point(8,3)],point(9,12)).

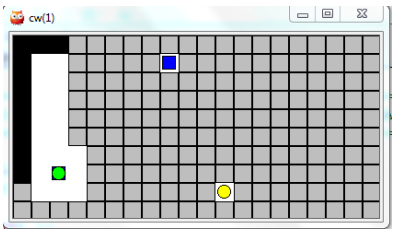


Il mondo cw(1) in cui si muove l'agente

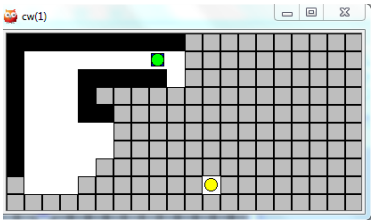


1. Lo stato di conoscenza iniziale. L'agente sa che le posizioni sua (verde), dei due oggetti da raccogliere (blu) e di deposito (gialla) sono libere (bianche), le altre non sono note (grigie)

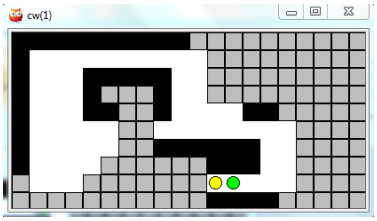
In questo stato di conoscenza l'agente assume che tutte le caselle siano libere, per cui il piano ottimo è andare a prendere l'oggetto in basso, poi quello in alto e infine depositare. Man mano che procede con questo piano impara, come illustrato nelle figure che seguono.



2. Nel percorso fino al primo oggetto ha imparato che le posizioni in nero sono ostacoli e quelle in bianco sono libere.

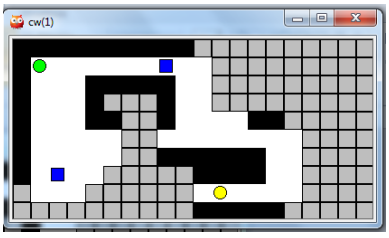


3. Andando dall'oggetto in basso a quello in alto incontra e aggira un ostacolo; la sua conoscenza è ora quella disegnata qui accanto.

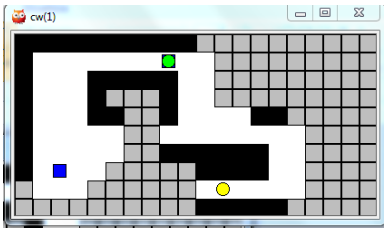


4. Andando a depositare incontra un altro ostacolo (in basso) e lo aggira; la sua conoscenza è aumentata.

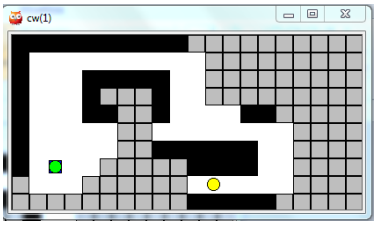
A questo punto, in pochi passi l'agente termina. Se lo rilanciamo con la stessa query
?- raccolta(cw(1), point(2,2),[point(2,9), point(8,3)],point(9,12)).
l'agente parte dall'ultimo stato di conoscenza raggiunto, in quanto sa(_,_) non viene azzerato.



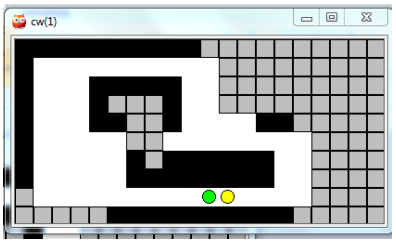
Dunque l'agente deve prendere e depositare i due oggetti a partire dallo stato di conoscenza qui accanto. Con questa conoscenza valuta che il piano migliore non è più andare prima all'ostacolo in basso, ma a quello in alto



Raggiunto l'ostacolo in alto, va a prendere quello in basso eseguendo il piano senza incontrare ostacoli imprevisti (il percorso pianificato avviene tutto nello spazio bianco)



Raggiunto l'ostacolo in basso, esegue il piano per andare al deposito; nel percorso pianificato ci sono caselle grigie, che l'agente ignora ma ha assunto libere



Eseguendo il piano per andare al deposito incontra e aggira l'ostacolo in basso; lo stato di conoscenza finale è mostrato qui accanto.

APPENDICE. Alcuni aspetti generali.

Rappresentazione dello stato "esterno" (mondo+agente). Si è scelto di rappresentare lo stato "esterno" mediante predicati dinamici per le proprietà che possono variare; dunque abbiamo una base di conoscenza KB = KBstatica KBdinamica. In ogni istante, il modello minimo della KB rappresenta uno stato del mondo esterno e l'esecuzione di una azione viene implementata mediante retract delle proprietà che diventano false e assert di quelle che diventano vere e visualizzata mediante stampe o, a livello grafico, modifiche della figura che rappresenta il mondo. Per operare correttamente mediante assert e retract bisogna ricordare quanto segue;

retractall(G) cancella dalla KBdinamica tutte le clausole G' che unificano con G; retractall non fallisce mai, anche se non ci sono G' che unificano.

retract(G) cancella dalla KBdinamica la prima clausola G' che unifica con G; cancella una sola copia e, se non trova almeno una G', fallisce.

assert(G) inserisce G nella KBdinamica; per governare l'ordine delle clausole nella KBdinamica si usi asserta(G) per aggiungere in testa (l'ultima asserta è la prima della KBdinamica) e assertz(G) per aggiungere in fondo.

Stato interno dell'agente. Non rappresenta lo stato del mondo esterno, per il quale usiamo la KBdinamica, ma uno stato interno dell'agente. Per lo più è legato alla attività che in quel momento l'agente sta svolgendo (ad es., se sta trasportando un oggetto si trova in stato "trasporto") o tiene traccia di eventi precedenti (ad es. una azione è fallita nell'esecuzione di un piano).

Stati di pianificazione. Rappresentano gli *stati possibili* del mondo e costituiscono i nodi dello spazio degli stati in cui avviene la ricerca del piano. Usando STRIP, i fluenti rappresentano proprietà del mondo e vi è un legame con i predicati della KBdinamica: un fluente F è sintatticamente identico uno dei predicati dinamici della KBdinamica, ma viene usato in modo diverso; mentre come predicato dinamico rappresenta una proprietà attuale del mondo, come fluente è un dato usabile negli strips_states.

Consideriamo ad esempio l'agente raccoglitore: i predicati dinamici in(point), deve_prendere(point), deposito(point) sono anche fluenti.

- Il contenuto della KBdinamica:
  in(point(5,4)), deve_prendere(point(5,4)), deve_prendere(point(12,11)), deposito(point(2,2)
  rappresenta lo stato corrente del mondo: l'agente *in questo momento* si trova *veramente* in (5,4), deve prendere gli oggetti in (5,4) E (12,11) e andare a depositare in (2,2).
- Lo strips_state
  [in(point(5,4)), deve_prendere(point(5,4)), deve_prendere(point(12,11)), deposito(point(2,2))]
  ha lo stesso significato, ***ma non corrisponde allo stato corrente del mondo***; rappresenta un ***ipotetico stato*** nello spazio di ricerca.

Il fatto che fluenti e predicati dinamici abbiano lo stesso significato ha due conseguenze utili:

- I fluenti della DEL_LIST di una azione corrispondono alla retract dei corrispondenti predicati dinamici, mentre i fluenti della ADD_LIST corrispondono alla assert.

- Se voglio lo strips_state che corrisponde allo stato attuale dell'agente uso la findall; ad esempio il seguente codice restituisce lo StripsState corrispondente allo stato attuale dell'agente raccoglitore:

```
fluent(in(_)).
fluent(deve_prendere(_,_)).
fluent(deposito(_)).
get_current_state( StripsState) :-
    findall(F,  (fluent(F), F),  StripsState).
```

PARTE II. Alcune idee di progetto.

Tipo A. Sviluppare un agente pianificatore che affronti un problema di pianificazione in un tipo di ambienti complesso. I punti chiave sono: a) dare un modello del mondo adeguato (si tratta di fare le astrazioni giuste); b) studiare euristiche adeguate c) sperimentare in più mondi e con diversi stati, confrontando le euristiche [si lavorerà con le statistiche attivate,ds_on].

Ad esempio un agente che deve raggiungere una data posizione in un mondo con diversi tipi di terreno (bosco, palude, strade percorribili in macchina o fuori-strada, ecc.), che può portare con se diversi equipaggiamenti (ad es. sci per essere più rapido sulla neve) e può trovare in date posizioni diversi mezzi (bici, auto, barca, …). I costi sono i tempi, che dipendono dal tipo di terreno, dall'equipaggiamento e dai mezzi. Si tratta di trovare il piano migliore per raggiungere l'obiettivo. Ad esempio se la strada più breve non passa da postazioni da cui prelevare una automobile, può convenire allungare, prendere l'auto e poi procedere più velocemente.

Oppure un chiamabus che deve passare a raccogliere i clienti in diverse posizioni della città e portarli all'aeroporto in uno o più viaggi (se è pieno deve fare un viaggio in più). Se il grafo delle strade porta a considerare piani molto lunghi, può convenire la pianificazione gerarchica.

Tipo B. Considerare un agente che si muove in un mondo che conosce solo parzialmente (conoscenza incompleta). L'agente ha come obiettivo raccogliere più oro che può, oppure salvare la principessa, oppure vedete voi. Il mondo in cui si muove non è popolato solo da ostacoli, ma anche da agenti nemici; la presenza di un nemico può essere percepita dal nostro agente quando si avvicina abbastanza (ad es. il mostro puzza e si sente la puzza entro 2 caselle). Un nemico può impedire il passaggio attraverso una strada, a meno che non venga ucciso. L'agente ha dei punti vita e può uccidere il mostro solo se ne ha abbastanza. Studiate voi altre regole del gioco.

Nella progettazione di un agente di questo tipo si hanno un livello strategico e uno tattico. Il livello strategico è dato dal predicato decidi, ovvero dalle decisioni che man mano prende. Diversi criteri decisionali possono portare a vantaggi strategici, ovvero vantaggi valutabili sull'intera vita dell'agente. Ad esempio si possono avere strategie offensive in date condizioni e difensive in altre. Oppure in caso di ignoranza di grandi porzioni del mondo può pagare una strategia esplorativa.

Una tattica ha vantaggi valutabili sul breve periodo, tipicamente nel calcolo di un piano. Ad esempio, se l'agente non vuol essere visto, un tattica è passare attraverso i boschi; per ottenere un piano che privilegia i boschi si agisce sul ***costo tattico*** delle mosse; un passo attraverso un bosco ha costo tattico basso, mentre una posizione scoperta ha un costo tattico alto. Si possono anche mescolare i costi tattici e i tempi di percorrenza.

Altro aspetto riguarda le assunzioni dell'agente in caso di ignoranza. Difficilmente le assunzioni hanno valore tattico, mentre possono avere valore strategico; ad esempio, meglio essere ottimisti (assumo che ci sia bosco in cui nascondermi, per cui do costo tattico basso alle posizioni che non conosco) o pessimisti (assumo che per lo più passo allo scoperto, per cui do costo tattico alto alle posizioni che non conosco)? Infine una forma di apprendimento consiste nel valutare la probabilità di trovare bosco o meno in base all'esperienza passata; assumendo una distribuzione mediamente uniforme delle zone boschive, posso usare il rapporto fra i quadretti in cui ho trovato bosco e il totale dei quadretti visitati come probabilità di trovare un quadretto boschivo e usarla per valutare il costo tattico.

NOTA. Lo stesso agente può implementare comportamenti diversi a seconda delle circostanze; in alcune si comporta in modo reattivo, in altre decide e pianifica, in altre ancora ha un comportamento reattivo con stato interno.